OFFENSIVE AUTOMOTIVE CYBERSECURITY

An engineering handbook for exploiting modern automotive platforms



DR. AHMAD MK NASSER
DR. DENNIS KENGO OKA

‹packt›

Offensive Automotive Cybersecurity

An engineering handbook for exploiting modern automotive platforms

Dr. Ahmad MK Nasser
Dr. Dennis Kengo Oka

‹packt›

Offensive Automotive Cybersecurity

Copyright © 2026 Packt Publishing
All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.
Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Vijin Boricha
Relationship Lead: Prachi Sawant
Program Manager: Ankita Thakur
Project Manager: Gandhali Raut
Content Engineers: Swathi Ajayakumar, Tazeen Shaikh
Technical Editor: Nithik Cheruvakodan
Indexer: Hemangini Bari
Proofreaders: Swathi Ajayakumar, Akanksha Gupta, Shubhra Mayuri, Tazeen Shaikh
Production Designer: Vijay Kamble
Growth Lead: Ankita Thakur

First published: May 2026

Production reference: 1280526

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83664-863-5
www.packtpub.com

To the memory of the children lost to war, and to the children whose lives were uprooted by it.

May we one day unite as a human family to protect every child beyond borders, nations, and tribes.

— Dr. Ahmad MK Nasser

To my loving wife, Mai, for your endless patience and support, and to my beautiful daughters, Mia, Elina, and Alyssa, for filling every day with laughter and happiness.

— Dr. Dennis Kengo Oka

Contributors

About the authors

Dr. Ahmad MK Nasser is a Lead Security Architect and Engineering Manager at NVIDIA, where he works on DRIVE OS, the software stack powering autonomous driving applications. His team develops advanced security architectures designed to strengthen the resilience, integrity, and availability of autonomous driving systems. Ahmad's current work focuses on holistic elimination of security vulnerabilities by deploying defensive AI across the software development lifecycle. Over the years, he has coached numerous engineering teams on securing mission-critical systems and building security into complex embedded and automotive platforms. He holds a PhD in Computer Science, an MS in Computer Engineering, and a BS in Electrical Engineering.

I want to thank the security research community for advancing our collective knowledge of how to build resilient, mission-critical systems. I am also deeply grateful to my family for giving me the inspiration to persevere when the challenges of research and writing felt overwhelming.

Dr. Dennis Kengo Oka is the Global Technical and Cybersecurity Advisor at IAV, where he helps OEMs and suppliers build security into software-defined vehicle platforms. An automotive cybersecurity executive and strategist with more than 20 years of global experience, he holds a PhD in automotive security from Chalmers University of Technology in Sweden. He has held senior roles at Volvo, Bosch Group (ESCRYPT), and Synopsys. He serves on the advisory board of Block Harbor and has authored more than 80 publications, including *Building Secure Cars* (Wiley, 2021) and *Building Secure Automotive IoT Applications* (Packt, 2024). He is a frequently invited speaker at international automotive and cybersecurity conferences.

I am grateful to the team at Packt for their support and guidance throughout this project. Working with my co-author Ahmad MK Nasser has been a true pleasure, and I deeply appreciate the knowledge and dedication he brought to this collaboration. I also want to thank my parents, Sven and Etsuko, and my siblings, Alex and Linda, for their constant encouragement and for always providing an escape back to what truly matters in life.

About the reviewers

Ayyappan Rajesh is an Offensive Security Engineer at Block Harbor Cybersecurity and a Cyber Fellow at NYU. His core technical expertise spans cybersecurity, wireless exploitation, embedded systems penetration testing, critical infrastructure, and IoT.

His first vulnerability disclosure, presented at DEF CON, led to his involvement with the Car Hacking Village, where he now participates as a member of the community. He has since published multiple high-impact CVEs across the automotive and IoT sectors. His ongoing research initiatives focus on wireless and cyber-physical systems security.

He holds a B.S. in Computer Engineering from the University of Massachusetts Dartmouth. He can be reached through his website at `https://ayyappan.me`.

Jerin Sunny is an Offensive Automotive Cybersecurity Specialist with close to six years of experience working across Tier-1 and Tier-2 automotive suppliers. He focuses on penetration testing and security assessments of embedded systems, IoT devices, ECUs, and full vehicle platforms.

He has a keen interest in hardware security, particularly in areas such as fault injection attacks and embedded security analysis. His work also spans in-vehicle network penetration testing, wireless security assessments, and reverse engineering of modern automotive and embedded platforms.

Jerin holds a master's degree in cybersecurity and actively publishes technical security research, write-ups, and practical findings on his blog at `https://jerinsunny.github.io/blogs`. When he's not breaking into cars, he's out driving them.

Table of Contents

Preface **xxvii**

Free benefits with your book • 0

Part 1: Reviewing the Basics and the Fundamentals of Offensive Security **1**

Chapter 1: Offensive Security Basics **3**

Motivation for offensive security **4**

Anatomy of vehicle E/E architecture **5**

Vehicle architecture types • 6

Domain-based architecture • 6

Centralized zonal-based architecture • 8

External vehicle interfaces • 11

Electronic control units (ECUs) • 11

Sensors • 12

Networking protocols • 13

Local interconnect network (LIN) • 13

Controller area network (CAN) • 13

Ethernet • 14

Understanding the operational environment of connected vehicles **15**

Attacker profile **18**

Attacker levels • 19

Script kiddies • 19

Motivated individuals • 20

Criminal groups • 20

Researchers and ethical hackers • 20

Organized crime • 20

Nation-state actors • 21

Psychology of developers • 21
Psychological tactics • 22
Weaknesses, vulnerabilities, and threats 23
Vulnerability scores • 26
Threats • 27
Introduction to penetration testing 28
Leveraging the NIST framework • 29
ISO 21434 and pentesting • 30
Cybersecurity assurance through penetration testing • 32
Types of penetration tests • 35
White box penetration test • 35
Black box penetration test • 36
Gray box penetration test • 37
Penetration testing phases • 37
Planning phase • 38
Discovery phase • 38
Attack Phase • 39
Reporting Phase • 40
Choosing a penetration tester • 40
White hat vs. black hat hackers • 41
Red team vs. blue team • 42
Summary 43
References 44

Chapter 2: Penetration Testing Phases 47

Planning the penetration test 48
Defining the engagement • 48
Defining the scope • 49
Defining testing conditions • 49
Information sharing and test types • 50
Legal considerations and documentation • 51
Prioritizing components for testing • 52

Defining deliverables and reporting • 52
Discovery of security weaknesses 53
Trust mapping in embedded systems • 53
Discovery phase in practice • 55
Passive reconnaissance • 56
Public and commercial information sources • 57
Passive wireless and RF observations • 58
Passive eavesdropping and baseline traffic capture • 59
Active reconnaissance • 59
Building a test bench • 60
Techniques for active reconnaissance • 60
Artifact analysis • 63
Static code analysis and information flow analysis • 63
AI-enhanced SAST • 64
Firmware reverse engineering • 65
Version-based vulnerability scanning • 66
Dynamic analysis and fuzzing • 67
Hardware reconnaissance techniques • 69
Non-invasive attacks • 69
Semi-invasive or invasive fault-injection attacks • 69
Fully invasive and destructive attacks • 69
Navigating dead ends during discovery • 70
Importance of strategic decision making • 71
Attack phase 71
Fundamentals of attack chains • 71
Frameworks for building attack chains • 73
Common attack pattern enumeration • 73
MITRE ATT&CK framework • 74
Lateral movement and pivoting in automotive networks • 76
Attack chains at vehicle vs. ECU level • 77
Attack methods • 78
Reporting vulnerabilities 79

Importance of communication during the penetration test • 79

Communication triggers • 79

Structuring and presenting findings • 80

Securing and storing reports • 81

Post-remediation activities: ensuring lasting security • 81

Summary 82

References 83

Chapter 3: Building the Tool Arsenal 87

Vehicle backend tools 88

External surface discovery and exposure validation • 88

Vehicle companion apps and backend APIs • 89

Interception, inspection, and API workflow reconstruction • 90

Cloud posture assessment and cloud exploitation simulation • 92

Reference implementations and vendor tools • 93

AI-augmented tools • 93

Wireless interface tools 95

Wi-Fi discovery and client-side manipulation • 95

Bluetooth/BLE reconnaissance and experimentation • 96

Cellular lab networks and rogue network testing • 98

RFID/NFC analysis tools • 99

GNSS/GPS simulation and test signal generation • 100

V2X prototyping and message stack tooling • 100

AI-augmented tools • 101

In-vehicle network tools 104

CAN/CAN FD and SocketCAN-based workflows • 104

Automotive Ethernet access and mixed-protocol tools • 107

AI-augmented tools • 108

High-performance and MCU-based ECU tools 110

Hardware interface discovery and debug access • 111

Fault injection and side-channel research tooling (validation) • 112

Firmware acquisition and reverse engineering • 112

Fuzzing and kernel/system stress testing • 114
AI-augmented tools • 114
Vehicle sensor tools 117
RF signal capture, protocol analysis, and injection • 117
AI-augmented tools • 119
How AI is changing penetration testing 121
Summary 122
References 123

Part 2: Attacking the External Interfaces and Vehicle Networks 131

Chapter 4: Attacking the Vehicle Backend 133

Vehicle backend architecture overview 134
Vehicle unlock use case example • 136
Understanding backend connectivity protocols 138
HTTP/REST APIs • 139
WebSockets • 139
SMS and legacy telephony • 139
AMQP, XMPP and CoAP • 139
Diagnostics protocols • 140
MQTT as a de facto standard • 140
Common weaknesses in vehicle backend architectures 141
Insecure API design and input validation weaknesses • 142
Broken authentication and authorization controls • 143
Weak secrets or credential management • 143
Insecure communication • 144
Cloud infrastructure flaws • 144
Supply chain and third-party integration weaknesses • 145
Penetration testing techniques 145
Cloud reconnaissance and misconfiguration exploitation • 146
Communication testing and protocol analysis • 146
Secret discovery and credential abuse • 147

API exploitation and input manipulation • 148
Authentication, authorization and privilege escalation • 149
Supply chain and integration security testing • 149
Case studies of real-life attacks on vehicle backends 150
Case study 1: Exploiting insecure vehicle backend API design • 150
Discovery and reconnaissance • 151
Initial access • 151
Exploitation • 152
Persistence • 152
Impact • 153
Case study 2: Remote vehicle takeover via vehicle backend vulnerability • 153
Discovery and reconnaissance • 154
Initial access • 155
Exploitation • 155
Persistence • 157
Impact • 157
Case study 3: Remote vehicle control through backend compromise • 157
Discovery and reconnaissance • 159
Initial access • 161
Exploitation • 162
Persistence • 164
Impact • 164
Best practices for securing the vehicle backend infrastructure 165
Strong authentication and access control • 165
Securing communication • 166
Input validation and secure API development • 166
Eliminating default credentials and protecting secrets • 167
Segmentation and network security • 167
Monitoring, detection, and incident response • 167
Protecting sensitive data and privacy • 168
Supply chain and third-party management • 168
Defense-in-depth principle • 168

Summary **169**
References **169**

Chapter 5: Attacking the Wireless Vehicle Systems **175**

Overview of wireless vehicle systems **176**
Understanding wireless vehicle systems protocols **177**
Wi-Fi • 180
Bluetooth • 181
Cellular • 182
Radio frequency (RF) • 182
Ultra-wideband (UWB) • 182
Near field communication (NFC) • 183
Global navigation satellite system (GNSS) • 183
Vehicle-to-everything communication (V2X) • 183
Common weaknesses in wireless vehicle systems **184**
Expanding wireless attack surface • 185
Cellular module architecture and trust boundaries • 187
Network isolation and private APN failures • 188
AT command interfaces and unsafe control paths • 188
Software vulnerabilities in wireless-facing components • 189
Weak credentials and hard-coded secrets • 190
Network services and debug interfaces • 190
RF systems and rolling code cryptography • 191
Wireless protocol design limitations • 191
Penetration testing techniques for wireless vehicle systems **192**
Wireless attack surface discovery and validation • 192
Cellular module compromise and privilege analysis • 194
Private APN lateral movement and network abuse • 195
AT command injection and control channel abuse • 195
Finding vulnerabilities in wireless-facing software • 196
Authentication weaknesses and secret extraction • 196
Exploiting network services and debug functionalities • 197

RF and rolling code attacks • 197

Wireless protocol-level attacks • 198

Case studies of real-life attacks on wireless vehicle systems 198

Case study 1: Cloning rolling code remote keyless entry key fobs via global master key exploitation • 198

Discovery and reconnaissance • 199

Initial access • 200

Exploitation • 200

Persistence • 201

Impact • 201

Case study 2: 4G module hacking – attacking the vehicle's external gateway • 201

Discovery and reconnaissance • 202

Initial access • 203

Exploitation • 204

Persistence • 205

Impact • 205

Case study 3: Zero-click remote code execution on infotainment system • 206

Discovery and reconnaissance • 207

Initial access • 209

Exploitation • 209

Persistence • 211

Impact • 211

Best practices for securing wireless vehicle systems..... 212

Treating all wireless interfaces as untrusted entry points • 212

Enforcing strong isolation between wireless domains and vehicle systems • 213

Designing authentication and authorization for adversarial environments • 213

Protecting credentials, keys, and provisioning workflows • 214

Eliminating unnecessary services, legacy control paths and unsafe fallback mechanisms • 214

Assuming partial compromise and designing for containment • 215

Validating all wireless input semantically, not just syntactically • 215

Hardening wireless-facing software and third-party components • 216

Building security into the development lifecycle • 216

Summary **217**
References **218**

Chapter 6: Attacking the In-Vehicle Communications **223**

Overview of in-vehicle communications **224**
Understanding in-vehicle network protocols **225**
Physical layer • 226
Controller area network physical layer (CAN PHY) • 226
Local interconnect network physical layer (LIN PHY) • 226
FlexRay PHY • 227
Automotive Ethernet PHY • 227
Data link layer • 227
CAN data link • 227
LIN data link • 228
FlexRay data link • 228
Ethernet data link • 228
Network and transport layers • 228
CAN network and transport • 228
LIN network and transport • 229
FlexRay network and transport • 229
Ethernet network and transport • 229
Application layer • 229
Unified diagnostic services (UDS) • 230
Scalable service-oriented middleware over IP (SOME/IP) • 230
Universal measurement and calibration protocol (XCP) • 230
Secure onboard communication (SecOC) • 230
Common weaknesses of in-vehicle communications **230**
CAN • 231
Broadcast architecture and lack of source authentication • 232
Lack of encryption and authentication support • 232
No access control • 232
Error frame and bus-off mechanism • 232

ISO-TP upper-layer weaknesses • 233

SAE J1939 upper-layer weaknesses • 233

UDS application-layer weaknesses • 234

XCP communication weaknesses • 235

SecOC implementation weaknesses • 236

Automotive Ethernet • 237

Misconfiguration or lack of network segmentation • 237

DoIP upper-layer weaknesses • 239

SOME/IP middleware weaknesses • 240

Lack of or misconfiguration of TLS • 242

XCP over Ethernet weaknesses • 243

Other common Ethernet weaknesses • 244

LIN • 245

Implicit master trust and lack of authentication • 245

FlexRay • 245

Lack of message authentication • 245

TDMA schedule as an attack primitive • 245

Network topology considerations • 246

Weak network segmentation and lack of enforcement boundaries • 246

Gateway and domain controller trust boundaries • 246

How topology amplifies protocol weaknesses • 247

Penetration testing techniques for in-vehicle networks 248

CAN • 248

Passive bus monitoring and traffic analysis • 248

Message injection and spoofing • 248

Replay attacks • 249

CAN bus fuzzing • 249

Bus-off attack • 249

ISO-TP testing • 250

SAE J1939 transport protocol testing • 250

UDS diagnostic exploitation • 251

XCP testing on CAN • 252

SecOC bypass testing • 252

Automotive Ethernet • 253

Network discovery and traffic capture • 253

DoIP enumeration and exploitation • 254

SOME/IP testing • 254

TLS configuration analysis • 255

XCP over Ethernet testing • 255

LIN • 256

Passive monitoring and frame analysis • 256

Master impersonation and frame injection • 256

FlexRay • 256

Passive traffic analysis and schedule reconstruction • 256

Frame injection and timing manipulation • 257

Case studies of real-life attacks on in-vehicle communications 257

Case study 1: Reconstructing vehicle routes by eavesdropping on CAN bus traffic • 257

Discovery and reconnaissance • 258

Initial access • 259

Exploitation • 259

Persistence • 260

Impact • 261

Case study 2: CAN injection attack to bypass the smart key system for vehicle theft • 261

Discovery and reconnaissance • 262

Initial access • 263

Exploitation • 263

Persistence • 264

Impact • 265

Case study 3: Remote exploitation of a passenger vehicle - from cellular network to CAN bus • 265

Discovery and reconnaissance • 267

Initial access • 268

Exploitation • 269

Persistence • 269

Impact • 270

Best practices for securing in-vehicle communications 271

Implementing message authentication • 271

Applying encryption and transport layer security • 272

Enforcing network segmentation and access control • 272

Securing diagnostic interfaces • 273

Establishing secure development and configuration practices • 273

Enabling continuous monitoring and anomaly detection • 274

Summary 274

References 275

Part 3: Busting the Myth of the Deeper Architecture Layers Security 281

Chapter 7: Attacking MCU-Based Embedded ECUs 283

Overview of MCU-based ECUs 284

Understanding the hardware and software anatomy 287

Hardware components of an automotive MCU • 287

Microcontroller • 288

Memory (Flash, RAM, EEPROM) • 288

Communication interfaces and buses • 288

Direct I/O and sensor interfaces • 289

Internal peripherals • 289

Debug and programming ports • 289

Software architecture and execution model • 290

Bootloaders and boot process • 291

HPSE firmware • 292

AUTOSAR basic components • 293

Calibration and debugging code • 303

Common hardware and software weaknesses in MCU-based ECUs 304

Hardware weaknesses • 304

Unsecured debug/programming interfaces • 306

Memory readout and bypass of protection • 306

Susceptibility to fault injection (glitching) • 307

Side-channel leakage (Timing/Power/EM analysis) • 308

Lack of encrypted memory support • 309

Software weaknesses • 310

Insecure bootloaders and firmware update mechanisms • 310

AUTOSAR-related weaknesses • 312

Lack of software access control for the HPSE • 313

Leftover debug code or backdoors • 314

XCP and legacy protocols in production • 315

Calibratable security • 315

Absent or weak cryptographic protection • 316

Penetration testing techniques and attack chain construction 318

Step 1: Physical reconnaissance and access • 319

Step 2: Firmware extraction or acquisition • 319

Step 3: Reverse engineering and vulnerability discovery • 320

Step 4: Exploitation • 322

Step 5: Escalation and persistence • 323

Step 6: Lateral movement across the vehicle network • 324

End-to-end attack perspective • 324

Case studies of MCU-based ECU compromises..... 324

Case study 1: Bypassing an ECU bootloader password via EM fault injection • 325

Discovery and reconnaissance • 326

Initial access • 326

Exploitation and persistence • 326

Impact • 327

Case study 2: Ways to exploit an insecure bootloader (DEF CON) • 327

Discovery and reconnaissance • 328

Initial access • 328

Exploitation • 329

Persistence • 329

Impact • 330

Case study 3: My car, my keys: obtaining CAN bus SecOC signing keys • 330

Discovery and reconnaissance • 331
Initial access • 332
Exploitation • 333
Persistence • 333
Impact • 333
Best practices for securing MCU-based ECUs 334
Locking down debug and development interfaces • 334
Implementing secure boot and firmware authentication • 335
Enforcing access control and memory protection • 335
Hardening diagnostic and update interfaces • 336
Employing robust cryptographic modules • 336
Integrating fault injection countermeasures • 337
Summary 337
References 338

Chapter 8: Attacking the High-Performance ECUs 343

Introducing high-performance ECUs 344
Understanding the hardware and software anatomy 345
Hardware components of HPCs • 345
Automotive SoC and companion MCU • 345
Memory and external storage • 345
Connectivity interfaces and peripheral buses • 346
Camera and sensor interfaces • 346
Software architecture of HPCs • 347
Bootloaders and hypervisors • 347
Guest operating systems and software partitioning • 348
Safety-critical domains • 348
Feature-rich and connectivity domains • 348
Mixed-criticality execution model • 348
MCU software domains • 349
Inter-process and inter-VM communication • 349
Network communication protocols • 349

Device access, passthrough, and debug tools • 350

Special hardware engine firmware • 350

Identifying common weaknesses in HPCs 350

Hardware weaknesses • 351

Insecure debug and peripheral access • 351

Weak hardware isolation • 352

GPU weaknesses • 353

Side-channel leakage • 354

Cryptographic key handling • 355

CPU vulnerabilities • 356

Susceptibility to fault injection • 357

ROM weaknesses • 358

Software weaknesses • 359

Basic implementation flaws • 359

Understanding integer and buffer overflows • 359

Exploitation • 361

Insecure bootloader and OS boot configuration • 362

Unnecessary services and expanded attack surfaces • 363

Overprivileged services and permission misconfigurations • 363

Android-specific IPC and debug interface vulnerabilities • 364

HPSE firmware weaknesses • 366

Weak inter-VM isolation and resource sharing • 366

Kernel vulnerabilities and outdated software versions • 367

Guest-to-host escape vulnerabilities • 368

Missing or weak authentication checks during updates • 369

Exploring penetration testing techniques 370

Step 1: Reconnaissance and initial access • 371

Step 2: Code extraction and analysis • 372

Step 3: Reverse engineering to uncover software vulnerabilities • 372

Step 4: Exploitation and code execution • 373

Step 5: Privilege escalation in HPC environments • 374

Step 6: Lateral movement across virtual machines • 375

Step 7: Persistence and remote command and control • 375
Advanced red team tactics: supply chain compromise 376
Case studies of publicly disclosed compromises involving HPCs 377
Case study 1: Mazda IVI • 377
Discovery and reconnaissance • 378
Hardware examination • 378
Software composition analysis • 379
Step-by-step breakdown of the attack chain • 381
Case study 2: Tesla IVI • 382
Discovery and reconnaissance • 383
Step-by-step breakdown of the attack chain • 384
Case study 3: Tesla Pwn2Own 2023 • 386
Discovery and reconnaissance • 388
Step-by-step breakdown of the attack chain • 388
Detailed steps of the exploit • 392
Bonus case studies • 394
Best practices for securing HPCs 395
Minimizing the software and interface attack surface • 396
Software bill of materials and vulnerability management • 397
Establishing hardware root of trust and secure boot • 397
Using trusted execution environments and isolated security subsystems • 398
Secure partitioning in hypervisor-based HPC architectures • 399
Memory protection and microarchitectural risk mitigation • 399
Strengthening input handling • 400
Secure in-field updates and supply chain protection • 400
Summary 401
References 402

Chapter 9: Attacking Vehicle Sensors and AI/ML 407

Overview of vehicle sensors and AI 408
Understanding vehicle sensors and AI interfaces 409
ECU-attached and network-attached sensors • 410

Externally mounted sensors • 411

Tire pressure monitoring systems • 411

Camera systems • 412

Radar systems • 413

LiDAR systems • 413

Ultrasonic systems • 414

Architecture of externally mounted sensors • 415

AI and ML systems • 416

ADAS perception and planning • 416

In-vehicle AI assistants • 416

Common weaknesses of vehicle sensors and AI systems 417

Acoustic signal injection weaknesses in MEMS inertial sensors • 418

Unauthenticated RF transmission and vehicle tracking in TPMS • 418

Camera system weaknesses at the physical, interface, and inference layers • 419

Unauthenticated RF reception and waveform predictability weaknesses in radar systems • 421

LiDAR physical-layer and interface weaknesses • 423

Ultrasonic proximity sensor weaknesses due to lack of echo source validation • 424

Asymmetric authentication between sensor data plane and control plane • 425

Trust boundary misconfiguration in domain and zonal architectures • 426

DNN perception system weaknesses at training and model integrity layers • 426

In-vehicle AI assistants weaknesses • 428

Penetration testing techniques against vehicle sensors and AI 429

MEMS inertial sensor testing • 429

TPMS testing • 430

Camera system testing • 431

Radar testing • 432

LiDAR testing • 433

Ultrasonic testing • 434

AI assistant and LLM testing • 435

Case studies of real-life attacks on vehicle sensors and AI/ML systems 436

Case study 1: Adversarial machine learning attacks on Tesla Autopilot • 436

Discovery and reconnaissance • 437
Initial access • 438
Exploitation • 438
Persistence • 441
Impact • 441
Case study 2: LiDAR spoofing via pulse recording and replay • 441
Discovery and reconnaissance • 442
Initial access • 443
Exploitation • 443
Persistence • 444
Impact • 444
Case study 3: Zero-click remote code execution via TPM sensors • 445
Discovery and reconnaissance • 446
Initial access • 447
Exploitation • 447
Persistence • 449
Impact • 449
Best practices for securing vehicle sensors and AI systems 450
Applying defense-in-depth across the sensor and AI stack • 450
Authenticating sensor data at the interface level • 451
Designing AI systems with adversarial robustness in mind • 451
Protecting in-vehicle AI assistants against injection and misuse • 452
Addressing sensor security throughout the development lifecycle • 452
Summary 453
References..... 454

Chapter 10: A Path Forward 459

Ad-hoc security approach 460
Compliance-only mindset • 460
PSIRT-only fallacy • 461
Why ad-hoc approach fails in the modern era • 461
Signs of an ad-hoc culture • 462

Managed security approach 463
Tool management and supply chain realities • 465
Acquisition and supplier management • 467
Planning • 467
Integration with engineering lifecycle • 468
TARA • 468
Need for a cybersecurity concept • 469
Design and implementation • 470
Security design principles and patterns • 472
Secure implementation • 477
Verification testing • 479
Validation testing • 479
Product release • 480
Production planning • 481
Operations and maintenance • 483
End of life and decommissioning • 484
Raising the security bar in automotive systems 485
Process pressures in rapid development cycles • 485
Human resource gap • 486
Technical challenges of centralized architectures • 486
Attacker's landscape • 486
Elevating automotive security • 487
Winning the race 487
From human-limited to autonomous attack scaling • 489
Taxonomy of AI offensive capabilities • 490
LLM-assisted human attackers • 490
Autonomous vulnerability discovery pipelines • 491
Multi-agent offensive systems • 491
Why vulnerability management fails to scale • 493
Vulnerability elimination: A reduction-oriented definition • 493
AI-augmented V-model • 493
Feature-level threat modeling • 494

Architecture-level security review • 495

Implementation-level analysis • 496

Automated fuzzing and white-box testing • 497

Offensive simulation agents • 499

Summary 500

Chapter 11: Unlock Your Exclusive Benefits 503

Unlock this Book's Free Benefits in 3 Easy Steps..... 504

Other Books You May Enjoy 508

Index 511

Preface

The idea for this book was sparked by a conversation I had with a bright colleague who challenged me on the likelihood of a specific attack chain against an automotive ECU. When discussing cybersecurity risks, encountering doubtful minds is not exclusive to the automotive industry. I have heard from numerous friends who work in cloud and enterprise security that they often face skepticism when raising security concerns, especially when the remedy is costly or has an impact on performance or project schedule. But when that colleague shared his concern, it was obvious to me that this was not a simple case of disagreement over attack feasibility, but rather a disconnect with the reality of modern-day cyberattacks.

Prior to that day, I had many conversations with engineers about threats against all sorts of vehicle components, and the one striking theme was always their lack of imagination about what is possible. These engineers, while quite competent in their respective domains, often failed to imagine how a security weakness in their component could ever materialize into a full-blown exploit. To their credit, I too would be highly skeptical if it were not for the hundreds of news stories and papers illustrating how attackers took advantage of seemingly innocuous vulnerabilities to carry out their attack objectives.

Engineers focused on chip design, writing embedded software, or developing vehicle supporting systems may perceive the vehicle as a multi-layered fortress of electronic complexity that no one can penetrate. Heck, engineers can barely get their designs working, so it must be magnitudes more difficult for adversaries to work through that complexity and hack into the vehicle. Since rebutting this argument is not as straightforward as citing a few incidents in which that was not the case, we turn to the discipline of offensive cybersecurity to build our case. We cannot convince someone who has already made up their mind that attacking vehicles is theoretical work from the realm of science fiction rather than reality, but what we intend to do is show the evidence of what is possible and what has already happened and let you decide. While this book is focused on automotive systems, most of the concepts and attacks discussed can easily be applied to IoT systems, as the underlying architecture of edge and cloud computing is gradually converging between the two domains.

Throughout this book, you will learn real-life hacks targeting each layer of vehicle architecture to show how attackers successfully achieved their objectives. As we do so, we hope to convince you that modern vehicles are more like porous membranes, with information flowing in almost every direction, as opposed to isolated islands that are hard to breach.

After introducing the basic concepts of offensive cybersecurity and penetration testing, we will explore each layer of the connected vehicle architecture through case studies that show how attackers managed to bypass multiple layers of security to achieve their objectives, starting with the vehicle backend and ending with deeply embedded electronic control units and sensors. For those engineers who argued that their systems were sufficiently secure because they had built-in first-order security measures, we will see that their results were often no better than those who ignored security altogether. By the end of this book, we hope to convince you that security must be systematically integrated across all vehicle architecture layers, all development processes, and all vehicle lifecycles.

Rather than overwhelming you with technical explanations about coding vulnerabilities and insecure protocols, we will introduce these concepts in the respective chapters where they were exploited in real-life attacks. Before presenting the case studies, we offer a survey of security weaknesses that impact each architecture layer to highlight the concepts and technology aspects that must be understood to fully internalize how the attacks were possible. Also, since offensive security tools are a big part of the domain, we dedicated a full chapter to the topic. This should serve as a useful reference that you can return to when reading case studies or choosing to set up your own penetration testing environment.

In the final chapter, we lay out our vision for the future of defensive automotive cybersecurity, which must leverage artificial intelligence as an enabling technology not only to keep up with offensive AI capabilities, but also to finally eliminate entire classes of security weaknesses.

Who this book is for

This book is for automotive engineers who struggle with understanding how their systems may be hacked. You may have been developing automotive software and hardware that must meet certain cybersecurity requirements but never fully internalized how attackers could penetrate the vehicle to affect your component. You may also be quite familiar with defensive security measures but wonder how to improve the robustness of your security controls through penetration testing. Perhaps, you may be an established penetration tester in enterprise networks who is looking to transition to the domain of pen-testing automotive and IoT systems. Regardless of your background or your objective, this book will walk you through the vehicle architectural layers and highlight security weaknesses that resulted in real-life attacks so you may achieve two things: first understand how to perform effective penetration testing against those systems and second learn how to build more robust security controls.

What this book covers

Chapter 1, Offensive Security Basics, starts by providing the motivation for why offensive security is critical to developing secure automotive systems, dispelling common myths about vehicles being unhackable. To lay the foundation for attacks, it explores the anatomy of modern E/E architectures, as well as external interfaces and networking protocols The chapter also examines attacker motivations and capabilities ranging from script kiddies to nation-state actors. It introduces weaknesses, vulnerabilities, and threats, scoring systems, the basics of penetration testing, contrasting it with real-world attacks.

Chapter 2, Penetration Testing Phases, provides a detailed look at each phase of the penetration testing lifecycle: planning, discovery, attack, and reporting. You will learn how to properly define the engagement scope and rules of engagement, as well as the essential legal and documentation frameworks. The chapter also explores mapping trust relationships and techniques for both passive and active reconnaissance and examines attack chain construction using established frameworks like MITRE ATT&CK and CAPEC. Finally, it will also cover the process of responsibly reporting vulnerabilities, including the structure for findings and post-remediation activities like root cause analysis to ensure lasting security.

Chapter 3, Building the Tool Arsenal, surveys the offensive security tool ecosystem across every layer of vehicle architecture and serves as a reference for tools used in both the discovery and exploitation phases, covering network scanners, binary analyzers, reverse engineering frameworks, and fault injection hardware. It explores specific toolsets for the vehicle backend, wireless interfaces, in-vehicle networks, both MCU and SoC-based ECUs, and vehicle sensors and AI systems. Furthermore, the transformative role of AI in offensive security, highlighting how AI-assisted and AI-driven tools accelerating bottlenecks like binary triage and fuzz harness generation are also discussed. This chapter is meant to be your guide for selecting the proper tool whenever you are attacking a specific aspect of the connected vehicle.

Chapter 4, Attacking the Vehicle Backend, explores one specific aspect of the vehicle architecture and its operational environment and begins by describing vehicle backend cloud architectures, their common building blocks, and connectivity protocols like MQTT and REST APIs. Common weaknesses affecting these systems, such as insecure API design, broken authentication, and cloud infrastructure flaws, are discussed. After outlining penetration testing techniques tailored for the backend, the chapter moves to real-life case studies involving major OEMs and security best practices to fortify backend infrastructure.

Chapter 5, Attacking the Wireless Vehicle Systems, explores the wireless networking interfaces that enable powerful remote attacks, starting by surveying the diverse wireless protocol landscape, then detailing common weaknesses such as unauthenticated RF transmissions, software vulnerabilities in wireless-facing components, and unsafe legacy control paths.

Specialized penetration testing techniques for RF signal capture, protocol analysis, and fuzzing are discussed and to ground these concepts, real-life case studies involving cloning remote keyless entry key fobs, 4G module hacking, and a zero-click Wi-Fi exploit on an infotainment system are studied. The chapter also provides best practices for securing these interfaces, emphasizing isolation, credential protection, and semantic input validation.

Chapter 6, Attacking the In-Vehicle Communications, focuses on the in-vehicle network protocols that form the communication backbone of the connected vehicle. The chapter examines the physical, data link, and application layers of protocols and delves into their inherent weaknesses, including the lack of source authentication and encryption. Penetration testing techniques, demonstrating how to perform passive monitoring, message injection, and bus-off attacks are also included. Through real-life case studies, the chapter illustrates the severe consequences of these vulnerabilities. The chapter concludes with best practices for hardening network communications using established security protocols.

Chapter 7, Attacking MCU-Based Embedded ECUs, dives into the offensive security domain of the embedded devices, beginning with an overview of their hardware anatomy and common software stacks like AUTOSAR Classic, followed by examining common hardware weaknesses and software vulnerabilities. The chapter outlines a comprehensive attack methodology, from physical reconnaissance to firmware extraction, escalation, and persistence. Finally, real-world case studies illustrate these techniques, including bypassing an ECU password via EMFI and exfiltrating SecOC keys.

Chapter 8, Attacking the High-Performance ECUs, builds upon the embedded concepts from the previous chapter to explore the larger attack surface of these centralized units and examines the hardware and software anatomy of HPCs and mixed-criticality domains. The chapter identifies critical weaknesses such as weak hardware isolation, GPU vulnerabilities, kernel CVEs, and guest-to-host escape paths, then detail a penetration testing flow that covers code extraction, software composition analysis (SCA), and privilege escalation. Several sophisticated case studies are shared, including a zero-click Bluetooth RCE on an IVI system and a multi-stage Pwn2Own exploit chain.

Chapter 9, Attacking Vehicle Sensors and AI/ML, explores the unique security challenges the components face when vehicles move toward higher levels of autonomy, and how the integrity of sensor inputs and AI systems become paramount to vehicle safety. These are categorized into ECU-attached sensors, network-attached sensors, externally mounted sensors, and AI and ML systems, explaining the operating principles of radar, LiDAR, camera, and ultrasonic systems alongside ADAS perception and planning pipelines and in-vehicle AI assistants. The chapter analyzes weaknesses such as camera saturation, LiDAR spoofing via pulse replay, and prompt injection in AI assistants. It also discusses specialized penetration testing techniques

for generating adversarial ML examples and manipulating RF or acoustic inputs with real-world case studies grounding these concepts.

Chapter 10, A Path Forward, is the final chapter and it pivots from offensive techniques to the defensive domain, charting a forward-looking strategy for automotive cybersecurity. It contrasts reactionary ad-hoc security with a managed approach centered on a Cybersecurity Management System (CSMS). The chapter emphasizes a secure-by-design philosophy, applying core principles like domain separation, least privilege, and defense in depth through established design patterns. It also explores how organizations can leverage Generative AI to move toward structural vulnerability elimination rather than mere reactive patching and discusses maintaining the security posture across the entire lifecycle, from secure production and supply chain protection to post-deployment monitoring and eventual decommissioning.

To get the most out of this book

To make the most out of this book, it is recommended that you are familiar with automotive embedded software, networking protocols, cloud-based development, and vehicle electronic components. We assume that you already understand the fundamentals of coding defects as these form the basis for the security weaknesses that will be exploited. We also expect that you have a background in embedded systems architecture both at the hardware and software levels. Additionally, you will benefit from a basic awareness of the automotive V-cycle and standard engineering management systems.

While this book does not offer step by step instructions on how to carry out attacks, you are encouraged to set up your own environment using a salvaged ECU and install some of the open-source tools cited throughout the book. To do so, you will need a Linux-based environment (preferably Ubuntu or a similar distribution) for example, to take advantage of native SocketCAN support and the can-utils suite. Depending on the layer of the architecture you are testing, you will require specialized hardware, including USB-to-CAN adapters for network access, Software-Defined Radios (SDRs) for wireless and sensor-layer reconnaissance, and J-Link or OpenOCD-compatible debug probes for ECU hardware analysis. For software analysis, ensure you have set up protocol analyzers like Wireshark and reverse engineering frameworks such as Ghidra.

While we have tried our best to share as much context about the attack classes as possible, we realize that attempting to provide a comprehensive list of every security weakness and why it is exploitable would make the book too verbose causing you to lose interest. So, we erred on the side of brevity with the hope that we will spark interests in you to pursue further learning after establishing the foundation in this book. Due to the diversity of topics presented in this book, you are advised to supplement this reading with other resources wherever you see a topic that requires pre-requisites which you currently lack.

The views, opinions, and statements expressed in this book are solely those of the authors and do not necessarily reflect the views, policies, or positions of the authors' current or former employers, affiliated organizations, or any other entities with which the authors are or have been associated.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to *Free benefits with your book* section at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and X handles. For example: "When the user in a mobile app presses the **Unlock Car** button, the backend issues an MQTT publish on topic `vehicles/<VIN>/commands/unlock`."

A block of code is set as follows:

```
"if(password_correct)
proceed;
else reset;"
```

Any command-line input or output is written as follows:

```
#define ITEM_SIZE 64
#define MAX_ITEMS 10
#define BUFFER_SIZE (ITEM_SIZE * MAX_ITEMS)
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "As evident by the list of use cases, these services create opportunities for attackers to cause severe **safety**, **financial**, **operational**, and **privacy** damages if compromised."

> **Note**
>
> Warnings or important notes appear like this.

> **Tip**
>
> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form. All valid errata will be timely updated on GitHub: `https://github.com/PacktPublishing/Offensive-Automotive-Cybersecurity`

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to Unlock*" section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:







How to Unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require one

Share your thoughts

Once you've read *Offensive Automotive Cybersecurity*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



https://packt.link/r/1836648634

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1

Reviewing the Basics and the Fundamentals of Offensive Security

This part establishes the conceptual and practical foundation required for attacking modern connected vehicles. You will gain a clear understanding of automotive electrical/electronic architectures, attacker profiles, and the vulnerability classification frameworks that inform real-world threat modeling. The part then walks through the structured phases of a professional penetration testing engagement before surveying the full offensive tool ecosystem spanning every layer of the vehicle architecture. By the end of this part, you will have the vocabulary, methodology, and tooling awareness needed to approach automotive offensive security engagements with confidence.

This part has the following chapters:

- *Chapter 1, Offensive Security Basics*
- *Chapter 2, Penetration Testing Phases*
- *Chapter 3, Building the Tool Arsenal*

1

Offensive Security Basics

Building security defenses in automotive systems without insights into how these systems can be attacked puts security engineers at a significant disadvantage compared to their adversaries, who spend their lifetimes learning about these defenses and how to defeat them. In this cat-and-mouse game between cybersecurity engineers and hackers, one side spends countless hours designing security defenses, while the opposing side may spend mere hours defeating them.

The asymmetry between cyber defense and offense stems from the basic reality that building anything is far more costly and labor-intensive than destroying it. However, this asymmetry, which clearly favors those who probe and hack, does not mean that those working in cyber defense must accept that their systems will inevitably be breached and resign themselves to a life of defeat. **Offensive security** is both the science and art of discovering vulnerabilities in computer systems and designing exploits that allow the attacker to achieve their objectives.

Before diving into the field of offensive security, we will begin this chapter by reviewing some basic concepts in automotive cybersecurity, which will serve as a foundation for understanding how these systems can be compromised. First, we will lay the groundwork for why offensive cybersecurity matters, then we will review some concepts about the **Electrical/Electronic (E/E)** architecture of modern vehicles within the context of a fully connected environment. We will then explore the attackers and their motivations. Next, we will study some fundamental concepts that will help us in later chapters as we explore the various vulnerabilities and the attack methods used to exploit them to breach automotive systems.

In the rest of this chapter, we will cover the following main topics:

- Motivation for offensive security
- Anatomy of vehicle E/E architecture
- Understanding the operational environment of connected vehicles

- Attacker profile
- Weaknesses, vulnerabilities, and threats
- Fundamentals of attack chains
- Penetration testing basics

Note

Your purchase includes a free PDF copy

Your purchase includes a DRM-free PDF copy of this book, and additional exclusive extras. See the *Free benefits with your book* section in the *Preface* to unlock them instantly and maximize your learning.

Motivation for offensive security

Engineers working in the cybersecurity field typically spend most of their careers analyzing threats and designing cybersecurity controls to mitigate those threats. They proudly consider themselves defensive security engineers, focusing on how the latest security technologies thwart various attacks and strengthen security controls. Defensive security engineers in the automotive industry work tirelessly to ensure that safety-critical automotive systems can continue to operate safely even while under attack.

Some of those vehicles may have limited autonomous driving capabilities, while others may be fully autonomous, with no possibility of driver intervention in case of a safety-impacting breach. While their primary focus is protecting road user safety, their other security objectives include protecting intellectual property and company reputation, which could suffer severe financial damage because of security breaches. Yet another objective is ensuring the availability of vehicle functions to prevent loss of consumer confidence and avoid costly recalls.

When focused solely on defense, those security professionals can easily develop blind spots in understanding how their systems may be attacked. In contrast, **offensive security** takes a different mindset, where one does not assume a security control is effective but rather explores how that control can be defeated to achieve the attacker's objective. Whereas **defensive security** is about building strong barriers, offensive security is about identifying weaknesses and gaps in those barriers. One focuses on perfecting security technologies, while the other focuses on finding and exploiting imperfections.

People working in offensive security dedicate themselves to understanding how systems are built to discover weaknesses and design exploits. It stands to reason that defensive security engineers must also invest time in understanding the attack methods, techniques, and tools that offensive security professionals use to bypass defenses. Without this knowledge, defensive cybersecurity engineers, especially those working on safety-critical systems like those found in modern vehicles, put themselves at a significant disadvantage.

That is why this book aims to provide defensive security engineers with insights into the offensive security domain to enable them to build the correct security measures and ensure those measures remain resilient against cyber-attacks. Another motivation for writing this book is to dispel the myth that modern vehicles are extremely difficult to attack and that successful breaches are purely theoretical. While significant effort has been made to disprove this notion, many automotive engineers still believe that attacks leading to dangerous outcomes in vehicles are unlikely due to the complexity and proprietary nature of these systems.

Throughout this book, we will study real-life attacks that target each layer of vehicle architecture. While doing so, we will see how attackers managed to bypass multiple layers of security to achieve their objectives. For those engineers who argued that their systems were sufficiently secure because they had built-in first-order security measures, we will see that their results were often no better than those who ignored security altogether. But before we can do that, let us first establish the basics about the E/E architecture.

Anatomy of vehicle E/E architecture

When planning an offensive security campaign against any system, the first step is to understand the system makeup, its components, and how it functions. Automotive systems are no different, and rather than focusing on a single vehicle sub-system, it is important to gain a perspective on the entire set of components that make up a connected vehicle and its environment. In this section, we will introduce the main layers of the **E/E** architecture, but we will leave the details of each layer to later chapters when we discuss targeted attacks on those layers.

> **Note**
>
> If you are reading this book as an automotive engineer, we assume that you are familiar with the E/E architecture of modern vehicles, so you may treat this section as a review or completely skip it.

Before delving into the various entry points of the vehicle E/E architecture, it is essential to understand the two most prevalent vehicle E/E architecture types: **domain-based** and **central**

computer-based. The former is a legacy architecture that has evolved over the past few decades, characterized by a physical separation of **electronic control units (ECUs)** into functional networking domains connected by a central gateway. The latter represents state-of-the-art architecture and is considered a key enabler of **software-defined vehicles (SDV)**, where one or more central compute units host numerous software applications, while zone ECUs and classical ECUs handle the bulk of sensing and actuation tasks.

While central computer-based architecture is gaining momentum, both architectures will likely exist for a long time. The distributed domain architecture will remain the choice for many legacy platforms, where business needs do not yet justify the transition to a central computer. Each of these architectures presents its own set of security challenges, making it important to understand their respective weaknesses as a prerequisite to building effective attack chains.

Vehicle architecture types

Modern vehicle E/E architectures are increasingly organized around two major modern variants: the gateway connected **multi-domain-based architecture** and the more **centralized zonal/server-based architecture** with a central computer. In practice, many vehicles use hybrid forms of these approaches. In this section, we will explore each to gain an understanding of how they can be penetrated.

Domain-based architecture

In this architecture, vehicle functions are separated into domains, each of which shares a common set of sensors and actuators. Typically, one would expect to find a chassis domain that manages braking, steering, and airbag control; a powertrain domain that includes engine and transmission control; and a body domain responsible for heating, cooling, seat control, light control, and door entry. An infotainment and telematics domain usually offers the digital cockpit experience and external connectivity. These domains are interconnected via a central gateway that supports multiple communication protocols such as ethernet, controller area network (CAN), and local interconnect network (LIN).

The domain-based E/E architecture, illustrated in *Figure 1.1*, comes in many variants based on the OEM's need to add new functions to various vehicle models.

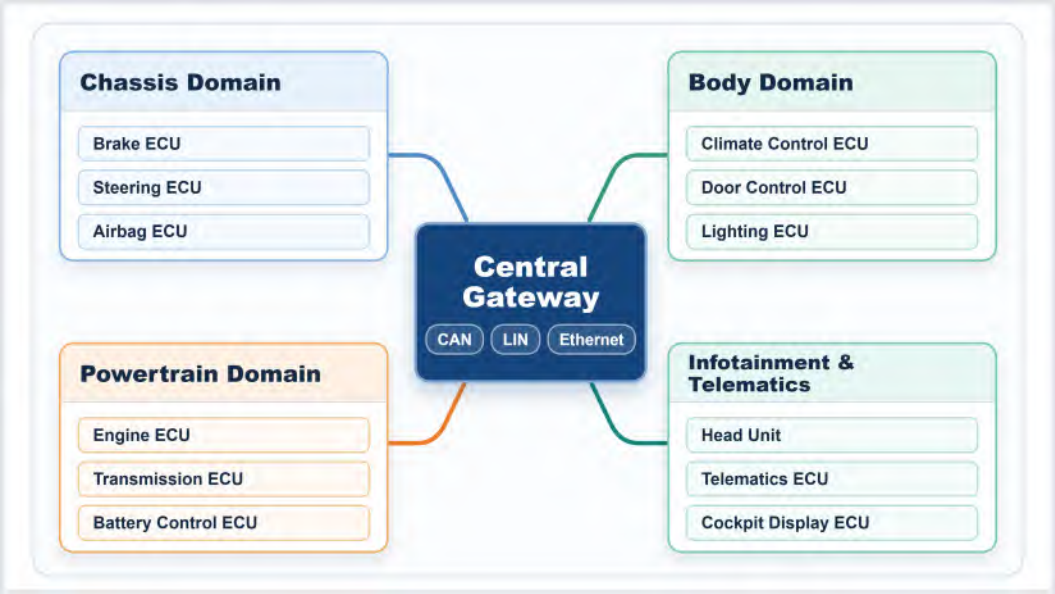


Figure 1.1: Gateway-connected multi-domain-based E/E architecture

As a result, it is common to find sub-bus components where a single ECU acts as a local gateway, integrating additional ECUs into the vehicle network. Ideally, ECUs are cleanly segmented along with vehicle functions, and the gateway provides strong physical and logical isolation. However, it is common to find ECUs with varying degrees of safety and security criticality sharing a network, such as an airbag controller and an infotainment ECU or an engine controller and a telematics unit.

To get a feel for a real-life example of domain-based E/E architecture, you can study *Figure 1.2*, which shows the topology of a 2017 Lexus NX300 [1]. Notice how the central gateway separates the various domains, such as the connectivity domain, body control, chassis, and powertrain.

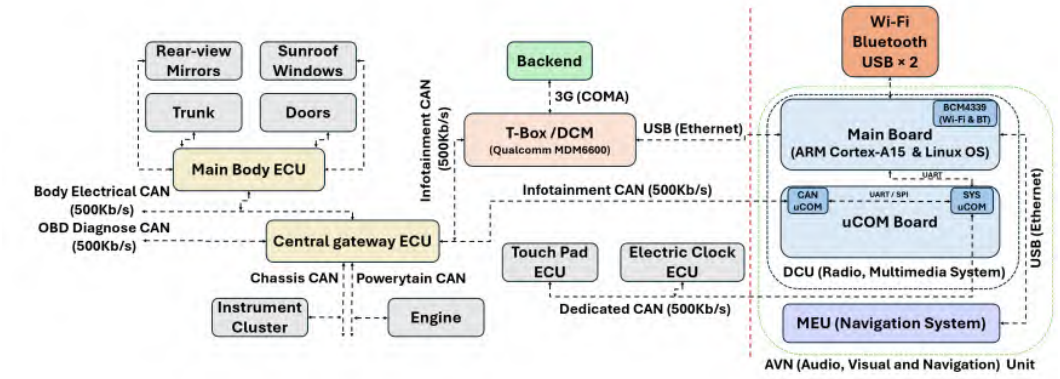


Figure 1.2: A gateway-connected domain-based E/E architecture in a 2017 Lexus NX300 vehicle [1]

Note

When conducting offensive security activities, discovering a single network segment that is shared by a safety-critical ECU and an externally connected ECU should raise an immediate red flag and signal an opportunity for exploitation.

In poorly segmented vehicle domains, attackers may only need to breach a single remotely accessible ECU to directly interact with a safety-critical ECU like the airbag module. In cases where the E/E architecture is well-segmented, the focus of attackers shifts to the gateway ECU. Breaching the gateway becomes crucial for hopping from an external access point, such as the infotainment unit, to the deeper layers of the E/E architecture, as we will see in later chapters.

Centralized zonal-based architecture

Central computer-based vehicle architecture is a hallmark of what is now widely known as the software-defined vehicle. This term simply refers to the trend of migrating vehicle functions from special-purpose ECUs to one or more high-performance computing units that realize the functionality through networked software. These architectures come in many flavors, but we will focus on the one that seems to be gaining the most popularity, consisting of a central computer networked with multiple zone controllers as seen in *Figure 1.3*.

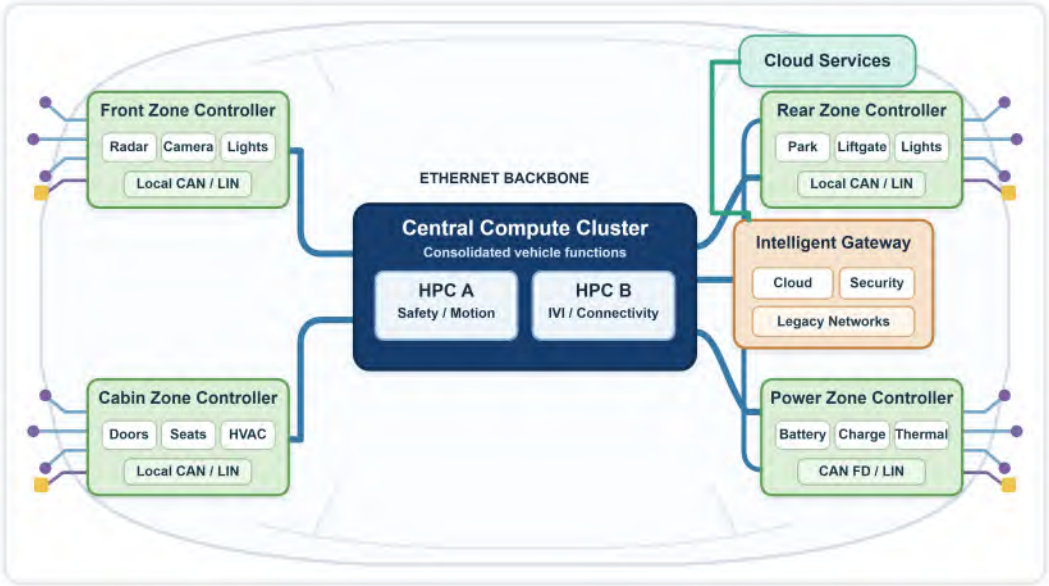


Figure 1.3: Central compute E/E architecture

In this architecture, the central computer platform may consist of one or more System on Chip (SoCs) that host multiple guest operating systems in a virtualized environment. This is enabled by a hypervisor that virtualizes the various hardware components of the system and manages multiple virtual machines. Typically, each virtual machine (VM) hosts a rich operating system on high-performance application central processing units (CPUs), while special-purpose embedded cores run real-time operating systems that manage a suite of special purpose hardware accelerators.

Due to the hyper-consolidation of functions into a single central computer platform, it is not uncommon for OEMs to integrate high-connectivity functions like Bluetooth/Wi-Fi stacks with safety-critical functions like sensor fusion within a single Guest OS. The downside of this approach is that a vulnerability in one of the externally connected applications has a high likelihood of impacting safety-critical applications within the same Guest OS.

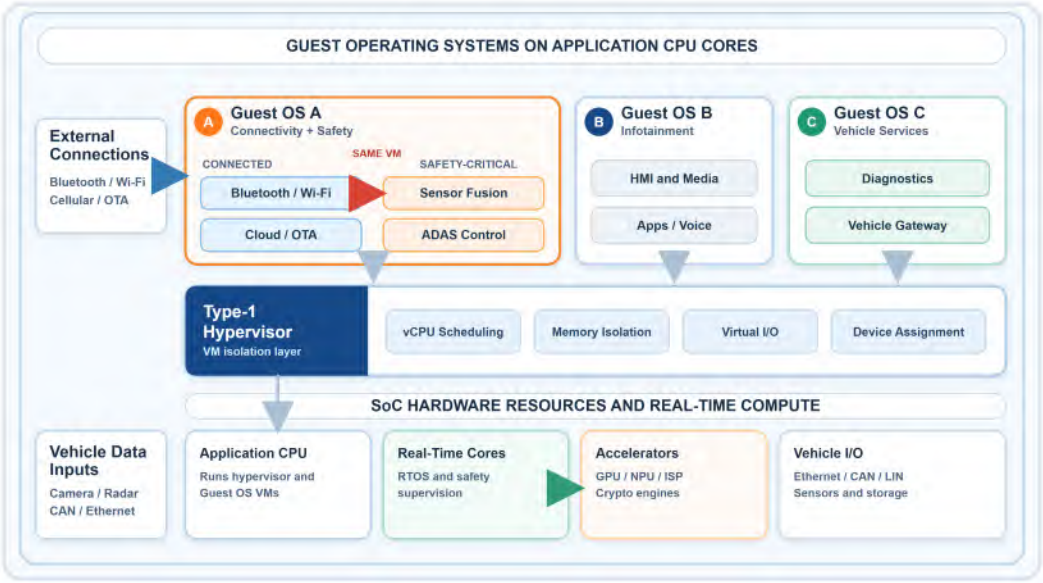


Figure 1.4: Central computer software partitioning

This type of architecture, depicted in *Figure 1.4*, differs significantly from domain-based architecture, where safety-critical applications were partitioned into standalone ECUs. Even if those ECUs had imperfect security, remote exploitation was still challenging due to multiple physical barriers. In a central computer architecture, the challenge of isolation shifts from physical domain separation to software and hardware separation within a single computer platform. It is easy to see why the security isolation guarantees offered by the hardware must hold true to emulate the physically separated domain functions.

Ideally, domain functions are isolated in dedicated VMs, spatially and temporally separated by a Type 1 hypervisor. The hypervisor ensures each VM is allocated its own time slice, and the memory and hardware resources of one VM are inaccessible to other VMs. If sharing resources is needed, the hypervisor normally will set up dedicated services that act as mediators to the common resources. In cases where VM management overhead is too great, it is common to find a single Guest VM with multiple containers or a mixture of VMs and containers, some hosting multiple applications and others hosting multiple containers. While this architecture presents a greater challenge for offensive security engineers, the attackers' focus shifts to finding vulnerable applications which can be the launching points to compromise other critical services of the system, as we will see in later chapters.

Whether evaluating the security of a domain-based or central computer-based architecture, one must first familiarize themselves with the externally exposed vehicle interfaces, which

serve as the entry point into the deeper layers of the E/E architecture, which will be the topic of the next section.

External vehicle interfaces

At the edge of the vehicle E/E architecture are the exposed external interfaces, such as Wi-Fi, Bluetooth, cellular (GSM/4G/5G), Global Positioning System (GPS) antennas, V2X, and Near Field Communication (NFC)/Radio Frequency (RF) wireless entry access points. In addition to wireless interfaces, numerous physical interfaces exist, such as Universal Serial Bus (USB) ports, the On-board Diagnostics port (OBD), **Electric Vehicle Supply Equipment (EVSE)** charging ports and any other pluggable interface that is meant to connect the vehicle to its environment.

Depending on the attackers' objectives, enumerating the external interfaces whether wireless or physical is the first step to determining which ones are most likely to yield a successful breach. For instance, when targeting the software update flow, Wi-Fi and cellular interfaces will be the first target, while circumventing the vehicle entry systems makes NFC and RF interfaces more interesting. Each interface exposes a suite of communication protocols that must be understood before developing an attack strategy.

By exploring the external vehicle interfaces, the attacker's goal is to understand the security perimeter of the vehicle and identify entry points that will likely expose exploitable vulnerabilities. As we move further into E/E architecture, the first type of component that we encounter is the ECU, which is the focus of the next section.

Electronic control units (ECUs)

A fundamental building block of automotive systems, the ECU is typically the primary target in any successful offensive security campaign as it is the one that is performing the actual control of the vehicle function. Whether it is unlocking the car doors, starting the engine, charging the vehicle's batteries or applying the brakes, the ECU is the networked component that is sensing, processing and actuating.

But ECUs are not all equal in security risk exposure or exploitation value. For example, the telematics ECU has a high degree of risk exposure but alone it cannot alter the vehicle's motion. In contrast, while an airbag module is not externally accessible, it can cause serious damage to the occupants if the airbags are deployed outside the normal operating conditions. To better understand the differences between these ECUs, we like to group them into three categories:

- **Low-performance microcontrollers**: Powered by a single microcontroller or Application-Specific Integrated Circuit (ASIC), these ECUs control a single major function of the vehicle and are interconnected to sensors and/or actuators via LIN, CAN

or both. Their software typically runs on a single core with embedded flash and Random Access Memory (RAM). Normally, these systems run proprietary software that is tightly coupled with the hardware due to performance reasons. Some smart sensors can fall into this category when they contain specialized software that executes in an ASIC environment and communicates over a low-bandwidth protocol such as LIN.

- **Medium-performance microcontrollers**: These implement one or more low-level vehicle functions into a single ECU by leveraging multicore microcontrollers with larger flash and RAM resources. In addition to communicating via LIN and CAN, these ECUs normally support Ethernet to enable communication with high-performance ECUs and possibly the external vehicle environment. Software for these ECUs is executed within a real-time operating system environment that is typically implemented using a standardized software platform like AUTomotive Open System Architecture (AUTOSAR) Classic Platform.
- **High-performance Controller (SoC-based)**: These are the most sophisticated ECUs as they are hosted on one or more SoCs which execute over a mixture of application and real-time cores, boot from external flash, and rely heavily on external Dynamic Random-Access Memory (DRAM). A High-Performance Computer (HPC) typically runs a mix of rich operating systems, like QNX or Linux, alongside one or more Real-Time Operating System (RTOS) instances and shares many features with enterprise systems, making them attractive to attackers. Unlike the previous two types, these systems have a complex software bill of material (SBOM) with many open source and off-the-shelf components.

ECUs depend on a large set of sensors to perform their functions. But sensors introduce their own set of security threats. In the next section, we will briefly survey the type of sensors we expect to find in a modern connected vehicle.

Sensors

Vehicle **sensors** in a connected vehicle are widely distributed, serving a diverse set of purposes, making them a large attack surface. Sensors on the vehicle body include cameras, ultrasonics, Light Detection And Ranging (LiDAR), Radio Detection and Ranging (Radar), and RF signal receivers. Inside the cabin, sensors serve many purposes such as helping in monitoring the driver's presence, controlling cabin temperature, sensing the seat position, and various other purposes that contribute to the safety and comfort of the vehicle occupants.

Sensors are also embedded deep within the ECU and other electromechanical components. For example, sensors in the engine compartment help in emissions control, while sensors within the braking system help with brake fluid control. Attackers may target any of these sensors

depending on how the sensor contributes to their attack objectives and how easy it is to interfere with the sensor.

Three factors determine a sensor's level of security exposure: its location within the vehicle, its physical characteristics, and its interface method. Naturally, sensors that are mounted on the exterior of the vehicle are the most exposed to direct interference. Sensors whose physical characteristics make them susceptible to interference are also attractive targets, typically through proximity attacks. Finally, sensors that are accessible through a network link are also a valid target since the attacker only needs to be on the local vehicle network to influence the sensors. In later chapters, we will see how attackers can abuse sensors in real-life attacks to achieve their attack objective.

Networking protocols

Vehicle components are interconnected through a variety of communication protocols, each designed to support specific types of data exchange and performance requirements. As modern vehicles grow increasingly complex, with numerous subsystems interacting in real-time, the choice and implementation of these protocols play a crucial role in ensuring safe, efficient, and secure vehicle operation. In this section, we will explore three of the most common protocols used in automotive networks: LIN, CAN, and Ethernet, each presenting its own unique security challenges and operational benefits.

Local interconnect network (LIN)

LIN is a cost-effective protocol primarily used for controlling sensors and simple actuators in low-performance systems. LIN can be found in non-critical areas of the vehicle, such as controlling window lifts and seat adjustments, but also in critical areas such as communicating with sensors. Its low bandwidth (up to 20 kbps) and single-wire communication make it ideal for functions that do not require high bandwidth.

LIN networks are also more vulnerable to interference and have fewer built-in error-checking mechanisms, making them less robust compared to other protocols such as CAN. Despite these limitations, LIN continues to play a key role in automotive electronics due to its simplicity and low cost, especially in applications where its performance constraints are acceptable. When it comes to security, securing individual LIN frames through cryptographic methods can be challenging, given the small payload of LIN messages.

Controller area network (CAN)

The **CAN** protocol is one of the widely used communication standards in the automotive industry, known for its reliability, robustness, and real-time performance. CAN was initially developed to allow ECUs in vehicles to communicate without the need for a complex wiring harness, which is especially critical as modern vehicles can have 50 to 100 ECUs controlling various systems.

CAN is designed to support medium-performance ECUs, such as those used for engine management, transmission control, braking systems, and other real-time functions. The protocol includes built-in error detection and fault confinement mechanisms, which makes it highly reliable, even in harsh automotive environments where electromagnetic interference or signal degradation can occur.

However, as the complexity of vehicle systems increases, the limitations of CAN 2.0 and CAN FD, particularly in terms of bandwidth, become more evident. They struggle to accommodate the high data throughput requirements of newer applications such as **advanced driver assistance systems** (**ADAS**) and vehicle infotainment. Nonetheless, their proven track record and widespread industry adoption ensure that CAN will remain a fundamental part of vehicle networks for years to come, often operating alongside newer protocols like Ethernet.

When it comes to security, CAN suffers from being insecure by design and the attempts to secure it normally result in a high-performance impact on the already constrained automotive ECUs. Nevertheless, several security measures exist to address the weaknesses of the CAN protocol, and those will be discussed in later chapters as we evaluate attacks against the in-vehicle network.

Ethernet

As vehicles evolved into highly connected, data-intensive platforms, **Ethernet** emerged as a key enabler for advanced automotive applications. Ethernet is a relatively new but rapidly growing protocol in the automotive domain, bringing with it high data rates and flexibility traditionally associated with enterprise networks. With speeds ranging from 100 Mbps to 10 Gbps, Ethernet is well-suited for high-performance ECUs and complex systems that require fast, reliable communication, such as those involved in autonomous driving, real-time sensor fusion, and infotainment systems.

Unlike LIN and CAN, which are purpose-built for automotive use, Ethernet has its roots in enterprise networking. Its adoption in the automotive industry represents a shift toward more open and scalable communication frameworks. Ethernet's ability to handle vast amounts of data makes it an ideal solution for high-bandwidth applications, such as transmitting video data from cameras, Radar, and LiDAR systems, which are essential for the growing suite of ADAS technologies.

However, implementing Ethernet in vehicles presents its own challenges. The automotive environment is vastly different from traditional IT networks, requiring Ethernet to be adapted for higher levels of robustness, safety, and real-time performance. The development of Automotive Ethernet, a version of the protocol specifically tailored for vehicle use, has addressed many of these challenges, including the need for deterministic timing, reduced latency, and enhanced security. Additionally, as Ethernet continues to mature in the

automotive space, it must be integrated alongside existing legacy systems, such as CAN, creating a hybrid network architecture that balances the best of both worlds. When it comes to security, Ethernet offers many security protocols that are inherited from enterprise systems, but enabling all of them is not always feasible given the performance impact incurred on real-time systems.

The vehicle E/E architectures, hardware platforms, software layers, and communication networks described so far define how functions are implemented inside the vehicle. However, a connected vehicle does not operate only within its internal architecture. Its ECUs, interfaces, and services interact with various external systems such as cloud backends, diagnostic equipment, charging infrastructure, and wireless networks. To understand where attacks may originate from and how they can propagate, we must therefore expand our view from the vehicle's internal design to the broader operational environment in which it operates.

Understanding the operational environment of connected vehicles

The first step to understanding the vehicle's operational environment and the breadth of the attack surface is to enumerate the systems that interface with the vehicle within the operational environment. As shown in *Figure 1.5*, modern vehicles can no longer be viewed as isolated entities but are instead part of an integrated, complex, and interconnected ecosystem.

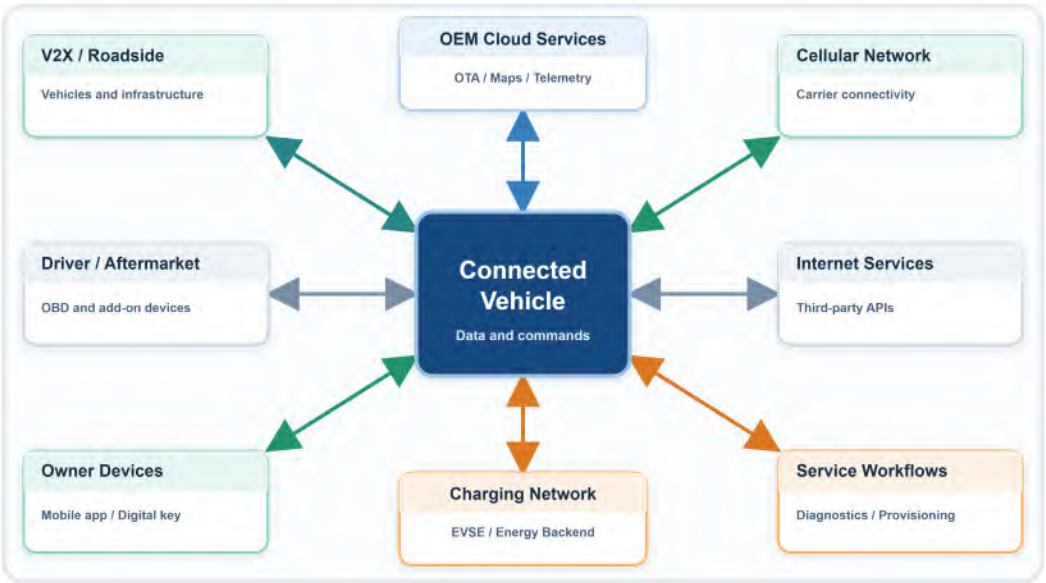


Figure 1.5: Threat sources against the connected vehicle

This ecosystem provides continuous connectivity to a variety of supporting systems, such as backend servers operated by vehicle manufacturers or service providers, cellular networks, the vehicle owner's personal devices, and even the wider internet. Each connection introduces potential pathways for cybersecurity threats.

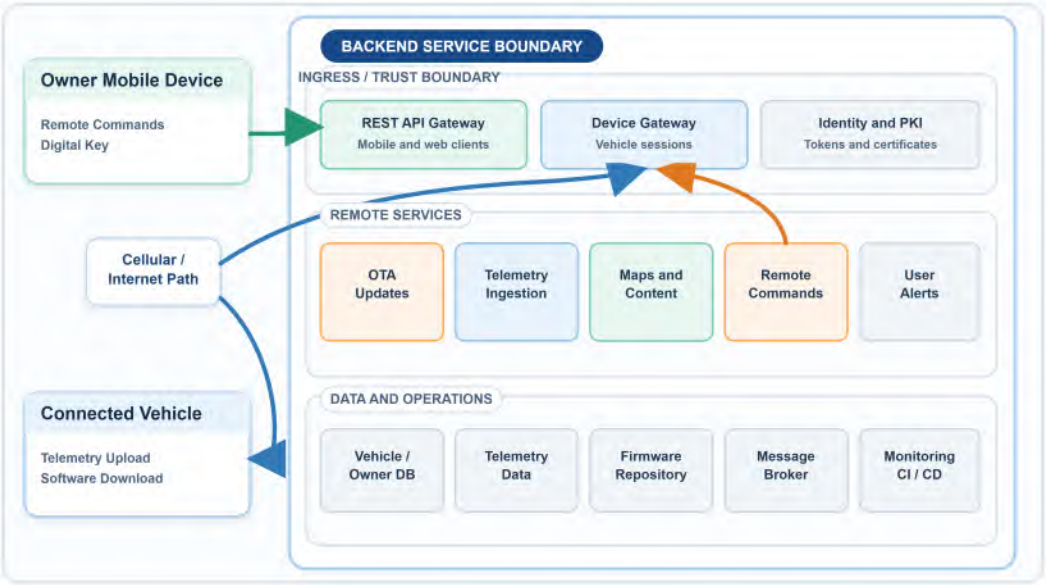


Figure 1.6: Sample connected vehicle backend architecture for remote services

Without integration with backend servers, drivers would not be able to receive frequent software updates, upload telemetry data, download map updates, or access a variety of functions that have become standard expectations in modern vehicles as shown in *Figure 1.6*. While this constant stream of communication is crucial for operational efficiency and enhancing the user's experience, it significantly expands the attack surface, creating numerous opportunities to construct attack chains against vehicle systems.

Similarly, vehicle owners have grown accustomed to linking their mobile devices to their vehicles, enabling them to access applications while driving and perform remote functions, such as unlocking doors, starting the engine, or tracking the vehicle's location via smartphones. However, if these channels are not properly secured, they introduce additional cybersecurity risks.

Outside normal operations, various tools and equipment come into contact with the vehicle's E/E architecture, such as diagnostic, development, and manufacturing tools. If the security of these tools is neglected, the overall security of the vehicle's operational environment cannot be guaranteed. Additionally, the inclusion of aftermarket devices and third-party services adds

another layer of complexity. These devices, ranging from OBD dongles to advanced aftermarket entertainment systems, may contain unknown vulnerabilities that serve as attack entry points into the vehicle's internal network.

Moreover, the evolving landscape of connected vehicles now includes direct communication capabilities with infrastructure (**Vehicle to Infrastructure** (**V2I**) communication) and other vehicles (**Vehicle to Vehicle** (**V2V**) communication). While these advancements promise significant improvements in safety and traffic management, they also raise cybersecurity concerns by increasing the attack surface. *Figure 1.7* illustrates the threat exposures in different lifecycle states.

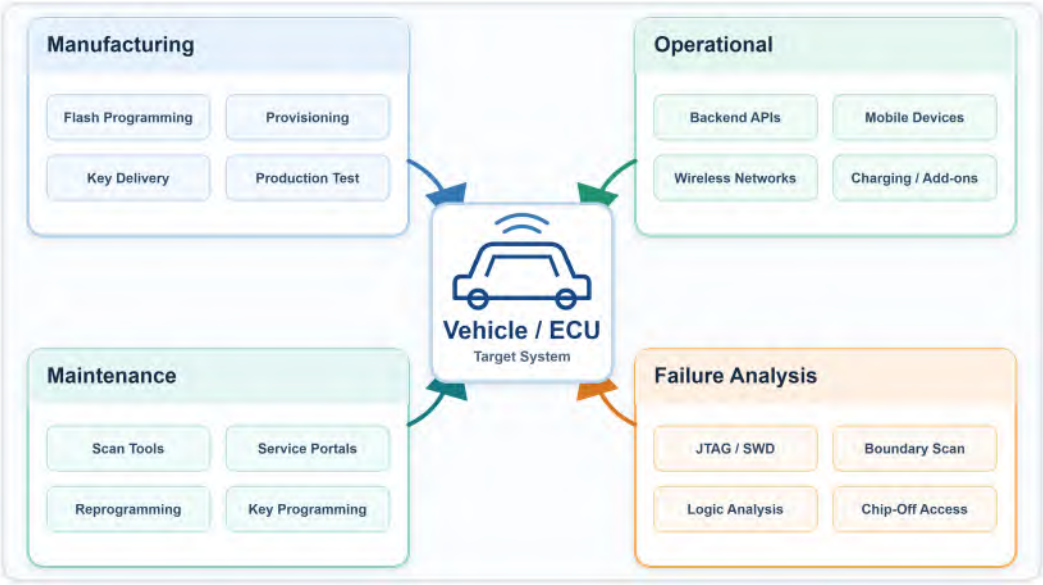


Figure 1.7: Threat exposure in vehicle lifecycle states

During manufacturing, the vehicle components interact with flash programming tools, provisioning software, fusing appliances, and various key management delivery services intended to install and configure the software and hardware in a secure and safe state. Once the vehicle is assembled and the car leaves the factory, it enters operational mode and there, a whole suite of operational environment components come into the picture such as backend servers, mobile devices, charging stations and even aftermarket equipment that adds new connectivity features.

Occasionally, the vehicle is expected to enter maintenance mode due to service needs at the dealer or mechanic shop. There, a host of diagnostic tools interact with the vehicle, and these can be used to diagnose and alter the vehicle's behavior if needed. Finally, there are special

cases where an ECU may exhibit anomalous behavior that is hard to diagnose using normal diagnostic tooling. In this case, the ECU is taken out of service, and special tools are used to perform invasive failure analysis such as **Joint Test Action Group** (**JTAG**) based debuggers and test equipment that is reserved for the manufacturer. By understanding each lifecycle state and the components involved, an attacker can focus on the ones most likely to expose exploitable weaknesses.

Now that we understand the various layers of the vehicle E/E architecture, we will turn our attention to the attacker profile to gain an understanding of who wants to attack automotive systems and to what end.

Attacker profile

Establishing consensus within an organization about the **attacker profile** for the system being secured is one of the most important yet often least discussed topics. A well-defined attacker profile influences how threats are assessed and the level of cybersecurity rigor required to address them. However, without consensus, organizations can waste time and resources debating which threats to consider and how much security to implement.

On one end of the spectrum, some organizations treat cybersecurity as a checkbox exercise, doing the bare minimum to satisfy compliance requirements. In such organizations, the attacker profile is rarely documented, and engineers often provide varying responses when asked who their systems should be resilient against, ranging from script kiddies to advanced threat actors. Without even a concept of an attacker profile, developers often rely on gut feelings regarding which attacks to consider and which ones to ignore.

On the other end of the spectrum, some organizations aspire to build products that can resist the most advanced attackers, including **advanced persistent threats (APTs)** from nation-states. While this ambition is commendable, it can lead to tension if there is no agreement among management, engineering, and business divisions about the target security level.

For example, faced with a specific threat of sensor manipulation, defensive security engineers may prescribe strong, multi-layered security controls to address this threat down to the sensor type being selected, while the business division might refuse the additional expense of acquiring a secure sensor. Management may also reject accepting the scheduling impacts and resource constraints for supporting such sensors. Instead, a seemingly endless debate ensues among all stakeholders about why the risk is important and what action is justified to reduce the risk. To avoid these debates, organizations should first document their attacker profile and then ensure that all stakeholders agree on the level of security resilience required. Once consensus is reached, cybersecurity risks can be systematically analyzed to address those risks that are considered within the agreed-upon scope.

For attackers, knowing the maturity level of the target security organization can be quite useful as it can give hints regarding what attacker capabilities the organization has considered and where the security limitations may exist in the product. *Figure 1.8* presents the attacker levels and capability escalation.



Figure 1.8: Attacker levels and capability escalation

Let's now look at these levels in detail.

Attacker levels

As shown in *Figure 1.8*, attacker capabilities increase as we go higher in the levels starting with script kiddies and ending with nation states as we will see next.

Script kiddies

Attackers have varied degrees of sophistication and a diverse set of motivations. At the lowest level are **hobbyists** and **script kiddies**, who know how to download malicious scripts and run them against target devices. These individuals often seek to modify their vehicles by circumventing OEM restrictions.

For example, engine tuners may flash new calibration sets to increase horsepower or use OBD diagnostic tools to roll back mileage to avoid paying lease overages or selling their vehicle for a higher price. While their technical skills are limited, they can still cause financial damage to their victims by exploiting simple vulnerabilities.

Motivated individuals

Next are **motivated individuals**, such as disgruntled employees or ideologically driven attackers (**hacktivists**). While their funding may be limited, their actions can still pose significant cybersecurity risks, especially to the OEMs. These individuals may research hacks and vulnerabilities to construct attack chains, purchase tools from the dark web, or leverage proprietary knowledge from their workplace.

Insider threat actors are particularly dangerous, as they have access to critical information and can cause severe damage, such as pushing incompatible software updates or inserting malicious code in the code base during development. Insider threats, though not widespread, can also involve espionage for financial gain, with individuals selling valuable intellectual property to foreign governments or competitors.

Note

Over time, the capabilities of script kiddies increase because more technically capable motivated individuals with specific goals may release their research to the public. This allows script kiddies to achieve the same outcomes. Popular examples of this include circumventing paid features through pirated software or custom tuning configurations.

Criminal groups

Criminal groups represent the next level of sophistication. These groups hire skilled hackers to craft specialized attacks, often offering "hackers-for-hire" services. In the automotive space, such groups produce vehicle theft equipment, constantly evolving their tools to stay ahead of security defenses. They possess a deep understanding of IT and embedded systems and can discover and exploit vulnerabilities.

Researchers and ethical hackers

Researchers and ethical hackers, while typically motivated by academic recognition or financial bounties, must also be considered in the attacker profile. Although they normally do not intend to cause harm, their improper disclosure of vulnerabilities can have significant financial and reputational consequences if the organization does not address the issues in time.

Organized crime

At a higher level of sophistication are **organized crime** groups operating at a multinational level. These groups use techniques like social engineering and ransomware, and while their presence in the automotive space is less documented, one could imagine scenarios where they target specific vehicles for political or financial reasons. For example, a high-paying client may

seek to "accidentally" disable a specific vehicle model in an untraceable way to pressure an OEM to pay a ransom.

Nation-state actors

The most advanced attackers are APTs supported by **nation-states** that are motivated by political and economic agendas. These attackers have significant resources and access to proprietary tools. They may masquerade as hacktivists for plausible deniability. Their capabilities make them formidable adversaries, and their attacks are often as secretive as national defense strategies.

These attackers may also possess the ability to infiltrate the supply chain through sophisticated attacks that are hard to discover, such as the injection of fake sensors, cloned chips, and counterfeit networking hardware. Given the depth of the automotive supply chain, it is easy to imagine that an APT group would target a weak link in the automotive supply chain to achieve their target, similar to how SolarWinds was targeted to impact private companies and government entities that use their software [2].

On the spectrum between script kiddies and nation-states, most automotive companies design their systems to be resilient against criminal groups. This is reflected in the level of security controls they implement. For instance, when designing protection for code running within a trusted execution environment, some companies may decide that attackers capable of breaking such specialized systems are out of scope, terminating their security analysis at that level. On the other hand, organizations that aim to protect against nation-state-level threats must go further, ensuring resilience even against highly sophisticated, multi-layered attacks. Once an attacker profile is agreed upon, defensive security engineers must remain consistent with that profile, especially during the threat modeling phase.

Note

Advances in artificial intelligence for offensive security have shifted assumptions about attacker profiles by increasing the capabilities available at every attacker level. This uplift should be considered when defining or documenting the attacker profile.

Psychology of developers

Security is as much about human psychology as it is about technology. By understanding the psychology of developers, we can predict the types of security weaknesses their products may contain. Similarly, by understanding the psychology of attackers, we can anticipate the lengths to which they will go to achieve their objectives and what those objectives might be.

Developers face numerous challenges when designing and building automotive systems and adding security constraints to those systems only increases the development complexity. For

example, debugging a CAN network that is experiencing intermittent signal timeout errors can be quite challenging. Now imagine that the network has already enforced CAN message authentication, and the developers attempting to troubleshoot the network lack the cryptographic keys needed to send valid authenticated messages. In this scenario, the developer who is trying to troubleshoot the issue under a strict delivery timeline may be forced to disable the authentication by introducing a backdoor that remains in the system even after production!

This fictional scenario is common in the automotive engineering space, where the psychology of problem-solving can be a source of new security weaknesses that are left behind and waiting to be exploited. In the example above, the developer should have worked closely with the security engineers to allow for authentication to be disabled through a secure mechanism like a build option that is removed once the system enters production. Offensive security professionals who understand the developer's psychology can anticipate the existence of such vulnerabilities, as backdoors and shortcuts are likely occurrences.

Along the same lines, it is possible to find leftover features that were intended for development only, such as verbose logs that reveal sensitive data, special test drivers, and even development-only cryptographic keys in production-intended systems. By understanding these psychological factors, offensive security professionals can narrow their search areas when looking for security weaknesses.

Psychological tactics

Attackers often exploit human psychological weaknesses through social engineering, using fear, scarcity, and urgency to push victims into poor decisions. For example, attackers may send emails or messages with malicious links that create a false sense of urgency and encourage users to bypass critical thinking. While direct social engineering has a more limited scope in automotive systems than in enterprise IT, attackers may still exploit urgency during incident response, software updates, supplier interactions, or service operations to influence decisions and bypass normal safeguards.

Scarcity is another tactic attackers use to pressure victims by creating the impression that time, access, support, or resources are limited. In automotive environments, this may take the form of impersonating a supplier, service provider, or internal stakeholder and claiming that immediate action is required before an update window closes, access expires, or a critical system becomes unavailable.

Fear can also be used to manipulate decisions. Attackers may exaggerate the consequences of inaction to pressure personnel into approving unsafe changes, disclosing sensitive information, or bypassing established procedures. At the same time, organizations should be careful not to let internal security culture become driven only by fear, as this may cause teams to focus only on the most visible threats while neglecting more subtle but still dangerous risks.

Now that we have established the various attacker capabilities and the importance of building the right attacker profile, it is time to examine the fundamental concepts of weaknesses, vulnerabilities, and threats, and how these terms relate to one another.

Weaknesses, vulnerabilities, and threats

According to NIST, a vulnerability is a "*weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source*". For automotive systems, the same definition applies with the slight modification that the weakness can exist in any component within the vehicle or its supporting systems. A weakness, on the other hand, is defined as a "*defect or characteristic that may lead to undesirable behavior*".

Naturally, weaknesses make a system more susceptible to abuse by an attacker. From this perspective, every vulnerability can be considered a weakness, but not every weakness is necessarily a vulnerability. When it comes to exploitation, defensive security engineers focus on eliminating weaknesses by design to reduce the risk of exploitation, while offensive security professionals search for weaknesses that have been missed by defensive teams and seek their exploitation.

> **Note**
>
> **Differing focus:**
>
> While offensive security engineers focus on vulnerability discovery and exploitation, defensive security engineers focus on vulnerability elimination.

Vulnerabilities differ not only by technical cause, but also by knowledge state: who knows about the flaw, whether it has been discovered, and whether it has been disclosed publicly, as shown in *Figure 1.9*.

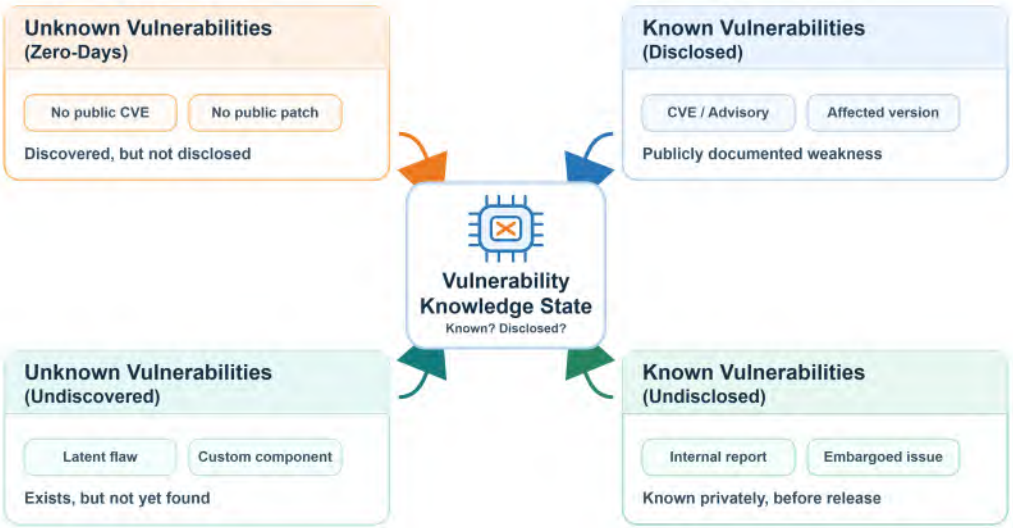


Figure 1.9: Vulnerability types based on their discovery state

The first category is unknown and undiscovered vulnerabilities which are essentially latent defects in ECU software, hardware, protocols, integration logic, or unusual operating states that no researcher, defender, or attacker has yet identified. These are the hidden weaknesses that security testing aims to uncover.

Once a flaw has been found, it moves into a different state. If it is known only to an attacker or researcher and has not been disclosed to the affected defenders, it is commonly treated as a zero-day. What makes a zero-day special is not simply that the security flaw exists but rather that defenders have had "zero days" to prepare because there is no public advisory, no CVE entry, and usually no broadly available patch or mitigation. Finding and exploiting such flaws often requires significant research effort, so attackers tend to reserve them for high-value targets where the payoff justifies the cost and exposure risk.

A vulnerability may also be known but undisclosed when it is privately tracked by an OEM, supplier, researcher, or attacker, but not yet public. Examples include internal bug reports, unpatched supplier issues, or embargoed research findings. By contrast, known and disclosed vulnerabilities are public: they appear in advisories, CVEs, affected-version lists, or mitigation guidance. These are often the attacker's first choice because they can fingerprint vulnerable versions, reuse public technical details, and chain multiple known weaknesses together before defenders have fully patched their fleets.

Note

APTs often stockpile multiple zero-day vulnerabilities but avoid using them unless the target offers significant value, since using them may expose a capability they want to keep hidden. As a result, defenders may remain unaware of these capabilities and become overconfident in their systems' security. Recent advancements in AI-based vulnerability discovery demonstrate this exact fact, as code bases that were assumed to be highly secure are suddenly discovering numerous vulnerabilities with high criticality.

Searching for known vulnerabilities is a common technique when the target system integrates off-the-shelf components. Automotive systems have traditionally relied heavily on custom solutions, making the exploitation of known vulnerabilities only partially effective. However, as the industry shifts toward SDVs with high-performance computing systems, off-the-shelf and open-source components have proliferated in modern vehicles. This makes the exploitation of known vulnerabilities more effective. Nevertheless, even with the proliferation of **commercial off-the-shelf (COTS)** components and open-source software (OSS) components, custom software and hardware remain prevalent, necessitating the discovery of zero-day vulnerabilities to breach such systems effectively.

Similarly, efforts in the automotive industry for standardizing software and hardware platforms as well as various protocols make the discovery of zero days much more lucrative as attackers can discover vulnerabilities in standard components and exploit them wherever those components are used.

Note

Advances in artificial intelligence have lowered the cost of vulnerability research by accelerating code analysis, reverse engineering, fuzzing, and exploit development, which is contributing to a rise in the discovery of zero-day vulnerabilities across many classes of software and embedded systems.

The art of offensive security is not just about finding weaknesses but also prioritizing those that are easiest to exploit. As with any profession, attackers prefer the path of least resistance, seeking to achieve their objectives with the least amount of time and effort. If a system has many easily exploitable weaknesses, attackers will target those first to determine if they are sufficient to meet their goals. Only after exhausting the readily exploitable weaknesses will attackers move on to more sophisticated methods. Skilled attackers can leverage seemingly unexploitable weaknesses to build effective attack chains.

For example, one particularly hard-to-imagine exploit evaluated by Google's Project Zero [3] involves using a Portable Document Format (PDF) parsing vulnerability to hijack an iMessage via a `fake gif` trick. Attackers exploited a bug in Apple's CoreGraphics PDF parser through an old compression standard, **Joint Bi-level Image Group 2 (JBIG2)**. This vulnerability allowed them to construct a complex, logic-gate-based exploit that emulates a Turing-complete computer within the image processing code. They then leveraged this to execute commands, all from a single image file.

The sophistication and creativity in turning a compression technique into an exploit was extraordinary. It is easy to see how a defensive security engineer may come across an old compression standard such as JBIG2 and not realize that it contains weaknesses that can enable such a sophisticated attack but that does not deter offensive security professionals from exploiting it in an advanced attack chain.

Tip

Food for thought:

Don't let your lack of imagination blind you to your adversaries' creativity and skills.

Vulnerability scores

Not all vulnerabilities are created equal, which is why an objective scoring method is essential for rating the severity of vulnerabilities. Attackers are naturally drawn to high-scoring vulnerabilities, as these tend to be easier to exploit and have more severe outcomes. A commonly used vulnerability scoring system is the **Common Vulnerability Scoring System (CVSS)**.

Note

When choosing a vulnerability scoring scheme, it is recommended to pick one that accounts for safety impacts such as CVSS version 4.0.

The CVSS scoring scheme considers safety impacts to integrity and availability of the system according to the following criteria: "*The exploited vulnerability will result in integrity impacts that could cause serious injury or worse (categories of "Marginal" or worse as described in IEC 61508) to a human actor or participant*."

Common Vulnerabilities and Exposures (CVEs) and vulnerability reports are crucial for conducting successful offensive security testing. After the discovery phase, attackers can identify which vulnerabilities exist within a product and begin preparing exploits. Useful sources for information on vulnerability include several well-established platforms. The

Cybersecurity and Infrastructure Security Agency (CISA) provide alerts and advisories on emerging threats and vulnerabilities. **Japan Computer Emergency Response Team (JPCERT)** offers vulnerability reports specific to Japan. The **National Vulnerability Database (NVD)**, maintained by NIST, provides a comprehensive list of known vulnerabilities.

These usually have a CVE number that denotes the year they were discovered, and the metrics used to arrive at the score as shown in *Figure 1.10*.

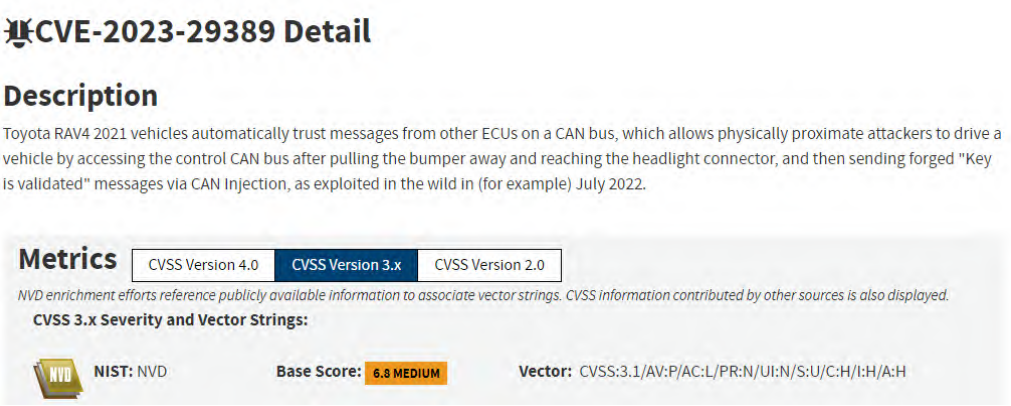


Figure 1.10: CVE example showing the ID and score

On the other hand, frameworks like **MITRE's Common Weakness Enumeration (CWE)** and **Common Attack Pattern Enumeration and Classification (CAPEC)** offer structured methods for understanding and categorizing vulnerabilities and attack patterns. A popular example of a CWE is CWE-120: "Buffer Copy without Checking Size of Input," which describes a common weakness where data is copied from an input to an output buffer without verifying that the size of the input data is less than or equal to the capacity of the output buffer. As a result, an oversized input can lead to a buffer overflow.

Before we conclude this section, it is important to draw our attention to another fundamental concept which is threats.

Threats

While weaknesses and vulnerabilities are internal to the system, threats originate from external sources. NIST defines a threat as "*any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service. Also, the potential for a threat-source to successfully exploit a particular information system vulnerability.*" In the context of automotive systems, one can think of threats simply as actions that aim to exploit vulnerabilities to achieve specific attack objectives.

A system may have several vulnerabilities, but without an active threat actor, it may go unbreached for an extended period. This can lead to the mistaken belief that the system's security posture is robust. Conversely, a system may be exposed to various threats, but if weaknesses are not feasibly exploitable, a breach may never occur. This can create a false sense of security, leading organizations to relax security measures due to the lack of security incidents. Both misconceptions need to be challenged, and offensive security testing is one way to prove the state of a system through rigorous security validation.

In automotive systems, threats originate from the backend, the supporting tools, the onboard diagnostic interface, mobile devices, and a myriad of other sources that exist in the operational environment. These threats aim to compromise properties of system assets such as integrity, authenticity, and availability through tampering, spoofing, and denial of service.

Now that we have covered the basics of weaknesses, vulnerabilities, and threats, we are finally ready to dive into the domain of penetration testing including its various phases and levels of depth.

Introduction to penetration testing

While the **cybersecurity management system (CSMS)**, as defined by ISO/SAE 21434 [9], aims to achieve an adequate level of cybersecurity assurance for automotive products, the ultimate way to measure a system's security comes down to how it behaves under two scenarios: when subjected to an independent penetration test and when faced with a real-life attack. The former is conducted under the product owner's terms and has the advantage of allowing findings to be addressed before real damage occurs. The latter is a real-world attack carried out by, for example, cyber criminals, and has real consequences such as financial or safety adverse effects. Consequently, when discussing offensive cybersecurity, we can study it from two perspectives: penetration test engineers and real-life attackers. One might wonder what the difference is between the two.

The simple answer comes down to permission, intent, scope, and access to information:

- Penetration testers obtain **permission** from the target owner before conducting an attack campaign, whereas real-life attackers do not wait for permission.
- Their **intent** also differs. One group seeks to compromise the system to identify weaknesses that can be addressed before they are exploited in the real world, while the other seeks compromise for personal, financial, political, or strategic gain.
  Another difference is how each group handles what it discovers. Penetration testers report vulnerabilities ethically and within a legal framework. Real-world attackers, by contrast, may never disclose what they have found, may sell the information, or may reveal it only through extortion or a ransom note.

- **Scope** is another important distinction. Penetration testers are bound by an agreed scope that defines which components, interfaces, and supporting systems they are permitted to assess. This can limit their ability to uncover end-to-end attack paths that cross system boundaries. For example, a test scoped to a single ECU may miss weaknesses that become exploitable only through a backend service, another ECU, or a mobile application. Similarly, a vehicle assessment that excludes the charging station or charging backend may fail to identify vulnerabilities that arise through those external components. Real-world attackers are not constrained by such scope limitations and can pursue any path that offers the best chance of success.
- **Resources and access to sensitive information** also differ. Penetration testers typically work within limited engagement timelines, constrained by cost and the availability of skilled resources. Real-world attackers, on the other hand, may spend months or even years attempting to penetrate a system, depending on their sophistication and level of support. Penetration testers may also receive sensitive information from the client under a gray-box or white-box approach (which we will explore later in this chapter) to make testing more effective. In contrast, real-world attackers generally begin from a black-box position with no legitimate prior access to such information. Therefore, one must remain cognizant of the limitations of penetration testing in fully uncovering security weaknesses when compared to real-life attacks.

Despite these differences, both groups often rely on similar methodologies, tools, and attack techniques.

Leveraging the NIST framework

Since penetration testing is a well-established field, we will study offensive security from the perspective of penetration testers. But first, we need to establish a baseline for what penetration testing is in the context of automotive systems. In later chapters, we will delve into each phase of performing penetration testing with the right level of depth, but for now, we want to introduce the basic concepts, so we have a roadmap to carry out successful offensive security initiatives. Following the guidance of NIST SP 800-115 [4], we describe penetration testing as the systematic approach to assess the security of a product against a set of security objectives through a set of methods, tools, and techniques, preferably before the system is deployed on the road. While the NIST standard uses the term **information security assessment**, we will avoid this term, so it is not confused with process compliance assessments which are normally done within the context of ISO/SAE 21434 for automotive systems.

NIST Special Publication 800-115, "*Technical Guide to Information Security Testing and Assessment*," provides a structured approach to security assessments that is also highly applicable to offensive security in embedded automotive systems. By adapting its core concepts, security professionals can more systematically identify and exploit vulnerabilities in modern vehicles, thereby improving the overall cybersecurity posture.

Security assessment planning is the foundational step. It begins with clearly defining the scope of the assessment, with particular focus on critical elements such as ECUs, communication interfaces, and embedded software. By applying these concepts to offensive security in automotive systems, professionals can adopt a more disciplined and effective approach to identifying and exploiting vehicle-specific vulnerabilities.

ISO 21434 and pentesting

ISO/SAE 21434 defines penetration testing as "*Cybersecurity testing in which real-world attacks are mimicked to identify ways to compromise cybersecurity goals*." As such, ISO/SAE 21434 relies on penetration testing as a method to identify vulnerabilities and a recommended practice for validating cybersecurity goals.

Note

Pentesting as a validation method:

A common pitfall is to confuse security validation activities with security verification testing. Security verification should be performed first to confirm that specified security controls have been implemented correctly and behave as intended, such as confirming that secure boot rejects unauthorized software or that diagnostic access restrictions are enforced. Penetration testing then complements verification by evaluating the system from an attacker's perspective and searching for ways to compromise cybersecurity goals despite the implemented controls. Rather than replacing verification, penetration testing helps uncover unexpected attack paths, configuration weaknesses, flawed assumptions, and previously unknown vulnerabilities that may have escaped the normal design, implementation, and verification process.

Even though it is defined as a recommendation, there is a general expectation within the automotive industry that a certain level of penetration testing is carried out depending on the criticality of the product, as evident by its level of risk exposure. It is common for OEMs to require a supplier to perform penetration testing of certain components that they deem to be highly critical; for example, the telematics ECU is a frequent candidate for testing due to its long-range attack surface.

Similarly, the OEM is expected to perform penetration testing of its own vehicle networks and supporting systems such as the backend, the manufacturing, and service environment. When performing penetration testing, one can leverage other work products prescribed by ISO/SAE 21434 such as the item definition, the threat model, cybersecurity specifications, and vulnerability analysis reports as shown in *Figure 1.11*.

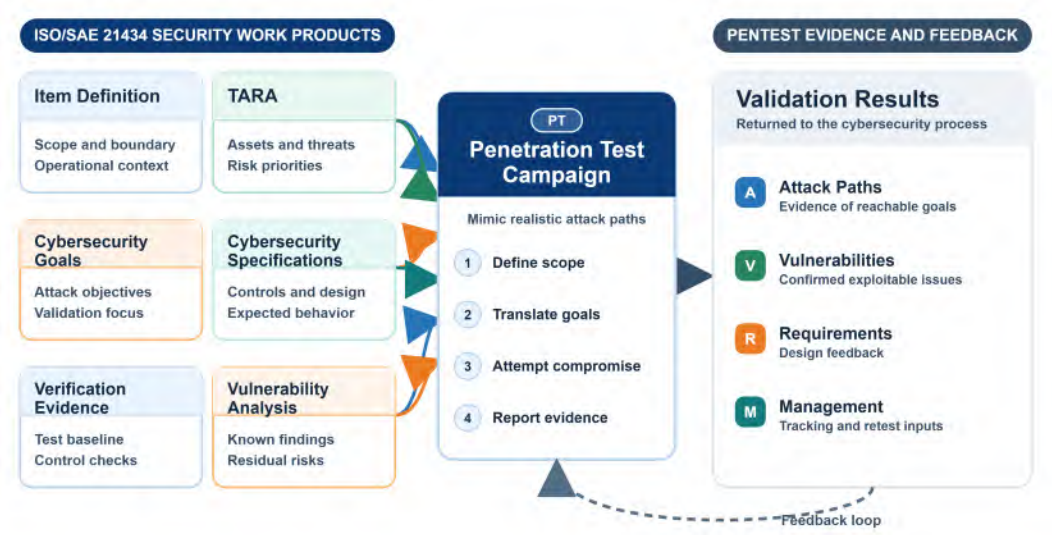


Figure 1.11: Pentesting in the context of a cybersecurity management system

For example, during the discovery phase, when defining what is in scope for the penetration test, the item definition can be a useful start because it allows us to understand the elements of the operational environment, the item boundary, and the target system. Sharing these details with penetration testers can lead to more insightful results.

The **Threat Analysis and Risk Assessment (TARA)** is another useful source of information that can help in identifying assets that are targets for the penetration test. Here again, assets deemed critical for the correct and safe operation of the system must be prioritized over other less critical assets. Cybersecurity goals identified during the TARA process can be translated to attack objectives that the penetration test campaign should attempt to achieve. Simply put, the penetration tester should aim to prove that violating security goals of the system is possible by uncovering exploits that defeat the system's ability to mitigate certain risks.

Depending on the type of penetration test, it may be possible to use the cybersecurity specifications as an additional source of input to identify security design weaknesses and security requirement gaps.

In addition to pentesting a product, an organization may want to test its ability to respond to vulnerability reports in adequate time to patch a critical issue. This type of tabletop exercise is meant to simulate a real-world attack to ensure that defensive security engineers first follow the process for proper incident management and second, that they can handle the report in the expected time to ensure the issue is patched promptly. Also, auditing the results of the vulnerability analysis and management process is a good practice to ensure that vulnerabilities are scored accurately and that the corrective action is being selected correctly.

Cybersecurity assurance through penetration testing

You design your system according to systematic cybersecurity standards. You are ready to ship your product, but no independent party has ever validated any of your claimed security assurances. You ship your product and, in a few weeks, or months, you start receiving reports of cybersecurity incidents, some of which may have legal, financial, or even life-threatening implications. You wonder how that could have happened? After spending weeks analyzing vulnerability reports, you conclude that there was a series of oversights that your normal security testing did not catch. Was there something you could have done to prevent that?

If building cybersecurity controls is about reducing the security risks of the system to a reasonable level, then penetration testing is about confirming that the security risk has been sufficiently reduced to the desired level. Given that even systems developed to a high security assurance level will contain residual risks and unmitigated risks that are unknown to the system designer, there is a need for a method to uncover those risks in a controlled manner.

Known risks are those that are documented during the TARA phase. **Unknown risks** are those that are hidden from the system designer due perhaps to the incompleteness of the TARA. **Residual risks** are those that result from mitigations that are not fully effective in addressing the original risk. For example, a system that implements secure boot checks to detect tampering of software binaries in storage media is still exposed to the residual risk that the boot measurement check itself may be subject to glitching attacks that can bypass the secure boot process. You can see different types of risks illustrated in *Figure 1.12*.

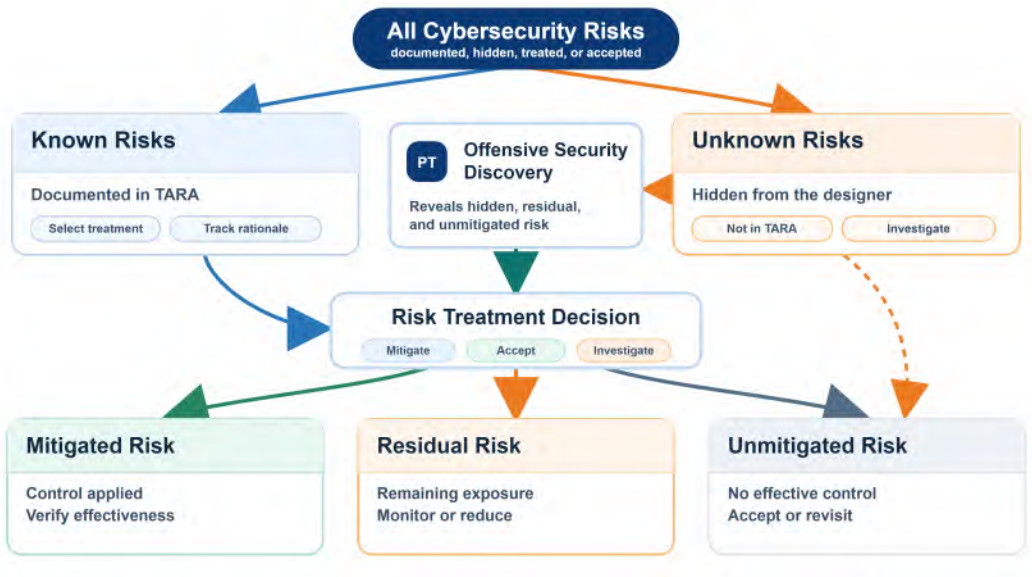


Figure 1.12: Cybersecurity risk types based on mitigation state

Unmitigated risks, on the other hand, are those risks for which no mitigation was possible. In case they are known, the system designer may have deemed the risk level to be too low or may have accepted the risk due to technical or performance reasons. For example, a system designer might choose to accept speculative execution vulnerabilities in the CPU due to the significant performance degradation if the mitigation is applied. Whether known or unknown, the number of residual and unmitigated risks create opportunities for exploitation by determined adversaries with the right skillset and resources. It is the job of offensive security professionals to expose how these risks can be exploited before they are discovered by real-world attackers.

Cybersecurity assurance is the confidence that a system meets the defined security goals and has effectively reduced its cybersecurity risks to an acceptable level. When examining the level of cybersecurity assurance of a given automotive product, we have two options: the process-driven approach and the technical-driven one.

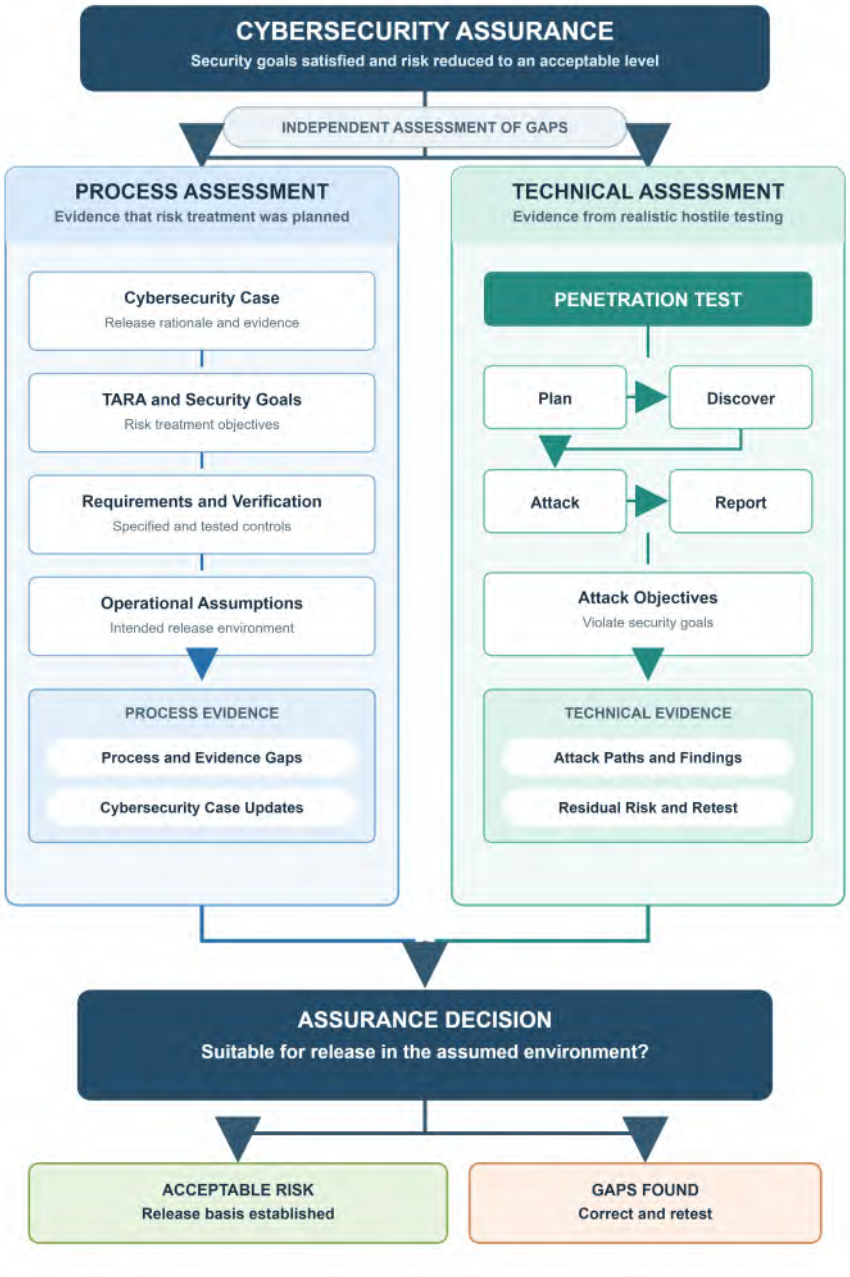


Figure 1.13: Cybersecurity assurance through process and technical assessments

These approaches, while different in scope and methodology, share a basic principle in that they are both carried out by an independent team that is motivated by discovering process-related and technical gaps. The first approach relies on performing a product assessment against a cybersecurity management system to determine if the performed cybersecurity activities were adequate in reducing the risk to a reasonable level. In the context of ISO/SAE 21434, this assessment would examine the cybersecurity case and the evidence provided to assert that the product is suitable for release given the assumed operational environment.

The second approach relies on performing penetration testing to verify that cybersecurity risks have been sufficiently treated through exposure of the system to hostile conditions encountered in the real world. The results of such a test provide engineering teams with insights into technical security gaps in their products either before the product is shipped or before the risk is exposed to the public (in case the product has already been delivered). Now that we understand the scope of penetration testing and the linkage to ISO/SAE 21434, we will explore the various types of penetration test methods.

Types of penetration tests

Penetration tests can vary in scope based on the level of information available to the tester. At one end of the spectrum there is the black box test in which the tester is given very limited information about the target, and on the opposite end of the spectrum is the white box test in which the tester is given almost unlimited information about the target. Each test type has advantages and disadvantages, which we will explore next.

White box penetration test

White box penetration testing is a comprehensive security assessment methodology where testers have complete knowledge of the system's internal structures, including source code, architecture, and design documents. This approach allows for an in-depth examination of potential vulnerabilities by analyzing code for security flaws, reviewing system configurations, and understanding interactions between various vehicle components. The process typically involves detailed code reviews, static and dynamic analysis, and the simulation of attack scenarios using all available information. This level of access enables testers to uncover vulnerabilities that might be overlooked in other types of penetration tests due to the technical depth required.

Despite its effectiveness, white box testing presents several challenges, including the significant time and resources required, as well as the need to handle sensitive proprietary information responsibly. Moreover, when commissioned by an OEM, white box testing may not be practical if the supplier is unwilling to share the code or design documentation. While it is understandable that organizations may hesitate to hire third-party testers for white box pentesting due to concerns over proprietary information, this may be unavoidable when

targeting high-value components that require specialized security knowledge not available in-house. For example, a company might engage a third party for an in-depth security analysis of bootROM code before it is committed to silicon, or for an analysis of secure IP before licensing it to multiple chip vendors.

Information commonly provided for white box or gray box testing includes CAN database files, CAN topology and wiring diagrams, ECU schematics, Hardware/Software block diagrams, security application notes, vehicle tear down information to find access to the proper networks, TARA, firmware images or binary files, and source code.

Black box penetration test

Black box penetration testing is a security assessment methodology where testers evaluate a vehicle's electronic systems without prior knowledge of their internal architecture, code, or proprietary protocols. In a sense, this type of penetration test simulates an external attacker's perspective who will likely start from a position of very limited knowledge about the target before attempting to discover exploitable vulnerabilities. The one main difference between the two is that the real-world attacker may be willing to spend significantly more time and resources than a professional penetration test team can under a limited time contractual agreement. This approach involves steps such as reconnaissance to identify external interfaces (such as CAN bus access points, OBD-II ports), vulnerability scanning through protocol analysis and firmware extraction, exploitation attempts like fuzz testing and wireless attacks, and post-exploitation analysis for privilege escalation and persistence mechanisms.

Typically, a black box test presents unique challenges like hardware accessibility, lack of familiarity with proprietary protocols, and unknown safety risks for certain testing operations. Therefore, it is desirable to perform the tests in a controlled environment, like **Hardware-in-the-Loop (HIL)** system, while collaborating with manufacturers to maximize the benefit of the test. OEMs and suppliers alike resort to black box testing when they have a limited time before product launch and when their staff is unable to provide extensive support to penetration testers. As a result, they rely on black box testing as a final step before launching their product to gain confidence in their product's security.

> **Note**
>
> If a professional pentest team performs two weeks of black box testing, it is unlikely they will uncover many weaknesses in that time frame, leading the automotive product manufacturer to believe they are totally secure when in fact they most likely are not.

Gray box penetration test

In **gray box penetration test**, testers have partial knowledge of the system's internal structures such as vehicle E/E architecture diagrams, protocol specifications, or code snippets, which allows them to focus on critical areas while still simulating realistic attack scenarios from an outsider's perspective. They may also be given access to special equipment and tools like special wiring harnesses that give access to the internal vehicle network. This method is particularly effective because it balances thoroughness and efficiency, enabling the identification of vulnerabilities that might be missed in black box testing due to the complexity and proprietary nature of automotive systems, while not requiring the exhaustive resources of full white box assessments.

The process involves analyzing available documentation, mapping key interfaces, conducting threat modeling to identify high-risk areas, performing targeted testing and exploitation based on the partial knowledge, and addressing challenges like limited information access and system complexity by collaborating with stakeholders and handling sensitive information ethically. Consequently, gray box testing allows OEMs and suppliers to uncover more vulnerabilities but still simulate a real-world attacker who may be an insider threat or a highly sophisticated attacker who is spending considerable time to gain the knowledge provided as part of the gray box testing engagement. Even though the engagement time can be limited, with sufficient knowledge about the target, the results can be more meaningful when compared to black box testing without placing an extreme burden on the OEM or supplier staff in supporting the penetration testing team.

At this stage, you have established a good foundation for what penetration testing is and the steps involved. Now, it is time to explore the penetration testing phases.

Penetration testing phases

Security standards offer several blueprints for carrying out penetration testing and while they can differ in flavors, they always cover four main phases as shown in *Figure 1.14*:

- Planning and scoping the target system
- Discovery through information gathering and vulnerability identification
- Building attacks and exploits
- Reporting and communication

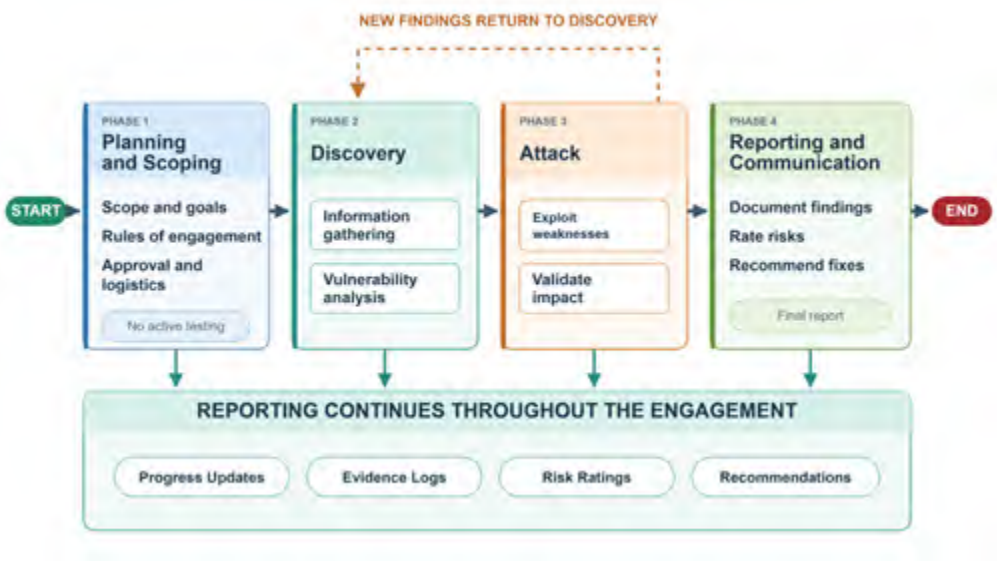


Figure 1.14: The four major phases of penetration testing

We will cover each of those phases now.

Planning phase

This phase involves setting the groundwork for a successful penetration test by defining rules, finalizing management approval, and establishing testing goals. The planning phase does not involve any actual testing activities. Instead, it focuses on developing a comprehensive strategy, including defining the scope, objectives, limitations, and rules of engagement (ROE) for the test. Detailed logistics, such as scheduling, resources, and coordination with involved parties, are also determined during this phase.

Discovery phase

The discovery phase includes two primary components: information gathering and scanning and enumeration.

Information gathering

This is the initial part of discovery and involves collecting data about vehicle E/E architecture, including external vehicle interfaces, backend services, operational environment tools, in-vehicle network protocols, ECUs, and ECU functionality. Depending on the penetration test type, such as white box, gray box, or black box testing, information gathering may consume a significant portion of the engagement.

Scanning and enumeration

This step builds on information gathering by identifying reachable components, exposed services, communication behavior, software or firmware versions, diagnostic functions, network messages, and other observable characteristics of the target system. Scanning may be passive or active.

Passive scanning involves listening to existing communication without sending test traffic, such as observing CAN, Ethernet, wireless, or backend interactions to infer system behavior. **Active scanning** involves sending messages or requests and analyzing how modules respond, such as probing diagnostic services, backend APIs, or networked ECUs.

Automated vulnerability scanning tools are useful during this phase, especially for identifying known weaknesses, misconfigurations, exposed services, vulnerable software versions, or issues documented in public sources such as the NVD. However, these tools do not discover undocumented or unknown vulnerabilities on their own. Finding unknown vulnerabilities depends heavily on the tester's experience, creativity, and ability to interpret results, notice unusual behavior, form hypotheses, and dig deeper into areas of interest.

In automotive penetration testing, limited time often means that known vulnerabilities and clearly exposed attack surfaces remain the primary focus. Deeper research into unknown vulnerabilities becomes more prominent when known issues are limited, when standard scan tools cannot be applied effectively, or when the engagement specifically requires advanced vulnerability research.

Attack Phase

The attack phase involves exploiting vulnerabilities or suspicious behaviors identified during discovery to determine their impact and verify exploitability. This may include gaining unauthorized access, escalating privileges, bypassing security controls, accessing sensitive information, or attempting malicious vehicle control within the agreed rules of engagement.

In practice, this phase is highly iterative. Attackers rarely move through discovery once and then attack only once. Instead, exploitation often reveals new information, which sends the tester back into discovery. A successful diagnostic bypass might expose additional ECU functions, and access to one backend service might reveal new APIs. Similarly, reverse engineering one firmware image might uncover another component worth testing.

The process often cycles between discovery, attack, renewed discovery, and refined attack until the tester has sufficiently explored the reachable attack paths. Common issues examined during this phase include hardware weaknesses, network misconfigurations, buffer overflows, insufficient input validation, race conditions, insecure diagnostics, weak authentication, and processor or memory protection flaws.

Reporting Phase

The reporting phase is integrated with the other three phases and involves documenting findings and progress continuously. Reports typically include logs of activities, lots of screenshots, identified vulnerabilities, risk ratings, and recommendations for mitigating the identified weaknesses. The penetration tester may also aid in root cause analysis to not only determine the exploit but also recommend some mitigations. A final comprehensive report is prepared at the end of the testing process, detailing all findings, methodologies, and suggestions for remediation.

So far, we have learned about the penetration testing types and execution phases that describe how a penetration test is organized. The quality and relevance of its findings also depend on who performs the testing and what adversary capabilities they are expected to represent, which we will explore in the next section.

Choosing a penetration tester

While penetration testing campaigns vary in style and scope, an important deciding factor when choosing a pentester is the attacker profile in scope. For example, if a system is meant to be resilient to advanced persistent threats by nation states, then the penetration test campaign must rely on experienced offensive security professionals who possess capabilities on par with those who work for APT groups to ensure that the test is as thorough and as sophisticated as a real-world attack by an APT group might look like.

If you work in a large organization, you may have a dedicated offensive security team that focuses exclusively on evaluating the security of all cybersecurity-relevant products. These individuals have spent their careers developing deep technical proficiency in identifying vulnerabilities in embedded systems. They understand both the hardware and software architecture down to the finest details, from how a fuse bit is read at the hardware level to how the operating system kernel manages memory page tables. Equipped with an arsenal of tools to reverse-engineer binaries and perform intrusive attacks at both hardware and software levels, they provide a critical layer of security expertise.

However, if you do not have the luxury of such a team, you have two options: hire a third-party pentest team for the duration of the project or create an in-house independent test team—which might be one of the reasons you bought this book in the first place. Keep reading if you aim to build such a team internally.

Note

Even large OEMs with dedicated pentest teams may not be able to cover all their products. Therefore, they often outsource some of their pentesting to third parties

> while their pentest teams focus on managing the third parties and following up on the reported findings.

Knowing the necessary skill set and resources for a pentesting team is a basic step in hiring the right team or building one in-house. Since penetration testing requires a high degree of specialization, it is important to understand when to seek outside help and when it makes sense to invest in internal resources. For example, if you are a silicon vendor looking to identify security vulnerabilities in your hardware design before chip tapeout, it is often advisable to engage with external pentest teams early in the design process. These external teams should have a track record of discovering real weaknesses and producing detailed reports on their analysis and findings.

It's important to note that the success of a penetration test should not be judged solely on the number of vulnerabilities uncovered. A thorough report that shows the breadth and depth of testing, even if no major vulnerabilities are found, should be considered a positive result. This demonstrates that secure engineering lifecycles are functioning effectively. However, if the right team is hired and the system is thoroughly examined for the time needed, they are likely to find something of interest.

White hat vs. black hat hackers

White hat and black hat hackers may use some of the same technical skills, but they differ in intent, authorization, and how they handle discovered vulnerabilities. Black hat hackers exploit vulnerabilities for malicious purposes, such as theft, disruption, unauthorized access, or personal gain. In embedded automotive systems, this could include attempting to access sensitive vehicle data, manipulate vehicle functions, intercept communications, exploit software flaws, or inject malicious code. Their activities demonstrate why robust cybersecurity measures, regular security assessments, and clear vulnerability response processes are necessary.

White hat hackers, by contrast, use security knowledge to identify weaknesses so they can be remediated before malicious actors exploit them. This group includes internal security teams, professional third-party penetration testers hired by an organization, bug bounty participants, and independent security researchers who follow responsible disclosure practices. In formal penetration testing engagements, white hat activity is explicitly authorized and governed by a defined scope, rules of engagement, and reporting process. Independent researchers may not always have prior authorization, but they can still act in a white hat manner when they avoid harm, limit testing to what is necessary, protect sensitive information, and disclose findings responsibly through the proper channels.

When a penetration test or responsible research activity reveals severe vulnerabilities, the findings must be handled carefully. The testing team or researcher should share details only with designated trusted contacts within the affected organization and avoid accidental public disclosure. If a penetration testing team works with multiple customers, it must keep each customer's information separate to prevent information contamination, data leakage, or confusion between engagements.

Organizations can engage the broader white hat community through bug bounty programs, coordinated vulnerability disclosure programs, Pwn2Own events, and private hackathons. Bug bounty programs offer financial rewards or recognition to researchers who report security flaws in vehicle software, hardware, backend systems, or mobile applications. Pwn2Own and private hacking contests provide controlled environments where participants attempt to exploit specified automotive targets under agreed rules. Successful exploits often lead to collaboration with vendors to validate, prioritize, and remediate the identified issues.

These initiatives help uncover critical security issues while also building trust between manufacturers and the security research community. Tesla is a well-known example of an automotive manufacturer that has regularly used bug bounty programs and public hacking events to engage white hat researchers and strengthen vehicle cybersecurity.

Red team vs. blue team

An alternative to hiring third-party penetration testers is to establish an internal pentesting team, often referred to as a **red team**. Red teams operate independently within the organization, typically housed in a division separate from the defensive security teams, commonly known as blue teams. This separation is key to maintaining objectivity and ensuring that security assessments remain unbiased.

The role of the red team is to take on the perspective of a potential attacker, simulating real-world threats by identifying and exploiting vulnerabilities in the automotive product and its supporting systems. This includes testing both known vulnerabilities and attempting to uncover unknown weaknesses (zero-days). Red teams are highly skilled in offensive security techniques, using tools and methodologies that are often similar to those employed by malicious actors. Their mission is to find security gaps before attackers can exploit them, helping the organization bolster its defenses proactively.

Conversely, the **blue team** is responsible for defending the system. This includes threat modeling, defining a cybersecurity concept, performing vulnerability analysis both during and post-development, and maintaining the overall security posture. While the red team attempts to break into systems, the blue team works to prevent, detect, and mitigate these intrusions.

Collaboration between the red and blue teams is essential for improving product security. The red team's findings, whether in the form of vulnerabilities, misconfigurations, or weaknesses

in system defenses, are promptly shared with the blue team through formal reporting channels. This process ensures that critical vulnerabilities are addressed in a timely manner. Depending on the organization's structure, this may involve regular debriefs or handover sessions where both teams discuss the findings, the methods used to exploit vulnerabilities, and recommendations for remediation.

In many organizations, the combinations of red and blue team exercises lead to what is known as **purple teaming**, where both teams collaborate closely. This results in continuous improvement of security measures, as offensive and defensive experts work together to identify gaps and ensure they are closed as efficiently as possible. By fostering a culture of internal security testing through red teams, organizations can improve the robustness of their defenses, align with industry best practices, and reduce reliance on external vendors for security assurance.

Summary

In this chapter, we saw why offensive security is essential for understanding how to protect modern vehicles from increasingly sophisticated cyber threats. While defensive security focuses on building and maintaining strong barriers, offensive security emphasizes understanding and exploiting potential vulnerabilities to stay ahead of adversaries. We discussed how a systematic approach to offensive security allows engineers to identify weaknesses in vehicle E/E architectures, protocols, and operational environments before malicious actors can take advantage of them.

The chapter focused on the complexities of domain-based and central computer-based architectures, combined with the ever-expanding attack surface of connected vehicles, which makes it vital for cybersecurity professionals to continuously evolve their defensive strategies. Through the chapter, we established a consensus on the attacker levels to defend against, whether script kiddies, criminal groups, or even nation-state actors, to help align security priorities across an organization. The importance of a well-constructed attacker profile was also addressed, which informs the development of robust defenses and sets realistic goals for offensive security efforts.

Additionally, we covered offensive security methodologies, which provided critical insights into how attackers may compromise a vehicle's security. Whether through black box, gray box, or white box penetration testing, these methods uncover vulnerabilities at every stage of the attack chain, from initial access to privilege escalation and execution of malicious objectives. We hope that by integrating the concepts of offensive security into defensive strategies, organizations can better anticipate threats, enhance vehicle security, and ensure that their systems remain resilient against cyberattacks throughout their lifecycle.

Throughout this chapter, we laid the foundation for many of the concepts that we will need throughout this book. While we introduced the penetration testing phases, we left out the details of how to carry out those phases. This is the focus of the next chapter.

References

[1] Tencent Keen Security Lab, "*Tencent Keen Security Lab: Experimental Security Assessment on Lexus Cars*," Mar. 30, 2020. [Online]. Available: `https://keenlab.tencent.com/en/2020/03/30/Tencent-Keen-Security-Lab-Experimental-Security-Assessment-on-Lexus-Cars/`

[2] Cybersecurity and Infrastructure Security Agency, "*Supply Chain Compromise*," Alert AA20-352A, Jan. 7, 2021. [Online]. Available: `https://www.cisa.gov/news-events/alerts/2021/01/07/supply-chain-compromise`

[3] Google Project Zero, "*A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution*," Dec. 15, 2021. [Online]. Available: `https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html`

[4] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh, "*Technical Guide to Information Security Testing and Assessment*," NIST Special Publication 800-115, Sep. 2008. [Online]. Available: `https://doi.org/10.6028/NIST.SP.800-115`

[5] JPCERT/CC, "*Japan Computer Emergency Response Team Coordination Center (JPCERT/CC)*." [Online]. Available: `https://www.jpcert.or.jp/english/`

[6] ISECOM, "*OSSTMM 3 - The Open Source Security Testing Methodology Manual*," 2010. [Online]. Available: `https://www.isecom.org/OSSTMM.3.pdf`

[7] National Institute of Standards and Technology, "*National Vulnerability Database (NVD)*." [Online]. Available: `https://nvd.nist.gov/`

[8] MITRE, "*Common Attack Pattern Enumeration and Classification (CAPEC)*." [Online]. Available: `https://capec.mitre.org/`

[9] ISO and SAE International, "*ISO/SAE 21434:2021 Road vehicles - Cybersecurity engineering,*" 2021. [Online]. Available: `https://www.iso.org/standard/70918.html`

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

2

Penetration Testing Phases

Over the years, penetration testing has evolved into a systematic process for identifying components of the system that are critical for security testing, discovering vulnerabilities in those components, constructing exploits against those components, and responsibly reporting vulnerabilities along with suggested mitigations to the impacted stakeholders. In *Chapter 1*, we introduced the basic blueprint for penetration testing starting with planning and discovery, followed by attack creation and finally reporting. In this chapter, we will take a closer look at each of those phases and evaluate how they apply to the connected vehicle and its environment.

While doing so, we will learn about how to prepare a comprehensive plan that covers the scope of the penetration test and the rules of engagement. Then we will explore discovery from both passive and active reconnaissance perspectives. Next, we will discuss some attack techniques to carry out successful exploits in an iterative fashion until the attack objective is achieved. Finally, we will cover post-exploitation activities including reporting and remediation. We understand that many topics introduced in this book can take up chapters on their own. Therefore, our intent in this chapter is to introduce concepts that are considered prerequisites for building practical attack chains which will be the focus of part two of the book.

In this chapter, we will cover the following main topics:

- Planning the penetration test
- Discovery of security weaknesses
- Attack phase
- Reporting vulnerabilities

We hope that by the end of this chapter you will be ready to execute all the phases of the penetration testing lifecycle starting with the planning phase.

Planning the penetration test

When adhering to a systematic cybersecurity engineering process such as ISO/SAE 21434, we are required to plan for independent security testing during the cybersecurity planning phase. The **planning phase** sets the strategic direction for the penetration test by outlining the evaluation scope and determining how the engagement will be conducted. Careful planning ensures alignment between testers and product stakeholders, which is a crucial aspect given the complexity of interdependencies of automotive embedded systems and their communication interfaces, as well as their external supporting systems. However, determining the precise timing for conducting the penetration test can be challenging. Ideally, testing should be scheduled near the end of development when either minimal or no further software or design changes are expected to impact the security performance of the system under test. This approach requires incorporating a buffer period between the design freeze date and the final product release date to allow for testing and implementing any necessary fixes.

With the advent of software-defined vehicles, automotive software is often released continuously, making it difficult to pinpoint the perfect window for a penetration test. While this continuous release model can be advantageous for addressing security vulnerabilities, when it comes to safety-critical systems, continuous software updates present unique challenges. For one, safety qualification activities must be carefully coordinated with security patch releases to prevent the inadvertent introduction of quality or safety issues while fixing security vulnerabilities. This necessitates close collaboration between safety and security teams to ensure that updates that enhance security do not compromise safety integrity and vice versa. Consequently, the planning phase extends beyond merely defining objectives and scope. It also involves agreeing on a suitable cadence that considers other product requirements such as safety assessments, quality testing, and overall release planning.

Defining the engagement

Penetration testing planning begins with defining the engagement, which refers to the specific penetration test project tailored to meet the needs of the automotive manufacturer or supplier. The objectives of the test must be explicitly defined, focusing first on the most critical systems and components. The **Rules of Engagement (ROE)** outlines how the testing team should operate, including when and where testing can occur, for example, during vehicle idle times or in a test lab, and the extent to which testers can simulate an attack. For instance, a penetration test might target the vehicle's infotainment system but refrain from tampering with braking or steering systems.

Defining the scope

The scope outlines which systems, components, and networks will be tested. This can include everything from the vehicle's internal **Electronic Control Units** (**ECUs**), high-performance units like telematics modules and infotainment systems, to external communication protocols that interface the vehicle with back-end servers and other supporting systems such as diagnostic clients. Proper scoping ensures that all parties are clear on the test boundaries, preventing scope creep during the test execution phase.

Consider the case where you may have agreed to test the resilience of an embedded **Hardware Security Module (HSM)** against side-channel attacks aiming to extract cryptographic key bits. After reporting your results, the component supplier might realize they also wanted to evaluate the HSM's behavior against fault injection attacks such as electromagnetic glitching. If the scope of side-channel attacks was not clearly defined during the planning phase, this misalignment could result in either performing additional unplanned work or leaving the client unsatisfied.

> **Note**
>
> When defining the scope, we cannot ignore mobile applications that control vehicle functionality as they can offer a rich attack surface for adversaries.

Defining testing conditions

In addition to defining the areas to be tested, it is essential to clarify the conditions under which those tests will be performed. For instance, using the example given above, you must determine whether HSM is evaluated with production firmware equipped with side-channel mitigation features or with specialized firmware that isolates hardware behavior. This decision can significantly shape both the level of expertise needed and the overall test effort. Moreover, you are encouraged to use both development and production systems to perform the penetration test in a holistic manner.

A development device might intentionally retain access for example to SSH or unlocked debug ports to facilitate deeper exploration of potential exploits, whereas a production device can confirm that such interfaces are effectively restricted in line with defense-in-depth principles. Given that automotive products often remain in service for a decade or more, it is prudent to assume that outer layers of any defense-in-depth strategy will eventually be compromised. Therefore, performing the test on a development test device simulates that scenario, allowing you to verify that the product's layered defenses continue to hold up under more sophisticated attacks.

> **Note**
>
> To conduct penetration testing thoroughly, consider providing the tester with two or three systems. Although a properly scoped pentest reduces the likelihood of damage, it can never be fully ruled out. Multiple devices help maintain a backup unit for testing and allow parallel testing by multiple team members which is particularly beneficial under time-bound conditions.

On the other hand, when evaluating an integrated vehicle, it is vital to decide if testing will occur while the vehicle is in motion or within a controlled environment such as a garage. The scope should consider all technical constraints that protect safety-critical functions, ensuring these remain operational throughout testing. If the testing occurs on a bench in a laboratory setting, we need to ensure that safeguards are in place to prevent permanent system damage. Finally, for safety-critical components, you are strongly advised to rely on simulations and bench setups to avoid triggering unintended hazardous events, such as accidental airbag deployment or compromised braking functionality.

Information sharing and test types

A common oversight during the planning phase is defining the method and scope of information to be shared after the test is complete. Suppose you are asked to perform a penetration test of a central gateway ECU to identify software vulnerabilities that could be exploited to establish presence on the vehicle's **Controller Area Network (CAN)** bus. Having a prior agreement on the type of penetration test, whether it is black box, white box, or gray box, is crucial to ensure that resources are allocated appropriately and the engagement time is accurately estimated.

If the product team agrees to a white box test, they must be prepared to share detailed work products such as schematics, design documents, CAN database files, and even source code. Conversely, if the test is limited to black box testing, the system should be provided to the test team with basic instructions for setup and usage. It is highly recommended that information shared with the penetration tester corresponds to the test type requested. This ensures that the offensive security team spends their valuable time discovering vulnerabilities rather than searching for information that might be readily available to a determined attacker in a real-world scenario.

Additionally, since some penetration tests can leave the system in a bricked state (for example, after flashing firmware), the development team must share crucial details on how to avoid damaging an ECU. This prevents wasting precious time debugging issues that do not contribute to discovering genuine vulnerabilities or exploits. It is also important to note that

the development team should ensure that the system provided to the pentesting team is functional and when needed, they provide the necessary support to keep the target system in a working state.

> **Note**
>
> Ignoring special requirements of target setup often leads to damaged or bricked ECUs. Therefore, planning for the proper setup of the target is essential to avoid delays and added expenses in completing the penetration test.

Legal considerations and documentation

It is crucial to establish a legally sanctioned framework before any penetration testing begins. Obtaining documented approval, typically via a **Master Service Agreement** (**MSA**), a **Statement of Work** (**SOW**), and **nondisclosure agreements** (**NDAs**), not only prevents legal complications but also sets clear expectations on scope, timelines, and deliverables. These documents define the exact tasks to be performed, timelines, deliverables (such as detailed vulnerability reports), and how sensitive data, including proprietary vehicle information, will be handled.

> **Note**
>
> OEMs can request that the pentesting firm abide by frameworks such as TISAX [1] to confirm that the pentester has mature and independently verified controls for securely handling sensitive information throughout the engagement.

Even when the test is performed by an internal team within the organization, the scope must still be documented to avoid misalignment between expectations and results. For example, when working with internal penetration testing teams, it is necessary to agree on the level of access they will receive to the system, the duration of the engagement, and the level of detail required in the final report.

As we will see in the discovery phase, certain reconnaissance activities risk infringing intellectual property rights or intruding on user privacy. When tests involve direct interaction with active vehicle systems, there is a risk of triggering safety-critical malfunctions. Therefore, you must define clear boundaries for penetration testers, particularly when critical systems lie outside the scope of the engagement.

Engaging legal advisors early is vital to ensure contracts include indemnity and limitation of liability clauses, especially for more invasive tests that could expose proprietary data or damage live systems. When external testers are hired, both parties' legal teams must work

together to finalize NDAs and address privacy concerns, including adherence to data-protection regulations such as GDPR. For global automotive manufacturers, cross-border data flows must be managed to comply with export control laws and varying international privacy standards.

Additionally, new regulations, such as recent Chinese rules [2] prohibiting exposure of local road-user data to foreign entities, emphasize the importance of aligning penetration testing practices with local and international legal requirements. If a discovered vulnerability later causes an in-field breach, contracts should define the responsibilities and liabilities of the parties involved. Thus, while securing the vehicle remains the primary goal, a solid legal foundation is equally essential to minimize exposure to legal and financial risks.

Note

Penetration testing done on live systems such as an active OTA backend can trigger the bricking of thousands of vehicles if not performed carefully. Having a solid legal agreement in place can help avoid expensive litigation when parties disagree on who bears responsibility for unexpected damage.

Prioritizing components for testing

Another critical aspect of the planning phase is selecting which components to test. Since penetration testing is often constrained by time and cost, it is important to prioritize components and systems that are critical to the vehicle's operation and safety. Here, risk exposure plays a key role in setting testing priorities. During the **Threat Analysis and Risk Assessment** (**TARA**) process, certain components may have been identified as containing high-value assets or having complex architectures likely to harbor software or hardware vulnerabilities. This information is useful for prioritizing such components for penetration testing. Additionally, new features that have not been independently evaluated before should be given higher priority, as they are more likely to contain unknown vulnerabilities.

Defining deliverables and reporting

Effective communication is vital during the testing of a complex, real-time system like a vehicle. The penetration testing team must maintain regular communication with a trusted agent inside the manufacturer or testing lab. This agent acts as a liaison, ensuring the testers have the resources they need while keeping key stakeholders informed of progress and findings. Without proper contacts, the test team may spend unnecessary time engaging the wrong individuals in search of vital information such as flash programming procedures or power-up sequences of complex ECUs which are prerequisites to continue their tests.

Before the conclusion of the planning phase, the deliverables of the engagement should be clearly defined. Typically, this includes a comprehensive report detailing any vulnerabilities discovered, their potential impact on the vehicle, and recommendations for mitigating those risks. The format and content of the report must be agreed upon with all stakeholders to ensure the results are appropriately consumed and actionable. Furthermore, the planning phase should specify how and when the results of the penetration test will be shared. For example, stakeholders may request that vulnerabilities be communicated immediately upon discovery.

While understandable from a risk management perspective, moving to the reporting phase before completing the penetration test can distract the offensive security team and interfere with their ability to build extensive attack chains that may leverage discovered vulnerabilities. This underscores the importance of allocating adequate time to complete the test before product development is frozen to allow for remediation if needed. Now that we have completed the planning phase, we are ready to move on to one of the most important phases of a penetration test, which is the discovery phase.

Discovery of security weaknesses

Before initiating an attack on any automotive system, a foundational step is to thoroughly understand the system's architecture, use cases, and interactions with its operational environment. In most basic terms, we cannot effectively attack what we do not understand. This leads us to the **discovery phase**. It is a methodical process where offensive security professionals gather intelligence about the target system, map its attack surface, and identify potential vulnerabilities by examining its software, hardware, communication components, and environmental interactions. Before we delve deeper into techniques for discovering security weaknesses, we must first understand the importance of mapping trust relations in the target system.

Trust mapping in embedded systems

It is self-evident that trust is a cornerstone of any secure system. Rigorous examination of trust relationships within the vehicle helps in identifying potential vulnerabilities that arise from unverified trust, improper boundary enforcement, or over-extended trust relationships. Finding an undiscovered vulnerability can sometimes feel like searching for a needle in a haystack. Understanding the trust dynamics can help narrow the search space to those components that are most likely to produce a successful outcome.

Trust boundaries can be viewed as virtual separation borders between mutually distrustful entities in the system, as shown in *Figure 2.1*.

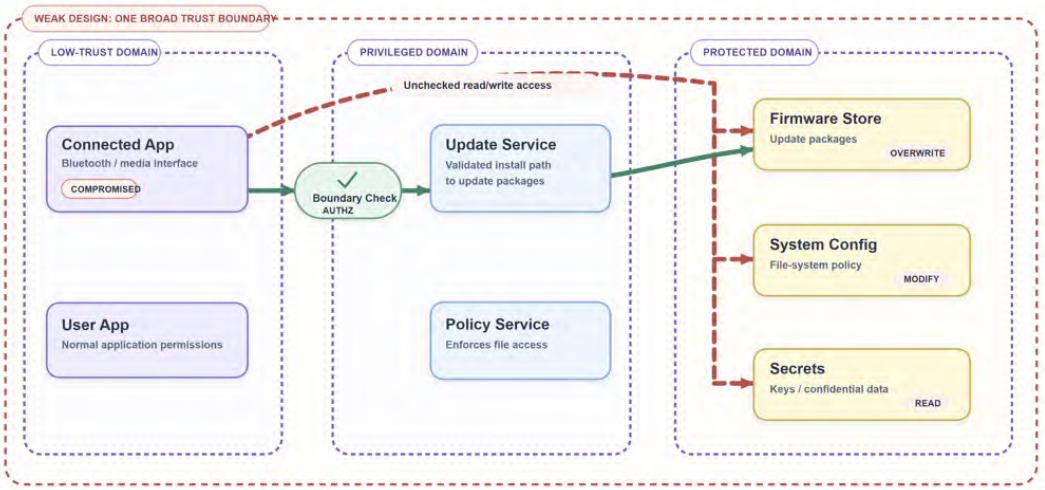


Figure 2.1: Sample trust boundaries inside an SoC

A typical trust boundary is the one between any two ECUs, such as the central gateway and the engine controller. The network link between the two is assumed to be untrusted. If during the discovery phase, we come to learn that ECUs on a single network segment consider that network link to be trusted (thereby foregoing any support for message authentication) then this is an obvious place to investigate security weaknesses.

By simulating trust violations, through man-in-the-middle or replay attacks, we can reveal vulnerabilities due to improper trust enforcement. Naturally, a system designed with a large trust boundary provides more opportunities for exploits when compared to a system where trust boundaries mirror a high level of security isolation. When penetration testers discover a weak trust relation, they can target it with invalid or tampered inputs in the hope of discovering a hidden vulnerability. Consequently, trust examination allows us to discover when critical components are interacting with unauthenticated communication partners, and when they are permitting harmful inputs from external facing systems.

Additionally, mapping trust propagation highlights any unnecessary or overextended trust relationships to identify scenarios where excessive permissions could inadvertently allow unauthorized access to sensitive data. Let's assume a scenario where a high-performance ECU considers all applications running within the **System-on-Chip** (**SoC**) belong to a single trust boundary.

A misconfiguration of the filesystem access controls, such as allowing every application read/write privilege to system configuration files, permits a malicious or compromised application to overwrite firmware updates or extract confidential data reserved for higher-privileged processes. As a result, an adversary who breaches a single low-level application could escalate privileges across the entire system, violating the principle of least privilege and compromising overall security. By examining file system permissions during the discovery phase, penetration testers can gain an understanding about the trust relationships within the SoC as they prepare for exploiting the over-extended trust boundary.

Discovery phase in practice

The discovery phase is often the most time-consuming part of penetration testing, as it involves alternating between information gathering and exploitation attempts. The success of the penetration test hinges on the quality of information collected about the target system. This is especially critical when performing black-box or grey-box penetration testing, where limited information is provided by the component owner.

Consider a scenario where you aim to penetrate a deeply embedded ECU on the CAN bus. Your attack objective is to establish command-and-control capabilities over the CAN network from outside the vehicle. Leveraging your knowledge of automotive networks, you seek either to communicate directly with the target ECU or to identify another ECU that can relay your messages. Your discovery activities involve mapping the vehicle's **Electrical and Electronic (E/E)** architecture to identify access points, such as the **On-Board Diagnostics (OBD)** interface. You may discover a central networking gateway that separates your target ECU from the OBD interface, essentially acting as a proxy. This prompts further investigation to identify the specific CAN messages that the central gateway permits to pass through to reach your target.

By the end of this process, you have collected sufficient information about the CAN message identifiers and their mapping to individual CAN channels to achieve your attack objective. This simple example underscores the importance of the discovery phase in successfully planning and conducting an attack. To carry out a successful discovery phase, we need to first understand the two approaches to discovery: passive and active reconnaissance as illustrated in *Figure 2.2*.

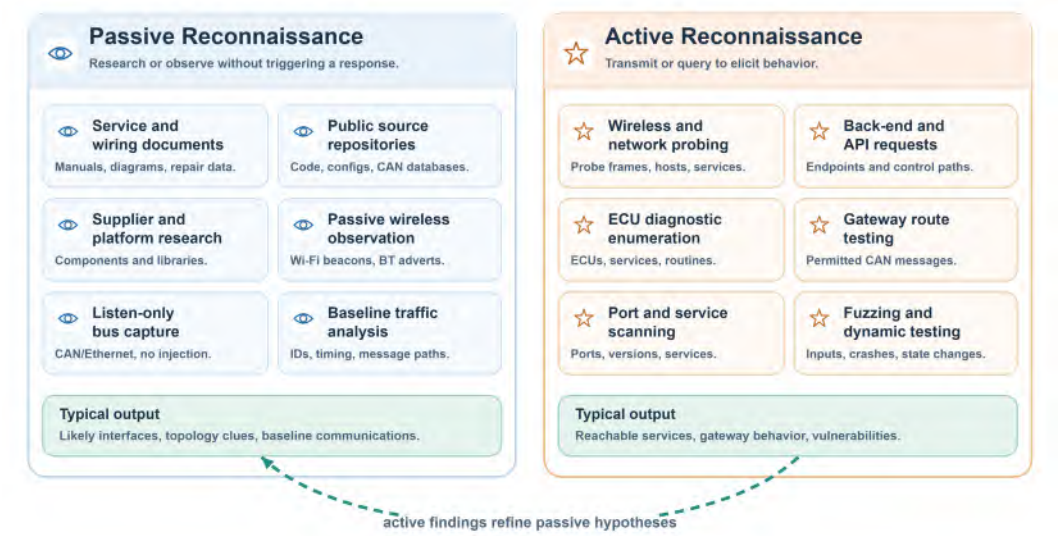


Figure 2.2: Passive and active discovery methods

Note

Look ahead:

In the attack phase, we will learn how to use the ATT&CK framework to identify relevant attack tactics and techniques. In the above example, the attack objective of sending malicious messages to the target ECU over CAN is known as the "Command and Control" tactic (TA0101). On the other hand, the technique used to reach the target ECU is the "Connection Proxy" (T0884), in which you can direct traffic to the target using an intermediary device.

Passive reconnaissance

Passive reconnaissance is the collection of information without directly probing, commanding, transmitting to, or otherwise interacting with the target system in a way intended to trigger a response. Its purpose is to build an initial understanding of the target architecture, exposed interfaces, likely components, software dependencies, communication paths, and potential attack surfaces before moving into offline artifact analysis or active testing.

In automotive penetration testing, passive reconnaissance usually takes two forms. The first is research based on publicly available, commercially available, or previously collected

information. This may include service manuals, wiring diagrams, repair documentation, recalls, patents, supplier announcements, public source code repositories, technical forums, research papers, and aftermarket installation guides. The second is passive observation of the target environment. In this case, the tester may have access to a vehicle, bench setup, wireless environment, or network tap, but only observes existing communications or emissions. The tester does not transmit frames, send diagnostic requests, attempt authentication, or otherwise cause the target to respond.

This distinction is especially important when discussing *scanning*. Listening for Wi-Fi beacons, Bluetooth advertisements, NFC activity, V2X broadcasts, or other radio-frequency emissions is passive. Sending probe requests, crafted wireless frames, diagnostic messages, API requests, or authentication attempts is active reconnaissance. Similarly, observing CAN, Ethernet, or FlexRay traffic is passive only when the tester uses a listen-only configuration or a passive tap that does not alter bus behavior. Once the tester injects messages, queries services, enumerates ECUs, or attempts to traverse network segments, the activity becomes active reconnaissance.

Passive reconnaissance is valuable because it helps the tester form hypotheses before interacting with the target. For example, service documentation may reveal the presence of a central gateway, supplier announcements may identify a telematics chipset, public repositories may expose configuration files or CAN databases, and passive wireless observation may reveal that the vehicle advertises Wi-Fi or Bluetooth services. These findings do not prove exploitability, but they help narrow the search space and guide later testing.

Public and commercial information sources

When information about the target is limited, offensive security teams research their target system by resorting to public and commercially available information sources. One popular technique is researching OEM service portals designed for mechanics and DIYers for wiring schematics, service manuals, and even firmware images for OEM-mandated software updates. For example, Škoda, a car brand that is part of the Volkswagen group, offers engineering documents, wiring diagrams, diagnostic protocols, and repair manuals available for purchase on a short-term or long-term basis through the Škoda Erwin portal [3].

Another popular resource is researching user forums and technical blogs to reveal critical details about system updates or vulnerabilities that can aid in building an attack chain. It is common to find technical documents on user forums from car enthusiasts who share tips such as how to modify calibration files or replace OEM components with aftermarket equipment [4].

If the goal is to compromise the vehicle's cloud backend, public DNS queries may reveal domains used for **over-the-air (OTA)** updates or telemetry services, potentially exposing associated vulnerabilities. Another often-overlooked source of information is social media and

corporate communications. Engineers may unknowingly reveal critical details about their connected vehicle systems when creating job postings, LinkedIn profiles, or even blog posts. This can provide insights into the product's software and hardware architecture.

Another unusual source of information is press releases announcing partnerships between ECU suppliers and component vendors which can reveal details about networking components, memory chips, and software libraries used in specific ECUs. Some software vendors are eager to highlight prominent automotive suppliers that use their products, including success stories that name exact products that integrate their components. This information can be useful if, for example, they offer a library which contains vulnerabilities that you may exploit. In some cases, an OEM will advertise that their vehicle integrates with certain aftermarket brands, such as suppliers of telematics or infotainment units. Acquiring these units is often easier, and their vendors are forthcoming in providing documentation for installation, which can reveal details not only about the unit itself but also about the E/E architecture in which it is integrated.

Yet another useful resource is public source code repositories. In recent years, there have been instances where automotive OEMs have inadvertently provided access to public repositories through misconfigured access settings. For example, in 2023, a major OEM accidentally exposed their keys in a public repository, allowing code to be available for everyone to see [5]. While automotive developers may consider their code confidential, the highly distributed nature of the automotive supply chain increases the chances of code leakage. In addition to code leakage through public repositories, we may find CAN databases for specific vehicle E/E architectures which is a critical first step to constructing malicious CAN frames when spoofing the identity of a CAN node.

As you exhaust the above resources, you may want to consider research papers, presentations, or blogs on older vehicle models or similar platforms which can provide valuable insights into security vulnerabilities that may still be applicable to newer models or models developed by sister companies. Many OEMs share components with subsidiaries so knowing which vehicle platforms are built on shared platforms can prove useful in identifying potential vulnerabilities that apply to your target system.

Passive wireless and RF observations

Passive wireless observation involves listening to radio-frequency activity without transmitting. In the automotive context, this can include monitoring Wi-Fi beacons, Bluetooth advertisements, NFC behavior, key-fob related signals, cellular modem behavior visible from test equipment, tire-pressure monitoring signals, or V2X broadcasts where applicable. The goal is to identify which wireless interfaces are present, when they become active, what identifiers they expose, and whether they reveal useful metadata about the vehicle or its components.

For example, a tester may observe that an infotainment system advertises a Wi-Fi access point, that a Bluetooth interface exposes a device name linked to the vehicle model, or that a nearby backend-connected device communicates at predictable times. These observations help identify likely entry points, but they remain passive as long as the tester does not transmit probes, attempt pairing, authenticate, associate to the network, or send crafted packets.

Passive eavesdropping and baseline traffic capture

Eavesdropping, also known as **network sniffing**, is the passive interception and analysis of communication without modifying the data stream. In connected vehicles, this may include observing traffic on CAN, Ethernet, FlexRay, LIN, V2X, Wi-Fi, Bluetooth, or backend-facing channels. Passive eavesdropping can reveal message identifiers, timing patterns, service dependencies, diagnostic behavior, sensor data flows, and relationships between ECUs.

When examining in-vehicle networks, testers may use tools such as CAN analyzers, Ethernet taps, packet captures, or logic analyzers to collect baseline traffic. For CAN, the interface should be configured in listen-only or silent mode when strict passive observation is required, because a normal CAN node may otherwise acknowledge frames and influence bus behavior. For Ethernet, passive capture should be performed through a tap, mirror port, or equivalent setup that does not inject traffic.

Baseline traffic capture is often the bridge between passive reconnaissance and later active testing. By observing normal communication first, testers can identify expected message rates, startup sequences, diagnostic sessions, backend connections, and dependencies between ECUs. Later, this baseline can be used to design replay tests, fuzzing campaigns, diagnostic probes, or gateway-filtering experiments during active reconnaissance.

Active reconnaissance

After completing passive reconnaissance activities, offensive security professionals advance to the critical phase of **active reconnaissance**. This stage involves direct interaction with the vehicle's systems to gain deeper insights into potential vulnerabilities. This phase encompasses the active probing of the vehicle's backend systems, external interfaces, in-vehicle networks, and delving deeper into ECU hardware interfaces and software APIs and services.

By enumerating and examining these components, penetration testers aim to identify hidden backdoors, insecure configurations, and exploitable communication protocols. This phase is crucial for determining whether the system harbors vulnerabilities that can be exploited later when building attack chains.

Building a test bench

The way a **test bench** is built often depends on the type of team performing the work. In a professional offensive security engagement, whether conducted by an internal red team or an externally hired penetration testing team, the automotive organization should typically provide the target ECUs, representative vehicle components, wiring diagrams, power requirements, interface documentation, and any required tooling. This helps ensure that the test environment reflects the intended vehicle platform and that testing remains within the agreed scope and safety constraints.

Independent security researchers, students, or hobbyist researchers may not have access to manufacturer-provided components. In those cases, researchers often obtain ECUs, gateways, infotainment units, telematics modules, wiring harnesses, or damaged vehicles from salvage yards or online marketplaces. Purchasing a heavily damaged vehicle from a salvage yard can sometimes be a cost-effective way to build a realistic bench because the wiring harness and many electronic components may remain intact even when the body or drivetrain is unusable.

Having a dedicated test bench allows testers to safely explore vulnerabilities, replay network traffic, perform repeated firmware updates, conduct diagnostic testing, and use intrusive techniques without risking damage to a roadworthy vehicle. It also supports repeatability: the tester can return the system to a known state, reproduce findings, and validate mitigations under controlled conditions.

A test bench is particularly useful because some vehicle functions are not testable when an ECU is isolated. Many ECUs expect periodic messages from other modules and when those messages are missing, the ECU may enter a degraded or timeout state, preventing testers from observing behavior that only occurs during normal vehicle operation. A bench with the right supporting ECUs, simulated messages, power supplies, and network conditions can expose functions and attack paths that would otherwise remain hidden. It also helps avoid practical risks such as draining the vehicle battery, disabling vehicle functions, or leaving a complete vehicle in an inoperable state during testing.

Techniques for active reconnaissance

Active reconnaissance usually progresses from external surface mapping toward increasingly targeted analysis. Testers often begin by capturing baseline traffic, mapping wireless and backend interfaces, and enumerating reachable ECUs and services. Depending on the scope of the engagement, they may move into version-based vulnerability scanning, source or firmware analysis, fuzzing, and, where justified, hardware-focused reconnaissance. In practice, the process is iterative: findings from fuzzing, firmware analysis, or hardware testing often send the tester back to earlier discovery steps with better hypotheses.

Understanding how these techniques apply to each layer of vehicle architecture is a building block for constructing successful attack chains. In later chapters, we will take a deeper look at how these techniques can be leveraged to discover security weaknesses that translate into valid exploits.

Mapping wireless networks in automotive ecosystems

As introduced in *Chapter 1*, modern vehicles are equipped with multiple wireless communication interfaces, such as Wi-Fi, Bluetooth, and cellular networks. These interfaces serve various purposes:

- **Infotainment systems**: Connect to the internet for navigation, streaming, and user applications
- **Telematics systems**: Communicate with cloud services for diagnostics, software updates, and vehicle tracking
- **V2X communications**: Allow vehicles to interact with their environment, including other vehicles and infrastructure

These networks connect vehicles to external infrastructure and cloud-based services, providing potential entry points for attackers.

Once the wireless networks have been enumerated, active scanning enables interacting with these networks to gather information about active hosts, services, and communication protocols:

- **Wi-Fi scanning**: May reveal poorly secured in-vehicle networks or default access points
- **Cellular network interaction**: Could expose flaws in the vehicle's remote communication mechanisms, such as vulnerabilities in cellular modems and protocol security downgrade attacks
- **Protocol analysis**: Understanding protocols used can help identify weaknesses in communication standards or implementation

One common finding during active scanning is the presence of improperly secured Wi-Fi networks, either within the vehicle itself or in the surrounding infrastructure. For example, a vehicle's Wi-Fi connection might be vulnerable due to weak encryption or default credentials, allowing an attacker to connect to the vehicle's network and gain access to sensitive systems.

An attacker who has discovered a specific Wi-Fi vulnerability in a particular vehicle model may choose to drive around searching for that model until they come within Wi-Fi range to carry out their attack. This was demonstrated by hackers who managed to gain access to the Tesla infotainment system by flying a drone carrying a Wi-Fi dongle within the vicinity of the vulnerable Tesla model [6].

Back-end and application API analysis

Since backend connectivity has become quite common, we cannot overlook the reconnaissance activities that include the backend systems. The key discovery techniques include intercepting connections between a vehicle's mobile application and its back-end servers, which can prove quite useful in identifying potential weaknesses. These may include insecure API endpoints that leak sensitive vehicle information or even permit dangerous remote-control functions.

For instance, analyzing traffic between the vehicle central computer and the OEM back-end servers can reveal weak or missing authentication mechanisms, and unencrypted data that could be intercepted by a man-in-the-middle. By emulating vehicle applications that connect to the backend, testers can manipulate variables, simulate different scenarios, and observe how such applications handle data transmission and reception. This analysis helps identify insecure data storage practices, inadequate encryption, and other vulnerabilities that could compromise user privacy and vehicle security.

ECU enumeration and diagnostic discovery

Another primary method used during active reconnaissance is the utilization of diagnostic tools specific to the vehicle's manufacturer to gather detailed information about the installed ECUs and the diagnostic routines they support. By interfacing with the vehicle's diagnostic systems, testers can achieve the following:

- **Identify available ECUs**: Detect which ECUs are present on the in-vehicle network
- **Discover available service routines**: Identify the diagnostic services and subroutines each ECU offers
- **Determine ECU interactions**: Understand how ECUs interact with both onboard and offboard diagnostic clients
- **Map CAN identifiers and supported services**: Acquire knowledge of the CAN identifiers used by diagnostic clients and the services they support

This information becomes valuable when simulating an attacker with access to the CAN network. Consider for example, a network topology where the Telematics ECU and the OBD interface are both connected to the central Gateway ECU. The latter may be designed to block certain diagnostic routines from being relayed to other networks if they originate from the telematics unit. Discovering how the gateway ECU routes diagnostic commands is the first step in constructing an attack chain where a compromised telematics unit can masquerade as an offboard diagnostic client to defeat the filtering mechanism at the gateway.

Network scanning and port enumeration

Assuming that an attacker has established a foothold within the vehicle network, scanning for insecure services or improperly configured communication protocols is a powerful technique

to discover network-based vulnerabilities. Since vehicle networks are gradually starting to mimic those of an enterprise network, common techniques that work on such networks can also be applied to vehicles as we will see next.

Port scanning

Port scanning involves using tools like Nmap to scan for open ports on a vehicle's embedded systems through various exposed interfaces. It is especially effective against high-performance ECUs that offer network services commonly found in a typical computer. This activity can identify services running on a vehicle's internal and external communication interfaces. By interacting with these services, testers can carry out the following actions:

- **Map the system's structure**: Understand how different components communicate and identify network topology
- **Identify potential weak points**: Detect unpatched software, exposed communication channels, or unsecured services
- **Assess service vulnerabilities**: Determine if running services are outdated or susceptible to known exploits

Artifact analysis

Besides the passive and active reconnaissance activities, a great deal of the offensive security team's work lies in the analysis of the discovered artifacts. Depending on the level of information available, the team can resort to several techniques to turn their gathered resources into potential findings.

Static code analysis and information flow analysis

When performing white box pentesting with access to source code, running static code checks, such as CERT C/C++ coding rule checks, is a powerful method for discovering security weaknesses. **Static analysis** inspects a program's code without executing it, focusing on understanding the possible execution paths. When source code is not available, binary-based **static application security testing** (**SAST**) can be employed to examine compiled executable files such as binaries through code disassembly and code structure analysis. Both types of SAST-based testing suffer from the following limitations:

- **High false-positive rates**: Misidentifying non-vulnerabilities as issues due to coding rule violations that lack a practical exploitation path
- **Failure to detect runtime-specific vulnerabilities**: Leading to false negatives due to inability to emulate resource contention and race conditions
- **Manual verification required**: Results cannot be automatically validated necessitating expert review to confirm the issue

AI-enhanced SAST

Generative AI is increasingly being applied to SAST to improve the usefulness of static-analysis results rather than to replace the underlying analysis engine. Traditional SAST tools remain valuable because they can parse code, enforce secure coding rules, and produce repeatable findings. However, they often struggle to determine whether a reported issue is practically exploitable, whether sanitization logic is sufficient, or whether the finding is relevant in the broader system context. **AI-enhanced SAST** addresses this gap by pairing static-analysis output with the contextual reasoning and code-understanding capabilities of large language models (LLMs).

In an AI-enhanced SAST workflow, a conventional scanner first identifies candidate weaknesses such as unsafe memory operations, missing input validation, insecure API usage, hardcoded credentials, or tainted data flows from untrusted sources to sensitive sinks. The LLM then reviews the finding in context, including the surrounding code, call paths, data-flow traces, API documentation, coding rules, and vulnerability descriptions. This allows for richer explanations to assess whether a finding is likely to be a true positive and recommend remediation steps that are specific to the codebase.

Recent research has shown that this hybrid approach is especially useful for reducing analyst workload [18]. For example, LLM-assisted systems such as IRIS [19] combine static analysis with LLM-driven contextual reasoning to detect vulnerabilities across whole repositories, while other approaches use LLMs to triage SAST alerts and reduce false positives. Commercial tools are also moving in this direction. GitHub Copilot Autofix [20], for instance, uses CodeQL code-scanning alerts together with code context and LLM-generated explanations to suggest possible fixes for security findings. While useful to defenders, these tools are equally beneficial to offensive security teams engaging in white box penetration testing. They can help sharpen the focus on the real issues worth pursuing so the team can invest time in building attack chains that are more likely to be fruitful.

Despite these benefits, AI-enhanced SAST should be treated as an assistive capability, not an authoritative security decision-maker on flagging a finding as a true vulnerability. LLMs can hallucinate explanations, misunderstand control flow, miss subtle exploitability conditions, or propose fixes that change intended behavior [21]. Therefore, generated findings and fixes should be validated through expert review, regression testing, secure coding checks, and, where appropriate, dynamic testing or fuzzing. Failing to do so, can result in false positive findings that will reduce the credibility of the penetration testing teams when proven to be unexploitable bugs.

Firmware reverse engineering

When source code is not available, as is the case with black box pentesting, **firmware reverse engineering** becomes one of the main methods for discovering code and logical vulnerabilities. Firmware reverse engineering is an effort-intensive process that demands both time and specialized knowledge to uncover vulnerabilities in embedded systems. The process begins with firmware acquisition, where firmware must often be extracted directly from device memory due to its proprietary nature and the manufacturer's efforts to safeguard intellectual property.

Techniques for this include Flash programming, bus monitoring, or **Joint Test Action Group (JTAG)** circuit modification to overcome hardware security features. Since **High Performance Computers (HPCs)** execute from **Dynamic Random-Access Memory (DRAM)**, the use of DRAM interposers is a common technique to extract firmware after it has been loaded and likely decrypted from external flash storage.

> **Note**
>
> While discussing these techniques, it's important to consider ethical guidelines and legal constraints to ensure compliance with intellectual property laws and respect for proprietary technologies.

After acquisition, the next step is firmware unpacking and extraction, which involves breaking down the firmware file to access its contents. This step is often complicated by proprietary formats or encryption, requiring specialized tools like Binwalk [8], or FRAK [9] for analysis. Success in this stage can vary, as even advanced tools may only partially unpack certain firmware types.

> **Note**
>
> Note that firmware reverse engineering relies on methods from both passive and active discovery phases. Once unpacked, binary, and firmware identification is conducted to filter relevant components, such as executable binaries and configuration files.

Understanding the structure and functionality of the firmware is critical for finding potential bugs or backdoors. Reverse engineers face unique challenges distinct from those encountered in standardized formats like **Executable and Linkable Format (ELF)**, which come with descriptive headers aiding in program structure identification. Tackling raw firmware requires a systematic methodology to identify entry points, code segments, and data structures without the usual formatting cues.

Here are some tips for effective firmware reverse engineering:

- **Determine the architecture early**: Use physical inspection when possible, and cross-check with automated tools for accuracy
- **Utilize memory mapping tools**: Tools like BinBloom [10] help deduce memory layout by clustering address values, highlighting probable code areas, data sections, and diagnostic functions.
- **Employ scripting for automation**: Automating the identification of code segments with scripts helps reduce manual errors and speeds up the reverse engineering process
- **Verify function pointers through analysis**: When conventional methods fail, analyzing code segments with scripts can reveal function addresses and entry points

A closely related technique is firmware re-hosting, where embedded firmware is executed or partially executed outside its original ECU environment to support scalable testing. Full emulation is attractive when the processor architecture, memory map, and required peripherals can be modeled accurately enough to run the firmware without the original hardware. This provides strong control, repeatability, instrumentation, and automation.

However, automotive firmware often depends heavily on MCU and SoC peripherals, such as timers, interrupts, watchdogs, and network traffic from other ECUs. When these dependencies cannot be fully emulated, testers may use partial emulation, peripheral modeling, stubbing, or hardware-in-the-loop re-hosting. In these setups, unsupported peripheral access may be simulated in software, replaced with simplified models, or forwarded to real hardware, so the firmware continues to execute in a realistic enough state.

Once the firmware is re-hosted, testers can apply techniques such as fuzzing, coverage-guided testing, symbolic execution, and runtime instrumentation. The practical success of re-hosting depends on how accurately the test environment reproduces the firmware's assumptions about memory, peripherals, timing, interrupts, and vehicle-network state.

Version-based vulnerability scanning

When software binaries are available or obtained through one of the methods mentioned earlier, software composition analysis enables testers to scan the vehicle software for known vulnerabilities. This technique helps identify exploitable software versions, misconfigurations, or outdated services that might provide entry points for an attacker. For instance, **binary scanning** (examining firmware binaries for known vulnerabilities) can reveal weaknesses not apparent through network scanning alone. If we can query the version of a specific software distribution on an ECU, then we can find CVEs without the need to extract the ECU software. This is a common technique applied, for example, when testing Linux-based ECUs.

Dynamic analysis and fuzzing

Dynamic analysis evaluates a program's behavior in real time as the software runs, helping identify vulnerabilities that static analysis may overlook. A key method within dynamic analysis is **fuzzing**, which tests software by generating varied inputs and observing how the system handles unexpected, malformed, or edge-case data [7]. Penetration testers often rely on fuzzing to uncover software vulnerabilities and protocol weaknesses that are difficult to find through manual inspection alone.

The feasible fuzzing scope depends heavily on the tester's access level and on the information provided by the client. In a white-box test, the tester may have access to source code, internal interfaces, file format specifications, protocol documentation, debug symbols, or harnesses that allow direct testing of internal software components. In a grey-box test, the tester may receive selected documentation, representative binaries, seed files, or partial interface details. In a black-box test, fuzzing is usually limited to externally reachable APIs, communication protocols, diagnostic services, file parsers, and observable input channels.

In all cases, the quality of fuzzing results depends on the quality of the inputs. Fuzzing the entire space of possible inputs is usually wasteful, so testers use techniques that focus testing on inputs most likely to trigger unexpected behavior. **Mutation-based fuzzers** generate test cases by taking valid seed inputs and systematically altering them, such as changing byte values, swapping segments, truncating fields, or inserting random data. Because they start from known-valid samples, mutation-based fuzzers can quickly produce malformed but still partially realistic inputs without requiring a complete understanding of the underlying protocol or file format. This approach is especially useful when testers have sample binaries, captured messages, OTA image files, or diagnostic traffic, but do not have detailed specifications.

Generation-based fuzzers create test inputs from scratch using a model of the target protocol, file format, or data structure. They systematically build fields, messages, sequences, and edge cases that follow the expected format while varying values, lengths, order, and validity. This approach can provide deeper and more structured coverage, but it depends on accurate knowledge of the target format or protocol. For example, generation-based fuzzing may be appropriate for software that processes user-provided inputs, such as OTA update images, only when the tester has access to the relevant file format specifications, signing or packaging details, parsing rules, or other internal documentation. This is more common in white-box or grey-box engagements. If such information is not available, mutation-based file fuzzing can still be performed using supplied or collected image binaries as seed inputs.

A common question is how fuzzing helps if the only visible result is a crash. Crashes, reboots, hangs, memory exhaustion, watchdog resets, and unexpected state changes are often indicators that the fuzzer has reached a fragile or poorly handled code path. These symptoms

may point to exploitable vulnerabilities such as buffer overflows, integer handling errors, parser bugs, race conditions, or resource exhaustion flaws. Additional debugging, instrumentation, crash triage, and root cause analysis are usually needed to determine whether the behavior is exploitable.

Fuzzing is also valuable for communication protocols commonly used in embedded systems, including CAN, UART, SPI, and Automotive Ethernet. By sending unexpected message sequences, malformed frames, unusual timing patterns, or invalid field values, testers can observe how ECUs and supporting software handle inputs outside normal operating assumptions. One final fuzzing method worth noting is AI-enhanced fuzzing, which uses machine learning or automated reasoning techniques to guide input generation, prioritize promising test cases, or improve coverage based on observed program behavior.

AI-enhanced fuzzing

AI-enhanced fuzzing can be applied in two related ways. At the protocol level, an AI-assisted fuzzer uses prior knowledge of embedded system vulnerabilities, communication protocols, and command formats to prioritize inputs more effectively than purely random mutation. This makes fuzzing more targeted and context-aware, especially when testing protocol parsers and stateful interfaces.

When source code is available, AI-enhanced fuzzing begins to resemble white-box fuzzing. In that setting, an LLM can analyze the code, identify promising entry points, and assist in generating fuzzing harnesses, seed inputs, or test logic that drive execution toward security-relevant paths. Rather than replacing the fuzzing engine, the AI component improves setup and guidance so that coverage-guided fuzzing can reach deeper program behavior with less manual effort. This trend is increasingly visible in industry tools; for example, code intelligence has described an AI-driven approach in which an agent analyzes code, generates fuzz tests, executes them, and reports discovered vulnerabilities autonomously [17].

For embedded and protocol-driven systems, a layered approach to AI-enhanced fuzzing helps focus testing on distinct classes of weaknesses:

- **Command structure testing**: The fuzzer generates realistic, protocol-specific command sequences to expose flaws in parsing and message handling
- **Parameter bounds testing**: The fuzzer probes minimum, maximum, malformed, and unexpected parameter values to uncover overflow, underflow, and validation errors
- **Timing and sequencing**: The fuzzer varies the order, delay, and concurrency of commands to reveal race conditions, state-machine errors, and synchronization flaws

This layered approach improves efficiency by combining protocol awareness with deeper analysis of likely failure points, and it is especially powerful when paired with source-aware, white-box fuzzing techniques.

Hardware reconnaissance techniques

So far, we have focused our discovery techniques on both software and networking areas, but hardware is also an attractive target for discovery activities due to its foundational role in providing a base security platform.

Hardware-focused reconnaissance usually begins by identifying the device package, processor or SoC family, memory devices, exposed pins, debug interfaces, and security-relevant components. When documentation is available, testers can use it to understand boot modes, fuse maps, JTAG or SWD behavior, secure key storage, and debug access controls. When documentation is unavailable, testers may rely on physical inspection, pinout enumeration, interface detection, and signal analysis to infer how the device is configured.

When attempting discovery techniques against hardware, attackers will resort to three broad categories of attacks, which we will discuss next.

Non-invasive attacks

Non-invasive attacks observe the device without physically modifying the silicon or intentionally disturbing its operation. The objective is often to infer secret information from unintentional leakage. Examples include side-channel techniques such as Simple Power Analysis (SPA), Differential Power Analysis (DPA), timing analysis, electromagnetic analysis, cache-based leakage, and optical observation. These methods are typically used to evaluate whether cryptographic operations leak enough information to recover keys or infer protected state.

Semi-invasive or invasive fault-injection attacks

Semi-invasive or invasive fault-injection attacks intentionally disturb the device to induce errors, skip checks, alter control flow, or bypass security mechanisms. Examples include voltage glitching, clock glitching, electromagnetic fault injection (EMFI), laser fault injection, thermal stress, and memory-disturbance attacks such as Rowhammer. These techniques may be used to test whether an ECU or SoC can be forced past secure boot checks, bypass debug locks, authentication routines, or access-control decisions.

Fully invasive and destructive attacks

Fully invasive and destructive attacks require de-capping, modifying, or directly probing the chip. The objective is to inspect internal structures, read internal data paths, or extract secrets from protected storage. Examples include microprobing, silicon reverse engineering, focused ion beam (FIB) modification, laser ablation, and direct inspection of fuse or one-time-programmable (OTP) memory structures. These methods are costly and specialized, but they may be justified when evaluating high-value assets such as root private keys, master keys, or security configuration bits.

The results of hardware asset extraction and hardware weakness discovery are useful for later phases of penetration testing. For example, discovering that fuse bits can place a chip into an insecure state, that a debug lock can be bypassed through fault injection, or that a cryptographic operation leaks key-dependent information may become the basis for a later exploit. In this sense, hardware asset extraction fits within active reconnaissance because it identifies protected assets, security assumptions, and attack paths that are not apparent from network scanning, firmware analysis, or application testing alone.

Navigating dead ends during discovery

While performing the discovery phase, it's not uncommon to delve into investigative paths that seem promising but ultimately lead nowhere, which is a scenario often described as venturing down a rabbit hole. Recognizing when you're approaching a dead end is essential for effectively managing your time and resources. As penetration testers operate under fixed budgets and tight schedules, it's crucial to adjust strategies promptly to avoid unproductive pursuits.

Making informed decisions about your next steps is also vital to avoid wasting time on unproductive leads. This judgment hinges on experience and the constraints of the engagement. Even if you find yourself on the wrong path, the lessons learned can still be useful for future endeavors. Therefore, knowing when to switch to a different investigative path is critical for avoiding dead ends and maintaining momentum toward your objectives. While perseverance is valuable, it's important to weigh the potential payoff of a particular approach against the time and resources it consumes. As such, setting clear objectives and timeframes for each investigative path helps prevent over-investment in unproductive areas.

Sometimes, during your investigation, you may uncover unexpected findings that, while not directly related to your initial goal, are worth exploring in their own right. For example, while analyzing an image file structure to locate the address where the image hash value is stored, you might discover American Standard Code for Information Interchange (ASCII) logs indicating that your target is using a protocol expected to be disabled in production builds. This discovery is significant and merits further investigation to determine if the protocol can be exploited later.

Every step, whether successful or not, contributes to your overall expertise. Even when a particular path doesn't yield the desired outcome, the insights gained can inform future engagements. For instance, understanding the security mechanisms that prevent firmware extraction via one method might help you identify similar defenses in other devices or suggest new angles of attack.

Importance of strategic decision making

Effective penetration testing hinges on making strategic decisions about where to invest your efforts. This involves the following:

- **Assessing the cost-benefit ratio**: Weighing the potential payoff of an approach against the time and resources it consumes
- **Setting milestones and time limits**: Establishing clear objectives and timeframes to maintain focus and efficiency
- **Documenting findings**: Keeping detailed records of your methods, observations, and results, which is invaluable for refining your strategies over time

This strategic mindset not only enhances the effectiveness of your current engagement but also contributes to your growth as a penetration tester, enabling you to tackle future challenges with increased confidence and skill. As part of the strategic decision-making process, it is often beneficial to include the client in discussions on the progress to assist with priorities and focus areas when facing time constraints.

Attack phase

The natural progression following a successful discovery phase is the preparation of an attack chain that leverages discovered vulnerabilities from the previous phase. Recall from *Chapter 1* that the discovery and attack phases may require multiple iterations before the attack objective is achieved. With each iteration, more vulnerabilities are discovered which in turn allows the attack to proceed further before more discovery is needed to determine how to circumvent additional roadblocks. In this section, we will focus on the steps followed during the attack phase while keeping in mind that transition back into discovery is expected.

Fundamentals of attack chains

Attack chains refer to the sequential steps an attacker takes to exploit vulnerabilities in one or more systems to achieve the attack objective. Rarely does a single weakness suffice to compromise a vehicle. Instead, a series of interconnected vulnerabilities is often required to achieve a successful breach. This multi-step process, known as an attack chain, is key to understanding how complex vehicle systems can be compromised and is vital for both offensive and defensive security professionals alike.

Being able to craft effective attack chains is the result of combining years of expertise, research, and experimentation in offensive security. The goal is to string together various weaknesses in a way that leads to unauthorized access, privilege escalation, and the manipulation of critical systems. In the automotive world, this could range from simply gaining unauthorized access to

an HPC like the infotainment system to more severe actions like altering engine behavior or disabling critical safety functions such as braking.

A critical step in forming attack chains is understanding the vehicle's E/E architecture, which we introduced in *Chapter 1*. When attacking a fully integrated vehicle, we start with choosing an initial access point. This is typically one of the externally visible interfaces such as Wi-Fi, Bluetooth, OBD, USB, etc. When penetrating a single ECU, the same approach applies, but the initial access is relative to the ECU's exposed interfaces such as CAN, Ethernet, JTAG, etc. The same approach applies regardless of whether you are targeting a software or hardware component and irrespective of where the component lies in the hierarchy of the fully integrated vehicle.

This is illustrated in *Figure 2.3* which shows how attackers select exposed interfaces at the layer they are targeting, then chain weaknesses toward their attack objective.

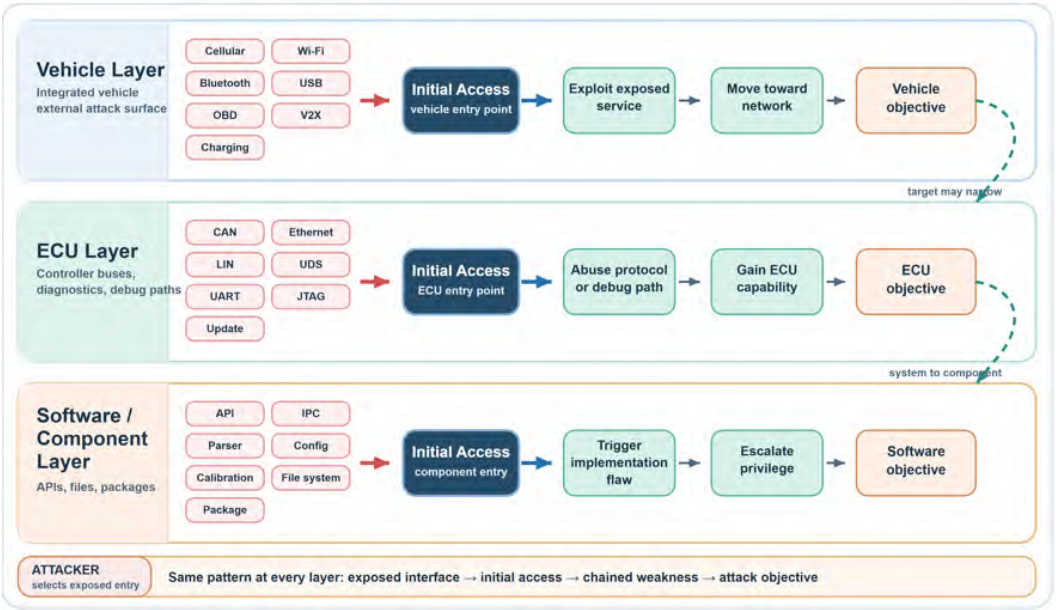


Figure 2.3: Establishing initial access at various levels of the E/E architecture hierarchy

Understanding how attack chains work is crucial for defensive security engineers. By identifying the stages of an attack chain, including initial access, lateral movement, privilege escalation, and execution, defensive measures can be implemented to thwart these attacks. This is achieved through a **defense-in-depth** strategy, ensuring that even if an attacker breaches one layer of security, additional controls prevent them from reaching their ultimate objective.

Moreover, by understanding how attack chains can be designed and used, defensive security engineers can gain deeper insights into how to model threats and create attack paths during the TARA. To help us build effective attack chains, let us first look at some established frameworks for attack path construction.

Frameworks for building attack chains

Leveraging existing attack frameworks can help us organize the steps involved in a successful attack chain in a structured approach. To do so, we will explore two major frameworks in this area.

Common attack pattern enumeration

MITRE's **common attack pattern enumeration and classification** (**CAPEC**) [11] is a comprehensive catalog of attack patterns that provides a structured framework for understanding the various mechanisms adversaries use to exploit vulnerabilities. CAPEC organizes attack patterns into hierarchical categories, from high-level conceptual patterns to detailed, technical attack techniques, enabling security professionals to analyze threats systematically.

Attack mechanisms in CAPEC cover a broad spectrum, including spoofing, where attackers disguise their identities to gain unauthorized access; flooding, where excessive requests overwhelm systems; and code injection, which exploits input validation flaws to execute arbitrary commands. Other mechanisms include resource manipulation, involving unauthorized modification of assets or processes, and probing, where adversaries gather information to identify vulnerabilities. By cataloging these mechanisms, CAPEC facilitates threat modeling, vulnerability assessment, and defensive strategy development for diverse environments.

Being familiar with mechanisms of attack defined by CAPEC is a powerful way to understand how to perform certain attack steps to achieve your attack objective. Granted, not all attack patterns defined in CAPEC are applicable to automotive systems; there exist a fair number of patterns that do.

Take, for example, CAPEC-682: Exploitation of Firmware or ROM Code with Unpatchable Vulnerabilities:

> *An adversary may exploit vulnerable firmware or ROM code that cannot be patched. Devices may be unpatchable because manufacturers intentionally or inadvertently designed them without a software update mechanism. Even when updates are technically possible, a device may become effectively unpatchable if the manufacturer ends support and stops releasing updates.*

The attack pattern first describes the attack method, then explains the typical execution flow. It also provides supporting information, such as prerequisites, required skill level, consequences, potential mitigations, example instances, and related **Common Weakness Enumeration** (**CWE**).

For CAPEC-682 specifically, the example describes attackers discovering devices that run unpatchable firmware affected by an authentication bypass vulnerability. The attackers use that bypass to install malicious code on the devices. Once compromised, those devices scan for other vulnerable devices and recruit them into a larger botnet, which can later be used to perform **Distributed Denial of Service (DDoS)** attacks.

In the automotive context, this attack pattern is especially relevant when BootROM code in an SoC contains a vulnerability that allows malicious code to boot or secure boot protections to be bypassed. To reduce the feasibility of this pattern, teams must rely on both process and technical controls. On the process side, BootROM code should be developed under a secure development lifecycle, with high-assurance design practices, secure coding reviews, and adversarial testing such as fuzzing binary file inputs. On the technical side, even BootROM behavior may be designed with limited patchability through mechanisms such as fuses or **one-time-programmable (OTP)** memory, which can alter certain BootROM decisions if a defect is discovered after the chip enters the operational phase.

MITRE ATT&CK framework

Another useful resource to help in constructing effective attack chains is the **MITRE ATT&CK framework** [12]. This framework is a structured knowledge base that categorizes and describes adversarial **tactics, techniques, and procedures (TTPs)** observed in real-world cyber-attacks. ATT&CK provides a comprehensive map of the steps attackers take once they have gained access to a system, covering everything from initial access to data exfiltration and impact.

The framework is organized into tactics, which represent the goals of the attacker at each stage, and techniques, which outline the specific methods used to achieve these goals. By cataloging these behaviors, ATT&CK serves as a valuable tool for understanding adversarial behaviors to develop successful attack chains. When designing an attack chain, you do not necessarily have to use every single tactic and technique defined by the ATT&CK framework but being cognizant of which step of the attack chain you are in allows you to look up useful techniques that fall within that stage.

For example, when you are aiming to establish initial access within your target system, you have ten techniques that may help you achieve your objective. In the context of automotive systems, one technique that is quite pertinent is T0862 [13], also known as the **supply chain compromise**. Here's an excerpt from the technique description that fits well with automotive systems:

> *Adversaries may perform supply chain compromise to gain control systems environment access by means of infected products, software, and workflows. Supply chain compromise is the manipulation of products, such as devices or software, or their delivery mechanisms before receipt by the end consumer. Adversary compromise of these products and mechanisms is done for the goal of data or system compromise, once infected products are introduced to the target environment... Counterfeit devices may be introduced to the global supply chain posing safety and cyber risks to asset owners and operators. These devices may not meet the safety, engineering and manufacturing requirements of regulatory bodies but may feature tagging indicating conformance with industry standards. Due to the lack of adherence to standards and overall lesser quality, the counterfeit products may pose a serious safety and operational risk.*

This attack technique resonates with the automotive supply chain which suffers from being highly distributed with plenty of opportunities for supply chain compromise. One example of that is the introduction of counterfeit sensors or malicious networking equipment that misbehave under certain conditions to produce a hazardous event. As an offensive security professional, you should try to leverage those techniques that help you with your overall attack objective.

Tactics for the automotive context

Throughout this book, when we present a case study of a real-life attack, we will frame it using the tactics taxonomy from the MITRE ATT&CK framework as it provides a common structure for breaking down an attack chain. Now, we will highlight those tactics that are most useful in the automotive context:

1. **Initial access**: Techniques for gaining unauthorized entry into embedded systems, often exploiting unpatched firmware vulnerabilities, weak default credentials, or open communication ports
2. **Execution**: Methods to run malicious code on the device, such as remote command injection or exploiting insecure update mechanisms
3. **Persistence**: Strategies to maintain access on an embedded system through reboots or firmware updates, including modifying startup scripts or implanting malicious firmware
4. **Privilege escalation**: Approaches for gaining higher-level permissions and privileges, such as by exploiting weak access controls in the guest operating system
5. **Defense evasion**: Techniques to avoid detection by embedded security mechanisms, such as disabling logging or modifying telemetry outputs

6. **Credential access**: Methods to capture device credentials, including extracting them from memory, intercepting authentication protocols, or recovering stored keys
7. **Discovery**: Techniques to gather information on the embedded environment, such as identifying connected peripherals, network configurations, and accessible services
8. **Lateral movement**: Approaches to move across a network of embedded devices, often leveraging weak filtering at the central gateway or Ethernet switches
9. **Collection**: Techniques to gather data from the device, including sensor data, stored logs, or device-specific configurations
10. **Command and control (C2)**: Methods to establish communication with the embedded device from remote systems, often through modified network protocols or insecure services
11. **Exfiltration**: Techniques to extract collected data from the device to an external location, commonly involving encrypted channels or protocol spoofing
12. **Impact**: Actions that compromise device functionality or user safety, such as data tampering, sensor manipulation, or disabling critical components

> **Note**
>
> Since each automotive system has a unique attack surface, it is highly recommended that an offensive security team create a filtered list of the attack techniques that apply to their system. One such initiative has already been performed by the Automotive ISAC through the Automotive Threat Matrix [14]. This can be a great starting point for building a more focused threat matrix for a given product or class of products.

Understanding how attackers move throughout the vehicle E/E architecture to carry out their attack objective is another fundamental concept that is worth exploring. As you read about attack chains, you will often hear the terms lateral movement and pivoting, which we will explore next.

Lateral movement and pivoting in automotive networks

The techniques of **lateral movement** and **pivoting** are crucial for building multi-step attack chains that span multiple vehicle systems. After gaining an initial foothold, attackers often employ these methods to explore and compromise additional systems within the vehicle network, escalating their privileges and targeting critical assets such as the vehicle central computer, telematics systems, and infotainment platforms.

A simple way to distinguish lateral movement from pivoting is to think of lateral movement as the attacker's expansion and pivoting as one way the attacker makes that expansion possible.

For example, if an attacker compromises an infotainment ECU and then uses that access to reach other in-vehicle services, diagnostic functions, or ECUs, the attacker is performing lateral movement because they are moving beyond the original point of compromise. If the attacker uses the compromised infotainment ECU as a bridge or relay to send traffic into a vehicle network segment that was not directly reachable from outside the vehicle, then that infotainment ECU is acting as a pivot point. In this sense, lateral movement describes the broader attack progression from one system to others, while pivoting describes the use of one compromised component as the stepping-stone that enables the attacker to reach those additional systems.

Attack chains at vehicle vs. ECU level

The same attack-chain logic applies whether the target is the vehicle as a whole or a single ECU. What changes is the scope of the attacker's movement and the assets being pursued. At the vehicle level, the attacker typically begins with an externally reachable interface such as Bluetooth, Wi-Fi, USB, cellular connectivity, or the OBD port. After gaining initial access, the attacker executes code or otherwise abuses the vulnerable interface to establish a foothold. From there, the attacker may seek privilege escalation, establish persistence, and move laterally to other ECUs that expose more valuable functions.

In a multi-ECU attack, lateral movement becomes especially important because the attacker's initial point of compromise is often not the final target. For example, compromise of an infotainment ECU may only be the first step toward reaching a safety-relevant ECU. If network segmentation or a central gateway limits direct reachability, the compromised ECU may need to serve as a pivot point into another domain. In this way, a vehicle-level attack chain is best understood as a progression across trust boundaries, network segments, and privilege levels until the attacker reaches the intended function or asset.

At the ECU level, the same sequence appears in a more localized form. Consider a high-performance computing platform running multiple virtual machines or isolated processes. An attacker may first gain code execution in one guest environment, then establish persistence by modifying the file system, and then escalate privileges to the guest kernel or hypervisor layer. If isolation boundaries can be bypassed, the attacker may move laterally from one virtual machine or process boundary to another until the target function is reached. Thus, at the ECU level, lateral movement and privilege escalation may occur within the internal architecture of a single device rather than across multiple physical ECUs.

Seen together, these two perspectives reinforce an important point: attack chains are scale independent. Whether the attacker is moving from one ECU to another across the vehicle network or from one execution context to another within a single ECU, the core logic remains the same: gain access, expand control, overcome isolation boundaries, and progress toward the attack objective.

Attack methods

Once an attacker has identified relevant vulnerabilities, interfaces, or weak security controls, the next step is to select the methods that will advance the attack chain. These methods are not separate from the attack chain but are rather concrete techniques used to achieve code execution, privilege escalation, persistence, lateral movement, or safety impact.

Some attack methods target firmware and software integrity. These include tampering with software images, abusing insecure update mechanisms, performing code injections, and exploiting memory-corruption flaws. Others focus on system state and privilege, such as exploiting race conditions, **time-of-check to time-of-use (TOCTOU)** flaws, weak access controls, or insecure configurations to gain elevated permissions or bypass security checks.

Other attack methods operate at the network and protocol layers. These include man-in-the-middle attacks, denial-of-service attacks, replay attacks, spoofing, and port or service enumeration. Such methods are especially important because many ECUs depend on trusted message flows, timing assumptions, and permissive internal communication patterns.

Attackers may also target supporting security mechanisms rather than application logic directly. Cryptographic weaknesses, such as poor key management, fixed initialization vectors, or insufficient randomness, can enable message forgery, decryption, or impersonation. Likewise, environmental and sensor-based attacks, including signal manipulation and fault injections, can alter how the system perceives the physical world and may create unsafe or exploitable conditions.

Finally, some attack methods originate outside the vehicle itself. Supply-chain attacks, counterfeit components, and maliciously modified tools or subsystems can introduce vulnerabilities before the vehicle is even deployed. For offensive security teams, mastering these methods is a must to be able to construct effective attack chains that combine several of those methods. A real attack may begin with a protocol flaw, use a software exploit for code execution, rely on privilege escalation to deepen control, and then abuse network trust relationships for lateral movement. For that reason, attack methods are best understood as a toolkit that supports each stage of the broader attack chain. Starting in *Chapters 4* through *9*, we will dive deeper into such attack methods and the security weaknesses that they exploit as we explore the security of the various E/E architecture layers.

Note

The Automotive Attack Database [16], chronicles attacks as early as 2010. You are encouraged to survey these documented cases to gain a perspective about what adversaries have accomplished and anticipate what might be feasible in the future.

Reporting vulnerabilities

Effective communication and meticulous reporting are not just procedural formalities but foundational elements of a successful automotive penetration test. Clear communication aligns stakeholders, enables rapid response to critical findings, and maintains the safety and integrity of the testing process. Well-structured reports provide the actionable intelligence necessary for organizations to prioritize security risks and implement the necessary mitigations, thereby enhancing the protection of their connected vehicle systems. This section addresses the key elements of communication during a penetration test, including how to structure written reports and maintain clear lines of communication with all stakeholders throughout the testing process.

Importance of communication during the penetration test

By keeping stakeholders informed, we ensure that everyone shares a common understanding of the testing objectives and any emerging issues. Prompt communication is especially critical when significant vulnerabilities are discovered, enabling swift mitigation to protect vehicle systems.

In an automotive penetration test, it's essential to engage with several key contacts on the client side:

- **Primary contact**: Typically, a project lead or manager responsible for overseeing the entire engagement. This individual focuses on business impact, governance, and ensures that the testing aligns with organizational objectives such as compliance to cybersecurity standards and customer commitments.
- **Technical contact**: Handles the technical facets of the engagement, ensuring that technical questions about the target are directed to the right parties within the organization to avoid unnecessary delays for the penetration test. This individual is usually tasked with collecting data sheets, source code, design documents, software binaries, and target setup information that is required for the penetration testers to perform their work. This person must be aware of the terms of the engagement to avoid oversharing information beyond what the test boundary is.

Communication triggers

As the engagement proceeds, certain events can emerge during a penetration test which may require immediate communication with the primary or technical contacts. One such example is the discovery of critical vulnerabilities in an already shipped product or a product that is close to production. The penetration tester must evaluate such events to determine whether it

is appropriate to halt the test and inform the stakeholders or wait until the test engagement has achieved all its objectives.

Let's assume you were tasked with evaluating the security of bootROM code of an automotive chip to identify vulnerabilities that may defeat the secure boot process. While performing your test, you discover a backdoor that disables secure boot through an accessible procedure. As part of the engagement agreement, you were made aware of the project milestones to understand where the pentest fits within the project schedule. The chip is close to being taped out and given your knowledge about the project schedule, you realize that waiting to disclose the vulnerability will prevent the chip vendor from fixing it in time before further changes are possible. At this point, you must decide to prioritize disclosing the findings to treat the risk.

Outside such critical findings, regular updates such as weekly summaries can provide insights into tasks completed, upcoming plans, and any obstacles encountered. Keeping all parties informed and engaged with the testing process can be extremely useful, especially when roadblocks are faced, such as a device malfunction or an unexpected anomaly detection.

Structuring and presenting findings

Upon completion of the testing phase, all findings must be compiled into a comprehensive and structured report. This report should provide actionable insights to guide remediation efforts and enhance the organization's security posture.

Here are the components of the report:

- **Executive summary**: A high-level overview crafted for management, summarizing major findings and recommendations. This section should clearly articulate the business impact of the identified vulnerabilities, emphasizing their significance in the context of the organization's objectives and risk appetite.
- **Methodology**: A detailed description of the standards and frameworks followed during the test. For example, you may refer to threat classes that are required by UNECE R155, or list FIPS standards that were used in verifying cryptographic modules.
- **Findings**: The core of the report, listing the vulnerabilities discovered along with their severity, potential impact, and recommendations for remediation. Each finding should include the following:
    - **Risk rating**: Assign a risk level based on factors such as exploitability and potential impact, following established frameworks like ISO/SAE 21434 levels or CVSS scores.
    - **Threat level**: Assess the overall threat posed by the vulnerability, considering the likelihood of exploitation.

- **Damage scenarios**: Outline possible adverse outcomes if the vulnerability were to be exploited, helping stakeholders understand the real-world implications.
- **Exploitation details**: Provide technical specifics on how the vulnerability can be exploited, including proof-of-concept examples or attack paths utilized during testing.

Securing and storing reports

Given the sensitive nature of the information contained within penetration testing reports, ensuring their security is paramount. First, the method of sharing the report must be sanctioned by the receiving party, such as a secure company portal. This is to avoid the interception of the report in transit and the leakage of the discovered vulnerabilities to the public.

Second, the report itself must be marked as confidential or highly confidential using the appropriate watermarking to avoid inadvertent sharing outside the authorized parties. A good practice here is to generate the report with a watermark of the recipient's name and email address. This informs the recipient that they are responsible for not sharing it with other parties and if they do, the breach will be traced back to them. While the reports are stored at the penetration testing organization, they must be protected using strong access controls and encryption methods. Ideally, the report should be deleted after a period to avoid inadvertent leakage in a future cybersecurity incident.

Maintaining detailed records of who accesses the reports and when helps with accountability to ensure that any unauthorized access attempts can be tracked and investigated promptly. As the receiver of the report, your responsibility after digesting the results is to triage any reported vulnerabilities and confirm their scoring. Assuming that you are developing your automotive product according to a cybersecurity management system, it is expected that you will follow a **Product Security Incident Response Team (PSIRT)** process to properly ingest and manage the vulnerability report. Following the PSIRT process allows tracking the vulnerability from identification through to final resolution, ensuring that remediation efforts are managed effectively.

Post-remediation activities: ensuring lasting security

The identification and remediation of vulnerabilities are crucial to ensuring vehicle safety and security. Penetration testers are expected to not only offer vulnerability information but also recommendations for remediation action. They may propose a code fix, the addition of a security control, or an architectural design change. In all cases, these recommendations must be evaluated by the receiving parties to prioritize which findings should be fixed first and how to validate that the fix was effective in reducing the risk of exploitation.

Key post-remediation activities include the following:

- **Revalidation testing**: Conduct follow-up tests to confirm that the applied fixes have resolved the vulnerabilities. This may involve reattempting exploits to ensure they are no longer viable. Here it may be necessary to seek the support of the penetration tester to recreate the attack condition. For example, if the attack required specialized equipment such as fault injection or side channel analysis probes, then relying on the penetration testing party for evaluating the effectiveness of the fix may be the only viable option to consider the issue resolved.
- **Root cause analysis**: Once a vulnerability is discovered, a root cause analysis must follow to identify process deficiencies and technical flaws for how vulnerability was introduced in the first place. It may be a deficiency in secure coding practices, or a weakness in the threat modeling process. Whatever the reason, it is important that the organization understands the root cause and prepares a plan for addressing it in future products. In many cases, this may result in updating training materials to ensure that the lessons learned are distributed across the organization and not only the team that experienced the issue.
- **Updating security testing**: A common lesson learned from vulnerability remediation is that if an issue occurs once, it is likely that the mistake behind the vulnerability will be repeated in the future. Besides the root cause analysis and normal remediation activities, it is helpful to evaluate whether security testing procedures can be updated to detect the problem in the future before it makes it to the final product. Defining new test cases that can capture the bug ensures that any future reintroduction of the bug will not go undetected.

Summary

Throughout this chapter, we have seen that penetration testing within the automotive industry is far more than a regulatory requirement or procedural formality. Instead, it is a critical process integral to ensuring the security and safety of modern vehicles. We saw how each phase of the penetration testing process supports the overall objective of identifying and mitigating vulnerabilities before real adversaries can exploit them. By initiating the process with a well-defined plan that carefully considers scope, legal requirements, and risk management, we saw how testers can prioritize the right aspects of the system for evaluation. Moreover, establishing clear objectives and parameters not only guides the testing process but also fosters a shared understanding among all stakeholders to avoid misalignment throughout the process.

During the discovery phase, we examined how employing both passive and active reconnaissance techniques allows testers to comprehensively map the vehicle's attack surface to maximize opportunities for security weakness discovery. By meticulously gathering information, testers can identify potential entry points and weak spots that might otherwise go unnoticed. Then we moved on to the attack phase which builds upon the information gathered during discovery, enabling testers to exploit identified vulnerabilities in a controlled and iterative manner. By simulating real-world attack scenarios, this phase provides invaluable insights into how an adversary might infiltrate the system, emphasizing the importance of proactive detection and remediation of vulnerabilities. Seeing proofs of exploits allows organizations to appreciate the feasibility of the findings and understand the broader impact of their security decisions on preventing similar issues in the future.

Finally, we moved to the reporting phase and the follow-up actions. A comprehensive report not only highlights the discovered vulnerabilities but also provides actionable remediation steps. By offering practical guidance and prioritizing risks based on severity and potential impact, the report aids automotive OEMs and suppliers in strengthening their defenses proactively. We also saw the importance of linking the penetration testing engagement to the PSIRT process to track vulnerabilities until resolution. Finally, performing the proper root cause analysis is essential to strengthen both process measures and technical controls to reduce the likelihood of reemergence of the same issue in the future.

We avoided diving deep into the tools used in each phase of the penetration test in this chapter because we wanted to draw your attention to the fundamental concepts and techniques of the penetration testing phases. In the next chapter, we shift our focus to the various tools that are available to carry out the discovery and attack methods introduced in this chapter.

References

[1] ENX Association, "*TISAX - Trusted Information Security Assessment Exchange*." [Online]. Available: `https://www.enx.com/tisax/`

[2] State Administration for Market Regulation and Standardization Administration of China, "*GB/T 44464-2024: General requirements of vehicle data*," 2024. [Online]. Available: `https://www.chinesestandard.net/PDF_MS.aspx/GBT44464-2024`

[3] Volkswagen Group Import China, "*VGIC Skoda erWin Online*." [Online]. Available: `https://vgic-skoda.erwin-store.com/erwin/showHome.do`

[4] NICOclub, "*Factory Service Manuals*." [Online]. Available: `https://www.nicoclub.com/service-manual`

[5] P. Roberts, "*Lessons from the Mercedes-Benz GitHub source code leak*," ReversingLabs, Feb. 1, 2024. [Online]. Available: https://www.reversinglabs.com/blog/lessons-from-the-mercedes-benz-github-source-code-leak

[6] R.-P. Weinmann and B. Schmotzle, "*TBONE - A Zero-Click Exploit for Tesla MCUs*," Kunnamon and Comsecuris, Oct. 16, 2020. [Online]. Available: https://kunnamon.io/tbone/

[7] J. Li, B. Zhao, and C. Zhang, "*Fuzzing: a survey*," Cybersecurity, vol. 1, article 6, 2018. [Online]. Available: https://doi.org/10.1186/s42400-018-0002-y

[8] ReFirmLabs, "*binwalk: Firmware Analysis Tool*," GitHub repository. [Online]. Available: https://github.com/ReFirmLabs/binwalk

[9] Red Balloon Security, "*OFRAK: Open Firmware Reverse Analysis Konsole*." [Online]. Available: https://ofrak.com/about/

[10] Quarkslab, "*binbloom: Raw binary firmware analysis software*," GitHub repository. [Online]. Available: https://github.com/quarkslab/binbloom

[11] MITRE, "*Common Attack Pattern Enumeration and Classification (CAPEC)*." [Online]. Available: https://capec.mitre.org/

[12] MITRE, "*MITRE ATT&CK*." [Online]. Available: https://attack.mitre.org/

[13] MITRE, "*Supply Chain Compromise, Technique T0862 - ICS*," MITRE ATT&CK. [Online]. Available: https://attack.mitre.org/techniques/T0862/

[14] Automotive ISAC, "*Automotive Threat Matrix.*" [Online]. Available: https://atm.automotiveisac.com/

[15] J. Jepson, R. Chatterjee, and J. Daily, "*Commercial Vehicle Electronic Logging Device Security: Unmasking the Risk of Truck-to-Truck Cyber Worms*," in Proc. VehicleSec 2024, San Diego, CA, USA, Feb. 2024. [Online]. Available: https://doi.org/10.14722/vehiclesec.2024.23047

[16] IEEM-HsKA, "*Automotive Attack Database (AAD)*," GitHub repository. [Online]. Available: https://github.com/IEEM-HsKA/AAD

[17] Kazankova, Natalia. "*Meet Spark, an AI Test Agent That Autonomously Uncovers Vulnerabilities.*" Code Intelligence, 29 Jan. 2025, www.code-intelligence.com/blog/meet-ai-test-agent-to-find-vulnerabilities-autonomously.

[18] Keltek, M., Hu, R., Fani Sani, M., and Li, Z. (2024). *Boosting cybersecurity vulnerability scanning based on LLM-supported static application security testing*. arXiv. https://doi.org/10.48550/arXiv.2409.15735

[19] Li, Z., Dutta, S., and Naik, M. (2024). *IRIS: LLM-assisted static analysis for detecting security vulnerabilities*. arXiv. https://doi.org/10.48550/arXiv.2405.17238

[20] GitHub Docs. (n.d.). About Copilot Autofix for code scanning. GitHub. Retrieved May 12, 2026, from https://docs.github.com/en/code-security/concepts/code-scanning/copilot-autofix-for-code-scanning

[21] Du, X., Feng, J., Zou, Y., Xu, W., Ma, J., Zhang, W., Liu, S., Peng, X., and Lou, Y. (2026). *Reducing false positives in static bug detection with LLMs: An empirical study in industry. arXiv*. https://arxiv.org/abs/2601.18844

Get this book's PDF version and more

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

3

Building the Tool Arsenal

In the first two chapters, we introduced the blueprint for performing offensive security activities. In this chapter, we focus on the tools that make it possible to mount our attacks on various layers of the vehicle architecture. You may treat this chapter as a curated offensive security tool map that should serve as a reference for you to revisit as you read *Chapters 4* through *9*.

To help you build your arsenal of offensive security tools, we will survey tools that cover every aspect of the vehicle architecture starting from the cloud, the wireless network layer, the in-vehicle network, the MCU and SoC based ECUs, and finally the sensor and AI/ML components. Because AI has changed the game of offensive security, wherever feasible, we will point to AI-assisted or -driven tools that augment or replace traditional tools.

By the end of this chapter, you should have a fairly good idea about the tooling needed to carry out an offensive security testing campaign as well as how to start performing ethical vehicle hacking. Note that the emphasis will be on the tools and techniques and not specifically on building attack chains that leverage those tools. Throughout the chapter, we will refer to the tools according to the two main phases:

1. **Discovery** (reconnaissance and analysis): making the target observable, enumerating interfaces, mapping services, extracting firmware, and building accurate hypotheses
2. **Exploitation and validation**: controlled manipulation to confirm weaknesses, measure impact, and generate test evidence and reproducible artifacts

In the rest of this chapter, we will group the tools along the following vehicle architecture layers:

- Vehicle backend tools
- Wireless interface tools
- In-vehicle network tools

- High-performance and MCU-based ECU tools
- Vehicle sensor tools
- How AI is changing penetration testing

Vehicle backend tools

Connected vehicles, as we will see in *Chapter 4*, rely on backend servers and cloud platforms (e.g., AWS IoT Core, Azure IoT Hub) for telematics, data ingestion, remote control features, and fleet management. Penetration testers targeting the vehicle backend must assess cloud APIs, communication links between vehicles and cloud, and the cloud infrastructure configuration. Tools in this area help intercept and analyze backend communications (discovery), audit cloud configurations, and exploit cloud-side vulnerabilities.

In this section, we will focus on the tools organized by assessment categories of cloud and backend systems.

External surface discovery and exposure validation

Internet-wide scanners often appear in automotive backend assessments when organizations need to confirm what is unintentionally exposed. These are not *car tools* per se, but they are commonly used to detect exposed MQTT brokers, API endpoints, or misconfigured public services that can become a vehicle-adjacent risk:

- **Shodan** [1] and **Censys** [2] act as search engines for Internet-connected devices. Penetration testers use these to discover exposed vehicle telematics endpoints or cloud services. For example, a misconfigured MQTT broker or API endpoint could be found via Shodan by querying for unique banners. These help in mapping the attack surface during the reconnaissance phase.
- **Amass** [3] enumerates the attack surface of internet-facing infrastructure through DNS enumeration, certificate transparency log analysis, and ASN lookups, mapping subdomains and IP ranges for a target organization. A pentester can run Amass in passive mode against vehicle backend domains to discover cloud API endpoints, OTA update servers, and telematics gateways not visible through direct web browsing, producing a host-relationship graph that guides subsequent scanning with Shodan and Censys.
- **SecurityTrails** [4] provides a DNS history and OSINT database for retrieving historical DNS records, subdomain lists, WHOIS data, and IP address associations for a target domain. A tester can use SecurityTrails for an OEM vehicle backend domain to discover staging environments, internal-facing services exposed over DNS, and infrastructure

predating the current production deployment, expanding the attack surface beyond what current DNS records reveal.

- **masscan** [5] is a stateless TCP port scanner built for high-speed scanning of large IP address ranges. A security tester may use masscan to rapidly identify all open TCP ports across the IP ranges assigned to a vehicle OEM cloud backend or APN private network, producing a complete service map before targeted enumeration with Nmap.
- **GitLeaks** [6] scans git repositories and commit histories for high-entropy strings and patterns matching API keys, passwords, and certificates across all branches and tags. A pentester can run GitLeaks against OEM source repositories to identify secrets that were committed and not removed, including live API keys for vehicle telematics services that remain exploitable even after being overwritten in the latest commit.
- **truffleHog** [7] searches git commit history and file content for strings with high entropy or matching known credential patterns, covering branches, tags, and stashed changes. A tester can use truffleHog alongside GitLeaks to detect credentials that were added and then overwritten in later commits but remain accessible in the git object store, a common oversight in OEM development workflows.
- **GitGuardian** [8] monitors public and private repositories in real time for exposed secrets and alerts security teams when credentials appear in source code. In a penetration test context, the platform demonstrates to an OEM that their credentials are or have been publicly visible, supporting the finding that secret management practices require remediation.

Vehicle companion apps and backend APIs

The vehicle backend is often accessed via mobile apps. Tools overlap with standard mobile/web API testing:

- **Jadx** [9] decompiles Android APK files to Java source code, including the application manifest, resource files, and all class definitions. A security tester might use Jadx to reverse engineer a vehicle OEM companion application, recovering API base URLs, authentication logic, certificate pinning implementation details, and hardcoded credentials. Jadx is the current standard Android decompiler and supersedes JD-GUI for most APK work.
- **MobSF (Mobile Security Framework)** [10] performs automated static and dynamic analysis of Android and iOS application binaries and runs as a local web application. A pentester can provide the OEM vehicle companion application APK or IPA to MobSF to extract hardcoded API endpoints, authentication tokens, cryptographic keys, and backend server addresses embedded in the application code, providing direct targets for subsequent backend testing without requiring source code access.

- **Magisk** [11] is a systemless Android root framework that roots the host device without modifying the system partition, bypassing many root detection checks that OEM companion applications implement. Once the device is rooted with Magisk, the tester uses tools such as Frida to hook into the application process at runtime, bypass certificate pinning, and inspect API calls and authentication flows that are not observable without root access.
- **OpenSSL s_client** [12] is a command-line **Transport Layer Security** (**TLS**) client that initiates a handshake with a remote server and reports the full negotiated session parameters, including whether the server issues a certificate request to the client. A pentester can run it against a vehicle backend API or OTA update endpoint to determine whether the server enforces **mutual TLS** (**mTLS**), connecting without presenting a client certificate to confirm whether the backend accepts the session and processes commands without validating the caller's identity.

Interception, inspection, and API workflow reconstruction

During the discovery phase, testers often capture the traffic between externally exposed vehicle components (such as mobile apps) and backend services. Intercepting proxies are among the first tools to use because they let the tester understand and manipulate the request/response contract between companion apps, head units, and cloud services:

- **Fiddler** [13] can intercept app-to-cloud control messages, allowing the tester to inspect authentication tokens and encryption. This tool facilitates replaying or modifying requests to test input validation (exploitation of cloud APIs).
- **Burp Suite** [14] provides similar capabilities with an extensible interface for web service fuzzing. Burp Suite documentation describes Burp AI as AI-powered features integrated into Burp Suite to uncover vulnerabilities more efficiently, understand complex web technologies, and streamline authentication setup; it also emphasizes that AI features run only when explicitly activated.
- **mitmproxy** [15] offers similar workflows in open tooling as an interactive, SSL/TLS-capable intercepting proxy for HTTP/1, HTTP/2, and WebSockets, with multiple front ends (console, web, and dump-style) that share the same core proxy engine.
- **Postman** [16] is an API client that allows crafting and sending custom HTTP requests with full control over headers and body. Testers use Postman to replay or modify captured telematics API calls (e.g., unlock car door commands) with various parameters, attempting exploitation (e.g., replay attacks, parameter tampering).
- **OWASP ZAP (Zed Attack Proxy)** [17] is an open-source web application scanner that intercepts, inspects, and modifies HTTP and HTTPS traffic between a client and a server. A security tester configures a vehicle companion application or OTA update

client to route traffic through ZAP, enabling passive scanning for missing security headers and active scanning for injection flaws, broken access control, and unauthenticated API endpoints. ZAP is the direct open-source alternative to Burp Suite for companion app and web API assessments.

- **MQTT Explorer** [18] is a cross-platform client that connects to an MQTT broker and displays all published topics in a tree-structured view, including retained messages and topic payloads. A security tester connects to a vehicle backend broker, such as AWS IoT Core or Eclipse Mosquitto, to observe the topics that ECUs, mobile applications, and fleet management services publish to, enabling identification of unauthenticated topics, cleartext payloads, and sensitive telemetry channels.
- **jwt_tool** [19] is a command-line utility for testing JSON Web Tokens, including automated checks for the non-algorithm attack, RS256-to-HS256 algorithm confusion, and weak signing secrets. A pentester intercepts a vehicle companion application authentication token and runs **jwt_tool** to determine whether the backend accepts a token signed with a null algorithm or a forged signature, which would grant unauthorized access to vehicle control APIs such as remote door unlock or engine start commands.
- **sslyze** [20] performs automated TLS configuration analysis against remote servers, testing for weak cipher suites, protocol downgrade vulnerabilities, certificate validity, OCSP stapling, and deprecated TLS versions. A security tester can run sslyze against vehicle backend API endpoints and OTA update servers to produce a machine-readable TLS misconfiguration report, providing the evidentiary basis for findings related to traffic interception risk.
- **tcpdump** [21] captures network packets from a specified interface and writes them to a **Packet Capture** (**PCAP**) file for offline analysis using a libpcap-compatible filter syntax. A pentester uses tcpdump on a backend server or gateway in headless environments where Wireshark is unavailable, capturing raw **Transmission Control Protocol** (**TCP**) streams, **User Datagram Protocol** (**UDP**) broadcasts, and protocol handshakes for subsequent decoding with Wireshark or Scapy.
- **boofuzz** [22] is a Python-based network protocol fuzzer that supports stateful fuzzing of custom binary and text-based protocols by defining request graphs that model valid session sequences. A security tester may use boofuzz to fuzz vehicle backend REST APIs and MQTT command channels by generating malformed request sequences,

identifying crash conditions and unexpected error responses that indicate memory corruption or logic flaws in backend message handlers.

- **Peach Fuzzer** [23] is a model-based fuzzer that uses XML pit files to define protocol grammar and generate valid and malformed test cases in a structured, repeatable manner. A security tester can use this tool by first writing a pit file describing the vehicle backend API protocol or a custom telematics binary protocol and then run Peach to systematically exercise parsing logic, attempting to identify buffer overflows, integer errors, and input handling failures.

Cloud posture assessment and cloud exploitation simulation

Cloud security auditing tools are valuable because the most common backend weaknesses in practice are configuration and identity mistakes rather than novel zero-day vulnerabilities.

- **Scout Suite** [24] is an open-source multi-cloud security auditing tool that gathers cloud configuration data and highlights risk areas to present a clear view of the attack surface.
- **Prowler** [25] is an open-source cloud security assessment tool that evaluates AWS, Azure, and GCP environments against CIS benchmarks, PCI-DSS, ISO 27001, and custom compliance frameworks. A security tester can run Prowler against the OEM vehicle backend cloud account to enumerate misconfigured IAM policies, overly permissive S3 buckets, unencrypted storage volumes, and logging gaps that an attacker could exploit to gain persistent access or exfiltrate telemetry data.
- **Pacu** [26] is especially useful when moving from audit to controlled offensive simulation. It is an open-source AWS exploitation framework designed for offensive testing against AWS environments using an extensible set of modules.
- **Trivy** [27] is an open-source vulnerability scanner for container images, file systems, and infrastructure-as-code configurations that produces a machine-readable CVE inventory by image layer. A security tester could scan the Docker images used to deploy vehicle backend microservices with Trivy to identify vulnerable base image layers, misconfigured package installations, and exposed secrets within the image file system.
- **OWASP Dependency-Check** [28] analyzes project dependencies and identifies components with known CVEs by comparing library names and versions against the **National Vulnerability Database**. A security tester runs Dependency-Check against the build artifacts of a vehicle backend service or companion application to enumerate vulnerable third-party libraries, such as outdated TLS stacks or authentication frameworks, providing a prioritized remediation list before deployment.

Reference implementations and vendor tools

Cloud providers themselves offer security assessment frameworks as discussed here:

- **AWS IoT Device Defender** [29] is a monitoring service for AWS IoT that can audit IoT configurations and detect anomalous device behavior. While primarily a defense tool, knowledge of Device Defender's checks (e.g., policy audits, certificate expiration) helps testers understand what misconfigurations to look for (discovery phase). For example, a pentester might verify if AWS IoT policies follow least privilege where a violation would be a finding.
- **Microsoft Defender for IoT** [30], formerly known as **Azure Security Center for IoT**, is a cloud-native security monitoring service for IoT and operational technology (OT) environments that audits Azure IoT Hub configurations, detects anomalous device behavior, and maps the attack surface of connected device fleets. A security tester can use its agentless device discovery and configuration audit outputs as a structured starting point for a manual cloud posture review, identifying IoT Hub access policy gaps, unauthenticated device enrollments, and misconfigured device twins before moving to active exploitation testing.
- **Eclipse Mosquitto** [31] is an open-source MQTT message broker that a security tester deploys locally to simulate a vehicle backend broker during controlled testing. Running Mosquitto with default or weakened ACL configuration allows the tester to verify that a vehicle telematics unit or companion application connects to an unauthenticated broker, confirming susceptibility to a rogue broker attack in which an attacker intercepts and modifies vehicle telemetry or command messages.

AI-augmented tools

In backend work, AI is already practical because backend APIs are text-rich, schema-rich, and workflow-driven. Burp AI is an example of AI augmentation that changes real-world workflow: it is integrated into the proxy toolchain and targets speed and understanding rather than "autonomous exploitation" [14]. Postman also now documents AI-related capabilities (for example, "AI Request blocks" in workflows), but the practical security value is usually in workflow acceleration and documentation rather than unique exploit capability; it remains primarily an API client and automation environment [16].

At the cloud configuration layer, some tools market "AI-driven" framing, but the operational value still comes from deterministic checks and enumerations. The defensible way to use AI here is to accelerate triage (summarization of findings, mapping misconfigurations to concrete attack paths), while keeping the underlying evidence anchored in cloud API facts and logs. The following table summarizes representative tools used for API interception, cloud posture

review, offensive validation, exposure discovery, and vendor-native enumeration, along with the phase and target area where each tool is most commonly applied:

Tool	Purpose	Usage Phase	Target	Reference
Burp Suite + Burp AI	Intercept, modify, replay requests; AI-assisted workflow acceleration and analysis	Discovery/ Exploitation	Telematics APIs; mobile-backend traffic	PortSwigger Burp AI docs. [14]
mitmproxy	Intercept/inspect/ modify/replay TLS web traffic with interactive tooling	Discovery/ Exploitation	HTTP APIs used by vehicle apps and services	mitmproxy docs.[15]
Scout Suite	Multi-cloud posture assessment; attack surface mapping from cloud APIs	Discovery	Cloud resources supporting vehicle services	Scout Suite repo [24]
Prowler	Automated cloud security checks and reporting	Discovery	Cloud posture in AWS/Azure/GCP environments	Prowler repo [25]
Pacu	Offensive simulation and exploitation modules for AWS accounts	Exploitation/ Validation	AWS-hosted telematics backend misconfigurations	Pacu repo/ wiki. [26]
Shodan/ Censys	Internet exposure discovery for public endpoints	Discovery	Exposed vehicle-adjacent services and brokers	Shodan site; Censys scanning docs. [1,2]

Tool	Purpose	Usage Phase	Target	Reference
AWS CLI / Azure CLI	Realistic enumeration and action validation using vendor APIs	Validation	Cloud resources, IoT registries, identity, policies	AWS CLI docs; Azure CLI docs [32, 33]

Table 3.1: A survey of common cloud security test tools

Now that we have surveyed offensive security tools in the backend layer, we will turn our attention to the vehicle wireless layer, where the tooling mix shifts toward RF visibility, protocol-specific hardware, and controlled lab networks.

Wireless interface tools

As we will see in *Chapter 5*, connected vehicles expose multiple wireless interfaces, from Wi-Fi and Bluetooth used by infotainment and pairing with mobile devices to cellular links (4G/LTE/5G) for telematics and V2X communications. Penetration testers require specialized **Radio Frequency** (**RF**) tools to discover and exploit vulnerabilities in these channels. This section covers tools for wireless attack and analysis across the radio technologies commonly found in and around modern vehicles, including Wi-Fi, Bluetooth and BLE, cellular telematics links, RFID/NFC, GNSS/GPS, and V2X communication stacks.

Wireless testing has a unique characteristic compared to backend testing: whereas backend compromise is dominated by identity and API correctness, wireless compromise is dominated by physical-layer and network stack realities (range, interference, timing, and protocol state).

Wi-Fi discovery and client-side manipulation

Many connected cars provide Wi-Fi connectivity, for example, a vehicle Wi-Fi hotspot for passengers or Wi-Fi used during software updates/diagnostics. Standard wireless pentest tools apply here:

- **Aircrack-ng** [34] is a suite of open-source tools for 802.11 Wi-Fi auditing. In discovery, airodump-ng can be used to scan for the vehicle's Wi-Fi SSID, signal strength, and connected clients. If the Wi-Fi network uses weak security (e.g., WPA2 with a weak passphrase), testers use aireplay-ng to capture handshakes and aircrack-ng to attempt password cracking. In exploitation, Aircrack's packet injection can replay captured network traffic or deauthenticate clients, which might disrupt vehicle systems using Wi-Fi. This tool is actively maintained and widely used in general wireless security testing.

- **Kismet** [35] is a passive wireless network detector and sniffer, useful for discovery of rogue or hidden networks. Kismet can detect if a vehicle is advertising a hidden Wi-Fi network or if unauthorized Wi-Fi devices (e.g., aftermarket dongles) are present. It logs SSIDs, MAC addresses, and can alert on adhoc networks or unusual beacons. Penetration testers might use Kismet to map the RF environment around a vehicle to identify all transmitting devices (e.g., to find a wireless debugging interface not meant for consumer use).
- **Pineapple (Hak5)** [36] is a specialized hardware device running custom firmware, essentially a rogue access point toolkit. Testers might use it to perform **man-in-the-middle** (**MitM**) attacks on vehicles that connect to known Wi-Fi networks. For instance, if a car automatically joins the owner's home Wi-Fi, a Pineapple could emulate that SSID and capture traffic or perform phishing on the infotainment system. The Pineapple automates discovery and attack (exploitation) phases against Wi-Fi clients.
- **Flipper Zero** [37] supports multiple features including Sub-GHz signal reading and 125 kHz RFID, and Flipper's own product description frames it as a multi-tool for "exploring access control systems and radio protocols". This kind of device is common in field reconnaissance precisely because it reduces setup effort, though serious validation typically still requires more specialized RF and lab-grade tooling.
- **hashcat** [38] is a GPU-accelerated password recovery tool that supports dictionary, brute-force, rule-based, and mask attacks across a wide range of hash types. A security tester can use hashcat to crack WPA-PSK handshakes captured with airodump-ng from a vehicle Wi-Fi diagnostic access point, assessing whether the passphrase protecting the vehicle diagnostic interface can be recovered offline. hashcat is the standard cracking step following any successful WPA handshake capture and completes the airodump-ng workflow already described in this section.

Bluetooth/BLE reconnaissance and experimentation

Bluetooth is used in vehicles for hands-free systems, smartphone pairing, and increasingly for digital keys (Bluetooth Low Energy). Unlike Wi-Fi, Bluetooth requires specialized hardware to sniff and inject, due to frequency hopping and protocol complexity.

We will review the key tools here:

- **Ubertooth One** [39] is an open-source 2.4 GHz wireless development platform that serves as a Bluetooth sniffer. Ubertooth can monitor Bluetooth Classic and BLE communications, capturing packets, and even sniffing certain connections by following the frequency hops. It is widely used by researchers to analyze Bluetooth traffic and identify pairing vulnerabilities. In discovery, a tester uses Ubertooth to

passively intercept the pairing process between a phone and the car's head unit, possibly extracting device addresses or even cracking Just Works pairing (with tools like Crackle for legacy Bluetooth). The Ubertooth toolkit supports identifying advertisement channels, intercepting BLE connection requests, etc., making it invaluable for exploring the Bluetooth attack surface.

- **BtleJack** [40] is a BLE-specific sniffing and attack tool that runs on Ubertooth One hardware and implements BLE connection following, passive packet capture, and active connection hijacking. BtleJack allows testers to hijack BLE connections by impersonating a BLE master or jamming and taking over a session. This can be used in exploitation: for example, to inject malicious commands into a BLE-based digital key communication between a phone app and a vehicle (if the protocol lacks proper encryption or authentication). BtleJack is an actively maintained tool focusing on BLE 4.x and 5.x security.
- **BlueZ** [41] is the official Linux Bluetooth protocol stack and provides a set of command-line utilities including hcitool, btmgmt, and gatttool that allow a tester to interact with Bluetooth and BLE devices directly from a Linux host. A security tester can use these tools to scan for nearby Bluetooth devices, query supported services and profiles, and enumerate BLE GATT characteristics on a vehicle interface, covering discovery of telemetry endpoints, digital key services, and passive entry BLE advertisements without requiring additional software.
- **Bettercap** [42] is a network attack and monitoring framework that covers Ethernet, Wi-Fi, and Bluetooth Low Energy from a single interactive environment. A security tester can use Bettercap's BLE modules to enumerate GATT services and characteristics on a vehicle BLE interface, read and write characteristic values, and attempt exploitation of insecure pairing implementations, covering both the reconnaissance and active testing phases of a BLE assessment in a single tool.
- **Crackle** [43]: Once an adversary has captured a pairing sequence using a tool like Ubertooth, they must still contend with the encryption protecting the link. Crackle is a specialized utility designed to crack **Bluetooth Low Energy** (**BLE**) long-term keys by exploiting weaknesses in the Legacy Pairing models, specifically "Just Works" and "Passkey" entry. By deriving the encryption keys from captured traces, a tester can decrypt subsequent traffic, such as "unlock" or "start engine" commands, rendering the protocol's confidentiality moot. It stands to reason that any assessment of BLE-based digital keys is incomplete without verifying the resilience of the association model against such automated cracking attempts.
- **InternalBlue** [44]: For researchers operating at a deeper architectural layer, InternalBlue provides a framework for firmware-level debugging of Bluetooth

controllers. Unlike host-side tools that interact with the Bluetooth stack via standard APIs, InternalBlue allows for the inspection and manipulation of the controller's internal state. This is particularly valuable when hunting for lower-level vulnerabilities within Bluetooth chip's firmware itself, which can bypass OS-level security controls. While more specialized than standard sniffer kits, it represents the "white-box" extreme of Bluetooth offensive testing.

Cellular lab networks and rogue network testing

Connected vehicles typically include a cellular modem (LTE or 5G) for telematics to support services like remote unlock and OTA updates. Directly attacking cellular communications requires advanced tools, often **software defined radios** (**SDRs**) combined with open-source network stacks, since one cannot simply "sniff" encrypted cellular traffic in the air without a base station emulator. Penetration testing the cellular interface focuses on two aspects:

- The vehicle's ability to authenticate the cellular network (protect against fake base stations)
- Any services exposed via cellular (vehicle IP address accessible services)

In this section, we will survey the commonly used tools to test those aspects:

- **Software Defined Radios (SDRs)**: Hardware like the **USRP (Universal Software Radio Peripheral)** [45] or the more affordable **HackRF One** [46] are essential for cellular testing. An SDR can transmit and receive arbitrary radio signals. In the vehicle context, a tester can use an SDR to set up a rogue base station for discovery and exploitation. For example, using a tool like srsRAN, the tester can deploy a fake 4G LTE cell tower which the vehicle may connect to if it has no authentication of network identity. SDRs also allow intercepting broadcast messages or performing denial of service by jamming. HackRF One (operating 1 MHz to 6 GHz) is widely used for testing modern wireless technologies. Hackers typically use SDR tools like HackRF or even RTL-SDR to intercept and manipulate vehicle wireless communications.
- **srsRAN** (formerly srsLTE) [47] is an open-source 4G/5G radio access network software suite. srsRAN allows a tester with an SDR to emulate an LTE eNodeB (base station) and core network. In a controlled environment (e.g., shielded RF enclosure), a penetration tester can configure srsRAN to act as a cellular network and have the vehicle connect to it. This can test whether the telematics unit properly validates cellular network credentials. Research has shown that vehicles prioritizing connectivity mighcant connect to a strong rogue LTE signal, enabling potential MitM attacks despite LTE's mutual authentication (for instance, by exploiting fallback to 2G or using a jamming and downgrade attack). With srsRAN, the tester can also observe all traffic from the vehicle on the rogue network in plaintext (discovery of what the car transmits to the

cloud) and attempt exploits like manipulating DNS responses or delivering malicious OTA updates via a spoofed server.

- **OpenLTE/Open5GS** [48, 49] are alternative open projects to simulate cellular networks or core network elements. Tools like OpenLTE can passively monitor certain LTE broadcast channels to glean information about the network (e.g., cell IDs, neighbor lists), which can be part of reconnaissance to plan an IMSI-catcher attack. In practice, this tool serves as the initial *discovery* layer that informs the later *exploitation* phases of cellular network testing.
- **OpenBTS** [50] is an open-source software-defined GSM base transceiver station that implements the GSM Um air interface protocol stack in software on commodity SDR hardware. A security tester deploys OpenBTS in a RF-shielded laboratory environment to create a rogue GSM cell and assess whether a vehicle telematics unit or embedded cellular modem registers to it without validating the network operator identity, enabling a MitM position on cellular vehicle communications. OpenBTS complements srsRAN by covering the GSM/2G layer relevant to older telematics unit fallback behavior.
- **IMSI Catcher Detectors / Stingray simulators** [51]: These aren't exactly "penetration testing" tools but rather used to test the resilience of a car against fake cell towers. For example, using a lower-end SDR and Python scripts, a tester might send out strong 2G signals (since many modems still have 2G/GSM fallback) to see if the vehicle will camp on them, then use a tool to capture the IMSI (identifier) of the vehicle's SIM which is a privacy or tracking concern. The **Osmocom suite** [52] (OpenBSC, etc.) could be used to set up such a GSM base station.
- **Wireshark** (with 5G/LTE dissectors) [53]: If the tester can capture cellular traffic (e.g., using srsRAN or a base station emulator as mentioned), Wireshark can decode higher-layer protocols like NGAP, NAS, or diagnostic protocols (e.g., if the telematics module has an open diagnostic port reachable via the cellular IP).

RFID/NFC analysis tools

Vehicle access systems that rely on **Radio-Frequency Identification** (**RFID**) and **Near-Field Communication** (**NFC**) technologies, including immobilizers, passive entry systems, and NFC-based digital keys, require dedicated hardware tools capable of reading, emulating, and replaying transponder signals to assess whether the underlying authentication mechanisms are resistant to cloning and relay attacks:

- **Proxmark3** [54] is an open-source RFID and NFC research platform that supports reading, writing, sniffing, emulating, and cloning a wide range of transponder types across low-frequency (125 kHz) and high-frequency (13.56 MHz) bands. A tester can use

it to capture and replay vehicle immobilizer transponder signals, clone passive entry credentials, and analyze the challenge-response sequences of NFC-based digital key implementations to determine whether the authentication scheme is vulnerable to relay or replay attacks.

- **Chameleon Mini** [55] is a compact open-source hardware tool for emulating and sniffing NFC and RFID cards at 13.56 MHz. It can be used to emulate a cloned vehicle digital key credential in proximity to a passive entry reader, testing whether the system enforces cryptographic challenge-response authentication or accepts a statically replayed credential, and to log raw card-to-reader exchanges for offline protocol analysis.

GNSS/GPS simulation and test signal generation

GNSS security testing in automotive systems focuses on two primary vectors: signal simulation to evaluate receiver behavior under controlled conditions and spoofing to assess whether fused localization pipelines and dependent safety features reject implausible position, velocity, or time inputs.

- **GPS-SDR-SIM** [56] generates GPS baseband signal data streams that can be converted to RF using SDR platforms such as HackRF One or USRP, allowing a tester to produce synthetic GPS signals without access to a commercial GNSS simulator. A security tester can use it in a shielded environment to inject controlled position and time trajectories into a vehicle GNSS receiver and observe whether the navigation system, security features, or dependent services accept implausible location changes without sanity checking.
- **GNSS-SDR** [57] is an open-source software-defined GNSS receiver framework that processes raw IQ samples from an SDR front end to produce position, velocity, and time solutions, supporting GPS, Galileo, GLONASS, and BeiDou signal processing. A security tester may use GNSS-SDR to decode and analyze the signal structure of GNSS transmissions received near a target vehicle, providing insight into the signal parameters and timing characteristics needed to construct a credible spoofing scenario.

V2X prototyping and message stack tooling

V2X offensive work is often constrained by specialized hardware and regulatory considerations, but protocol-stack tooling still matters for testing and analysis.

- **Vanetza** [58] is an open-source C++ implementation of the ETSI C-ITS protocol suite, covering the Facilities layer messages including CAM and DENM. The Cohda Wireless SDK accompanies the MKx hardware family, a series of V2X on-board and roadside

units that serve as the radio front end in test bench configurations where Vanetza provides the protocol stack.

- **scapy-radio** [59]: While specialized V2X hardware like the Cohda Wireless MK5 is the industry standard for testing, it is often cost-prohibitive for early-stage research. scapy-radio acts as a crucial software bridge, extending the Scapy packet-crafting framework to support the encoding of 802.11p frames required for DSRC (Dedicated Short-Range Communications). When paired with a general-purpose SDR like HackRF, scapy-radio enables the injection of Basic Safety Messages (BSMs) or the testing of message-fusion robustness without the need for dedicated V2X development kits. This allows engineers to transition from standard "bus hacking" to "environmental spoofing" using their existing RF tool arsenal.

AI-augmented tools

Wireless is one of the hardest layers to "AI-ify" in a way that survives real test conditions, because RF environments are noisy and ground truth is expensive. That said, AI is already practical in one area: signal detection and classification. DeepSig describes OmniSIG as AI-powered detection and classification for real-time IQ data received by supported radio devices, with deployments on Linux across ARM/x86/GPU configurations [60].

Where LLMs are often discussed (protocol fuzz generation for stateful wireless stacks), it is common to find research prototypes and lab-only workflows. The following table summarizes representative wireless assessment tools by activity area, including passive RF discovery, Wi-Fi and Bluetooth testing, SDR-based lab work, cellular network emulation, RFID/NFC analysis, GNSS validation, V2X testing, and AI-assisted RF classification:

Tool	Purpose	Usage Phase	Target	Reference
Aircrack-ng	Wi-Fi capture, injection, assessment tooling suite	Discovery/ Exploitation	Vehicle hotspots; Wi-Fi update channels	Aircrack-ng docs.[34]
Kismet	Passive RF/ wireless discovery across multiple protocols	Discovery	Wi-Fi/ Bluetooth RF environment mapping around vehicles	Kismet site/ docs. [35]

Tool	Purpose	Usage Phase	Target	Reference
Flipper Zero	Portable multi-tool for sub-GHz/ RFID reconnaissance	Discovery	Access-control and radio-protocol field recon	Flipper docs; product description. [37]
WiFi Pineapple	Dedicated rogue-AP and wireless audit platform	Discovery/ Exploitation	Client-side Wi-Fi trust relationships	Hak5 PineAP docs.[36]
Ubertooth	BLE sniffing and Bluetooth experimentation hardware	Discovery	Bluetooth/ BLE pairing and traffic analysis	Ubertooth repo; product status note. [39]
BtleJack	BLE sniff/jam/ hijack workflow using micro:bit	Exploitation/ Validation	BLE protocol/ session manipulation	BtleJack repo/ firmware. [40]
srsRAN Project + Open5GS	Controlled LTE/ 5G lab networks (RAN + core) for testing	Discovery/ Validation	Telematics modem behavior in controlled networks	srsRAN docs; Open5GS docs [47, 49]
HackRF / USRP	SDR platforms for controlled RF experiments	Discovery/ Validation	LTE/5G, ISM-band devices, V2X-adjacent RF labs	HackRF page; UHD/ USRP manual [46, 45]

Tool	Purpose	Usage Phase	Target	Reference
OmniSIG	AI-based RF signal detection and classification from IQ data	Discovery	RF environment classification and emitter identification	DeepSig OmniSIG page. [60]
Proxmark3 (RFIDResearchGroup fork)	RFID/NFC interaction, analysis, automation	Discovery/ Validation	RFID/NFC surfaces in access control workflows	RRG proxmark3 repo; archived official repo noted. [54]
GPS-SDR-SIM	Generate GPS baseband for SDR-based RF conversion	Validation	GNSS receiver and fused localization testing in controlled setups	GPS-SDR-SIM repo [56]
Vanetza/ Cohda SDK	V2X protocol stack and device SDK ecosystem	Discovery/ Validation	ETSI ITS-G5 / C-ITS message stack testing	Vanetza repo. [58]

Table 3.2: Wireless networking security tools

Note

Best practice:

Treat wireless testing as a measurement problem first and a hacking problem second. If you cannot reproduce RF conditions (shielding, distance, interference), "findings" tend not to survive peer review.

In-vehicle network tools

Within the car, the in-vehicle network represents a large attack surface as we will see in *Chapter 6*. It is the medium by which ECUs communicate among each other and with various sensors and actuators. A connected vehicle has a mixture of classic communication protocols such as the CAN bus (Controller Area Network) and LIN for message-based communication and Automotive Ethernet for service-based communication to support high-speed data and **Advanced Driver Assistance Systems** (**ADAS**) sensors. Penetration testers often interface with these networks via the OBD-II port or direct wiring. This section covers two categories of tooling: CAN, CAN FD, and SocketCAN-based tools for testing classic bus protocols, and Automotive Ethernet tools for testing service-oriented and IP-based communication. Both categories include hardware interfaces, software capture and analysis tools, and protocol-specific exploitation utilities.

CAN/CAN FD and SocketCAN-based workflows

Testing CAN and CAN FD networks on Linux-based platforms relies on the SocketCAN framework, which exposes CAN interfaces as standard network devices and is supported by the majority of open-source automotive security tools. The tooling in this category is organized into hardware interfaces, software tools, diagnostic utilities, and LIN-specific tools.

CAN/LIN Interfaces (**Hardware**): Physical access to the CAN or LIN bus requires a hardware interface that connects a computer to the vehicle via OBD-II or direct harness connection. The following adapters are commonly used for this purpose:

- **CANtact**, which is an open-source USB-to-CAN adapter known as "The Open Source Car Tool." It shows up as a SocketCAN device on Linux. CANtact enables monitoring and sending CAN frames using open software. It's actively maintained and inexpensive.
- **USB2CAN** / **CANable** / **PCAN-USB** are similar adapters that provide one or two CAN channels. For instance, USB2CAN is a cheap interface that registers as a network device on Linux. These devices are discovery tools (sniffing bus traffic) and exploitation tools (sending crafted frames).
- **LIN interface modules**: E.g., the LinUSBConverter (LIN-to-USB with SLCAN support) for LIN bus analysis.

With a hardware interface in place, the following software tools are used to capture, decode, and manipulate CAN traffic.

- **can-utils** [61] is a suite of command-line tools in Linux SocketCAN. Key utilities include candump (log all frames for analysis), cansniffer (display changing signal bytes in real-time), cansend (send a frame), and canplayer (replay a logged trace). These are

fundamental discovery tools for CAN. For exploitation, cansend can manually inject packets to test ECU responses (e.g., sending a CAN message to unlock doors). These tools are low-level but very flexible and are maintained as part of Linux.

- **cantools** [62] is a Python library and command-line tool for decoding and encoding CAN messages using DBC or KCD database files, supporting signal extraction, message generation, and log file parsing. A security tester can use cantools to parse a vehicle DBC file and decode candump log files into human-readable signal values, enabling rapid interpretation of raw CAN frame captures without manual bit-field calculations, and to generate correctly encoded frames for injection tests.
- **SavvyCAN** [63] is a cross-platform GUI tool for CAN bus reverse engineering and capture. SavvyCAN can connect to any SocketCAN or native interface and provides visualization of CAN traffic, including graphing signal values, filtering by ID, and interpreting data with loaded DBC files (CAN database files). It supports sending frames and even basic scripting. In discovery, SavvyCAN greatly speeds up reverse engineering by allowing one to see patterns (for example, plotting a signal to find which CAN ID corresponds to vehicle speed). For exploitation, a tester can use SavvyCAN's transmit functionality or "replay attack" feature to resend a sequence of captured frames (e.g., to retrigger an action like honking horn).
- **Kayak** [64] is another open-source CAN GUI (Java-based) that allows CAN logging, sending, and UDS (Unified Diagnostic Services) scanning. Useful for discovery of diagnostic ECUs and sending diagnostic requests.
- **BUSMASTER** [65] is an open-source application for simulation, analysis, and testing of CAN, LIN, and FlexRay bus systems developed under the RBEI-ETAS initiative. A security tester may use BUSMASTER to log and replay CAN traffic, simulate ECU nodes on the bus, and inject specific frames while monitoring bus responses, providing a Windows-native graphical alternative to the Linux can-utils suite for environments where SocketCAN is not available.
- **CANalyzer** [66] is a commercial tool by Vector that is widely used in the automotive industry. It provides professional analysis of CAN/LIN with rich features such as timing analysis and node simulation. Testers in the industry often use Vector tools to simulate ECUs or craft complex communication scenarios. For example, Vector CANoe adds capabilities to simulate entire vehicle networks and perform fuzzing with CAPL scripts which can be powerful in designing CAN bus exploits.
- **CANToolz** [67] is an open-source Python framework specifically for car network pentesting. CANToolz (aka YACHT) allows chaining of modules in a "pipeline" to perform tasks like CAN MitM, fuzzing, replay, and even emulating ECUs. It includes

modules for scanning UDS services (finding diagnostic functions), and injecting errors. For example, using CANToolz, a tester can set up a proxy that modifies certain CAN messages on the fly. It is actively maintained on GitHub and integrates with SocketCAN hardware. While more complex than can-utils, CANToolz is very powerful for systematic CAN fuzz testing.

- **Caring Caribou** [68] is a purpose-built automotive security tool for CAN bus exploration that includes dedicated modules for UDS service discovery, seed-key analysis, and UDS-level fuzzing. A security tester can run Caring Caribou against an ECU over a SocketCAN interface to enumerate supported UDS services and session types, identify which diagnostic functions respond without authentication, and generate malformed UDS requests to test input handling in ECU diagnostic routines.
- **Scapy** [69] provides automotive-specific protocol layers covering CAN, ISO-TP, UDS, DoIP, and SOME/IP through its contrib modules, giving a tester a scriptable environment for constructing and parsing automotive protocol traffic at any layer. A security tester can use Scapy to build custom scanners, replay captured diagnostic sessions, and craft malformed protocol sequences that purpose-built tools do not support, making it the standard choice for protocol experimentation and custom tooling in vehicle network test benches.
- **python-can** [70] and **python-isoTp** [71] libraries enable writing custom scripts to perform UDS discovery. For instance, a tester might write a Python script to iterate through possible UDS service IDs and subfunctions, using python-can send requests and log responses.
- **UDSim** [72] is a simple ECU simulator for UDS that runs on Arduino or Linux. Testers use UDSim to practice or to simulate a target ECU when crafting an exploit.
- **Diagnostic scan tools (OEM)**: these are dealer-grade tools (like Snap-On Solus, Bosch scanners) or even the manufacturer's own (Tech2, VCDS, etc.). These are not intended for hacking, but a pentester might use them in discovery to identify CAN IDs and functions by observing what the official OEM tool sends. The risk is these tools themselves can be targets as researchers have demonstrated vulnerabilities in dealership diagnostic equipment (Snap-On, Bosch), which could propagate malware via OBD-II. In a penetration test, one might include verifying that a car's OBD-II cannot be exploited via such tools as a defense-in-depth check.
- **LIN Bus tools**: Often the CAN gateway will forward LIN data as CAN frames, but direct LIN testing might be needed for certain components (e.g., seat module on LIN). LIN sniffers like the mentioned LinUSBConverter or even a logic analyzer with LIN decoder can be used. One open approach is using an Arduino with a LIN transceiver and the

linbus Arduino library to inject or monitor LIN frames. Tools like TSMaster (a commercial CAN/LIN environment) support LIN analysis as well. Generally, once on LIN, testers look for the ability to send invalid commands to cause faulty behavior or escalate to CAN via a gateway.

Automotive Ethernet access and mixed-protocol tools

The practical blocker for Automotive Ethernet testing is often physical layer adaptation. Technica Engineering describes its 100BASE-T1 media converter as a physical layer conversion between Automotive Ethernet and standard RJ-45 NICs [73]. Intrepid describes RAD-Moon2 as a media converter bridging standard Ethernet and Automotive Ethernet (100/1000BASE-T1) [74].

Once connected, mainstream network tooling becomes relevant again, such as the previously mentioned tcpdump, sslyze, mitmproxy, and the tools mentioned in this section:

- **Nmap** [75] is an open-source network scanner that performs host discovery, port scanning, service version detection, and OS fingerprinting using a flexible probe and timing engine. A security tester can run Nmap against an Automotive Ethernet segment to enumerate all active nodes, identify open TCP and UDP ports on IVI units, gateway ECUs, and telematics control units, and fingerprint the service versions exposed on each host, producing a complete network map before targeted exploitation.
- **Wireshark** [53, 76] is a network protocol analyzer that captures packets from a live interface or a PCAP file and dissects them across a wide range of protocols, including SOME/IP, DoIP, and standard TCP/IP. A security tester might use Wireshark on an Automotive Ethernet test bench to inspect SOME/IP service discovery traffic, decode DoIP diagnostic sessions between an external tester and a gateway ECU, and identify cleartext payloads or missing integrity protection on inter-ECU communications that would be exploitable by a node with access to the segment.
- **Scapy** [69] provides automotive-specific protocol layers covering CAN, ISO-TP, UDS, DoIP, and SOME/IP through its contrib modules as mentioned above, giving a tester a scriptable environment for constructing and parsing automotive protocol traffic at any layer. Thus, a security tester could use Scapy also on an Automotive Ethernet bench to craft custom DoIP diagnostic requests, inject malformed SOME/IP service discovery messages, and build replay sequences targeting gateway ECUs and domain controllers, covering protocol interactions that purpose-built tools do not support.
- **Metasploit** [77] is a penetration testing framework that provides exploit modules, auxiliary scanners, and post-exploitation capabilities across a broad range of network services and vulnerabilities. A security tester would generally use Metasploit on an Automotive Ethernet segment to run auxiliary scanners against exposed services on IVI

units and telematics gateways, validate whether known CVEs in embedded Linux components are exploitable, and simulate post-compromise lateral movement between network domains to assess whether in-vehicle network segmentation controls limit an attacker's reach after an initial foothold.

- **Ettercap** [78] is a network interception tool that combines ARP poisoning, man-in-the-middle traffic capture, and protocol dissectors for analysis of local network traffic. A security tester uses Ettercap on an Automotive Ethernet test bench to intercept communications between an IVI unit and a gateway ECU by poisoning the ARP caches of both nodes, injecting crafted packets to test whether transport-layer integrity protection such as TLS or MACsec is enforced on the segment.
- **Yersinia** [79] is a network protocol attack tool that implements active exploitation modules for layer 2 protocols including STP, CDP, DHCP, HSRP, and DTP. A security tester runs Yersinia on an Automotive Ethernet segment to send crafted STP topology change notifications or DHCP exhaustion attacks, testing whether in-vehicle Ethernet switches enforce port security controls that prevent unauthorized protocol manipulation from a compromised network node.
- **CANalyzer** [66] can also be leveraged for automotive Ethernet penetration testing when paired with the VN5000 to analyze protocols such as SOME/IP, SOME/IP-SD, IP, TCP, UDP, AVB, RTP, PTP and many more.

AI-augmented tools

AI augmentation is currently most credible in in-vehicle networks when the target protocol is structured and stateful (SOME/IP, DoIP, UDS over IP). In contrast, raw CAN reverse engineering tends to remain human-intensive due to the need for observing the vehicle's physical behavior after manipulating a CAN signal. CyTAL describes ProtoCrawler as providing *intelligent fuzzing* for automotive protocols including SOME/IP, DoIP, UDS, and related stacks, framing it as a test tool to identify protocol implementation weaknesses across modern vehicle architectures [80]. This is exactly the point where AI-like capability can change workflow: it can accelerate generation of protocol-valid messages and state transitions, which reduces the time from "*I can speak the protocol*" to "*I can systematically stress the implementation.*" The following table summarizes representative tools for in-vehicle network assessment, covering CAN and CAN FD capture, replay, fuzzing, diagnostic enumeration, automotive Ethernet access, packet inspection, IP service scanning, and protocol-aware validation:

Tool	Purpose	Usage Phase	Target	Reference
can-utils	SocketCAN userspace tools for capture, replay, injection	Discovery/ Exploitation	CAN/CAN FD internal networks	linux-can can-utils README. [61]
SavvyCAN	GUI analysis, visualization, reverse engineering, replay	Discovery/ Exploitation	CAN/CAN FD signal inference and logging	SavvyCAN repo [63]
CANToolz	Modular CAN pipeline for MitM, fuzzing, scanning	Discovery/ Exploitation	CAN security testing workflows	CANToolz README; Black Hat deck (historical context).[67]
Caring Caribou	Automotive exploration + UDS discovery and fuzz support	Discovery/ Validation	UDS over CAN, ECU discovery workflows	Caring Caribou repo/docs.[68]
Scapy automotive layers	Custom packet crafting for ISO-TP/UDS/ DoIP/SOME-IP	Discovery/ Validation	Automotive diagnostics transport and service layers	Scapy automotive docs; SOME/IP contrib API. [69]
Wireshark	Inspection/ analysis with SOME/IP and DoIP support	Discovery	Automotive Ethernet, SOME/IP, DoIP sessions	SOME/IP and DoIP filter refs. [53, 76]

Tool	Purpose	Usage Phase	Target	Reference
Technica MediaConverter / Intrepid RAD-Moon2	Physical-layer access to 100/1000BASE-T1 networks	Discovery enabler	Automotive Ethernet physical access	Technica product page; RAD-Moon2 page. [73, 74]
Nmap/ Metasploit	Host/service enumeration and exploit framework	Discovery/ Exploitation	Services exposed on in-vehicle IP networks	Nmap docs; Metasploit overview. [75, 77]
ProtoCrawler	Intelligent fuzzing for automotive protocols	Validation	SOME/IP, DoIP, UDS protocol stacks	CyTAL automotive protocol fuzzing writeup; ProtoCrawler page. [80]

Table 3.3: In-vehicle networking tools

Now that we understand the buses and internal protocol stacks, we will turn our attention to the compute endpoints starting with the high-performance ECUs, where offensive tooling starts to resemble embedded Linux and enterprise endpoint exploitation.

High-performance and MCU-based ECU tools

High-performance ECUs include ADAS systems, infotainment units, and central computers running Linux, Android, QNX, or a combination of those on one or more powerful SoCs. As we will see in *Chapter 7*, in many cases, these platforms also host hypervisors and feature complex secure boot chains. The security consequence is immediate: when a vehicle consolidates functions into central computers, a single compromise can cascade across domains that were once physically separated.

On the other hand, MCU-based ECUs are still dominant in functions such as powertrain, braking subsystems, body controllers, and many peripheral controllers. As we will see in *Chapter 8*, these environments commonly involve AUTOSAR Classic, RTOS/bare-metal firmware, and architectures such as ARM Cortex-M, TriCore, PowerPC, and other automotive MCU families. The tools used for pentesting both types of ECUs are largely overlapped;

therefore, we will present both together while pointing out when a tool is applicable to one type and not the other.

Hardware interface discovery and debug access

Before software or firmware analysis can begin, testers often need to determine which physical interfaces are exposed on the ECU and whether they provide access to debug, programming, or boundary-scan functionality. This activity includes identifying test points, connector pins, JTAG or SWD interfaces, UART consoles, boot-mode pins, and other hardware entry points that may expose privileged access to the device.

The tools below are commonly used to discover, validate, or interact with these interfaces during embedded ECU assessments:

- **JTAGulator** [81] is an open-source hardware tool to assist in identifying on-chip debug (OCD) interfaces from test points and connectors.
- **OpenOCD** [82] aims to provide debugging, in-system programming, and boundary-scan testing for embedded target devices, and it remains a common bridge between discovered debug ports and software analysis.
- **SEGGER's J-Link** [83] is a commonly used debug probe ecosystem supporting multi-platform tools.
- **Bus Pirate** [84] is a versatile tool for working with serial protocols like I2C, SPI, UART, and more, enabling interaction with chips without custom code. This matters in MCU ECU work because many security-critical secrets and boot configuration artifacts are stored on peripherals that leverage bus protocols for interfacing to the MCU.
- **i2c-tools** [85] (i2cdetect / i2cdump / i2cset) are Linux command-line utilities for probing and interacting with I2C bus devices: i2cdetect scans a bus for responding device addresses, i2cdump reads all register values from a specified device, and i2cset writes a value to a specific register. A security tester may use these tools on an embedded Linux ECU or IVI unit to enumerate connected sensor ICs and peripheral controllers, read calibration registers, and write test values to confirm whether the I2C bus lacks access control that would prevent a local attacker from manipulating connected sensors used in ADAS subsystems.
- **minicom** [86] is a terminal emulation program for serial port communication that provides a text-based console interface to embedded systems over RS-232 or USB-serial adapters. A security tester uses minicom to access the bootloader or Linux serial console of a vehicle ECU or IVI unit, testing whether the serial interface is left open in production hardware, whether it requires authentication, and whether it provides root shell access or bootloader commands that bypass firmware signature verification.

Fault injection and side-channel research tooling (validation)

When conventional debug access is locked down and firmware protections are enabled, testers may move to physical validation techniques that examine how the device behaves under measurement or induced faults. Side-channel tools observe leakage such as power or electromagnetic emissions to infer secrets or implementation behavior, while fault-injection tools deliberately disturb execution to test whether security checks, secure boot logic, or cryptographic operations can be bypassed.

The following tools are commonly used to validate whether hardware-backed protections remain effective under realistic physical attack conditions:

- **ChipWhisperer** [87] is an open-source toolchain focused on power analysis and voltage/clock glitching to induce unintended behavior such as bypassing checks. It is commonly used in both MCU and HPC security assessments when debug and firmware paths are restricted.
- **ChipShouter** [88] is an electromagnetic fault injection platform that delivers high-voltage pulses through a handheld probe to induce transient faults in a target device without requiring direct electrical contact. A pentester can position the probe over an automotive MCU and time the pulse to coincide with a security-critical instruction, such as a secure boot verification, to force the device into an unintended execution state.
- **Riscure Inspector** [89] is a commercial side-channel analysis and fault injection platform that combines hardware acquisition probes with software for power trace capture and statistical analysis techniques such as **Differential Power Analysis** (**DPA**) and **Correlation Power Analysis** (**CPA**). A security tester can use it to recover key material from automotive ECU cryptographic implementations or perform precision fault injection against secure boot routines.

Firmware acquisition and reverse engineering

After hardware interfaces or update artifacts have been identified, testers often need to acquire firmware and translate opaque binaries into analyzable software components. This activity may involve reading external flash, unpacking OTA images, identifying file systems, disassembling bootloaders, emulating firmware, or instrumenting running processes on Linux, Android, or QNX-based platforms.

The following tools support the progression from firmware acquisition to static analysis, emulation, dynamic instrumentation, and deeper vulnerability research:

- **flashrom** [90] is a utility for detecting, reading, writing, verifying, and erasing flash chips, including support for numerous external programming devices. This becomes

relevant whenever ECUs use external SPI flash or when update images and boot artifacts can be acquired via off-board reads.

- **Binwalk** [91] is a firmware analysis tool that scans binary images for embedded file signatures, compressed archives, and known file system formats, and extracts identified components automatically. A security tester could use it to unpack an OTA update image or ECU firmware blob, recovering the underlying file system, configuration files, and executable components for subsequent static analysis with Ghidra or radare2.
- **QEMU** [92] is a generic open-source machine emulator and virtualizer that supports a wide range of processor architectures including ARM, MIPS, and x86. QEMU can be used to emulate vehicle ECU or IVI unit firmware extracted from the target hardware, enabling dynamic analysis, fuzzing, and vulnerability research without requiring physical access to the device. QEMU-based emulation is particularly valuable for early-stage assessment when hardware is not yet available.
- **Ghidra** [93] is an open-source reverse engineering framework developed by the NSA that supports static disassembly and decompilation of binaries compiled for a wide range of processor architectures including ARM, MIPS, and TriCore. A security tester can use it to analyze extracted ECU firmware, recover control flow and function boundaries in stripped binaries, and identify vulnerable parsing routines or disabled security checks in automotive software components.
- **IDA Pro** [94] is a commercial multi-processor disassembler and interactive debugger that supports analysis of binary executables compiled for a wide range of architectures, including x86, ARM Cortex-M, and TriCore. A security tester may use IDA Pro to analyze ECU firmware, companion application native libraries, and bootloader binaries, to identify known library functions, recover variable names, and map control flow in stripped production binaries.
- **Frida** [95] is a dynamic instrumentation toolkit that injects a scriptable agent into a running process, enabling function hooking, memory inspection, and runtime modification without recompiling the target binary, with explicit support for Android, Linux, and QNX. A security tester might use Frida to hook certificate pinning routines in a vehicle companion application, intercept cryptographic function calls on an IVI unit, or monitor inter-process communication on QNX-based HPC environments at runtime.
- **radare2** [96] is an open-source reverse engineering framework providing a command-line environment for binary analysis, disassembly, debugging, and scripting across multiple architectures. A security tester can use it to analyze automotive firmware

binaries in resource-constrained environments, script custom analysis workflows, and cross-validate findings from Ghidra on architectures where community plugins are limited.

Fuzzing and kernel/system stress testing

Once firmware, binaries, drivers, or system interfaces have been identified, testers can use fuzzing and stress testing to exercise code paths that are difficult to cover manually. The following tools are commonly used to achieve those objectives:

- **AFL++** [97] is a fork of AFL with more speed, mutations, and instrumentation options making it a common choice for coverage-guided fuzzing of userland components once harnessing is possible
- **Syzkaller** [98] is an unsupervised coverage-guided kernel fuzzer that began with Linux kernel fuzzing but extends to additional kernels making it relevant to automotive Linux kernels (and device-specific drivers) used in HPCs

AI-augmented tools

AI is most mature in high-performance ECU work where it augments the two bottlenecks that dominate cost, which are binary triage/understanding and fuzz harness creation:

- **BinaryAI** [99] is a security analysis platform that analyzes binaries with AI algorithms and provides capabilities such as parsing and software composition analysis; it also provides plugins for Ghidra and IDA-style workflows. In practice, this can accelerate "*what is in this firmware?*" and "*is this component reused across variants?*" questions that directly affect exploitability and patch scope.
- **VulHunt** [100] is a binary vulnerability analysis framework designed to bring semantic, code-level analysis to binaries, and can be integrated with LLMs for triage and hunting. This matters in automotive because HPC firmware images can be dependency-heavy and frequently incorporate third-party components. Naturally, semantic binary analysis is useful to shorten the time from firmware extraction to vulnerability discovery.
- **oss-fuzz-gen** [101] generates fuzz targets for real-world projects with various LLMs that are benchmarked via OSS-Fuzz. OSS-Fuzz documentation similarly describes LLM target generation as a way to reduce manual integration effort.
- **CI Fuzz Spark** [102] uses workflows to automatically generate fuzz tests and run them. In practice, this is one of the most defensible AI applications: generating and validating fuzz harnesses is time-consuming, and automation that produces runnable harnesses has immediate value.

The following table summarizes representative tools used across HPC and MCU-based ECU assessments. These tools support the workflow from hardware interface discovery and firmware acquisition to reverse engineering, runtime instrumentation, fuzzing, kernel stress testing, binary triage, and physical validation through side-channel or fault-injection techniques:

Tool	Purpose	Usage Phase	Target	Reference
JTAGulator	Discover JTAG/ UART-style debug interfaces	Discovery	HPC ECUs with exposed test points	JTAGulator page; slides [81]
OpenOCD	Debugging and in-system programming via JTAG/SWD	Discovery/ Exploitation enabler	Embedded SoCs and supported debug adapters	OpenOCD user guide.[82]
SEGGER J-Link	Debug probe ecosystem and tooling	Discovery/ Exploitation enabler	ARM-based ECUs and debug access	SEGGER J-Link docs.[83]
Binwalk v3	Firmware structure identification and extraction support	Discovery	ECU firmware blobs and update images	Binwalk v3 repo. [91]
Ghidra	Static analysis and decompilation for compiled code	Discovery	ECU binaries across common platforms	Ghidra repo/ releases.[93]
Frida	Runtime instrumentation and behavioral validation	Validation	Android/Linux/ QNX userland services	Frida docs.[95]

Tool	Purpose	Usage Phase	Target	Reference
AFL++	Coverage-guided fuzzing for userland components	Validation	Parsers and network-facing services	AFL++ repo/site. [97]
syzkaller	Coverage-guided kernel fuzzing	Validation	Linux kernels and drivers	syzkaller repo/ docs.[98]
VulHunt	Semantic binary vulnerability analysis; LLM-assisted triage	Discovery/ Validation	Extracted ECU binaries and libraries	Binarly VulHunt posts/site. [100]
OSS-Fuzz-gen / CI Fuzz AI Test Agent	LLM-assisted fuzz harness generation and fuzz automation	Validation	ECU software components suitable for fuzzing	OSS-Fuzz-gen repo; OSS-Fuzz LLM page; CI docs. [101, 102]
flashrom	Read/write/ verify flash chips; supports external programmers	Discovery	External SPI flash and firmware storage	flashrom manpage/ README.[90]
Bus Pirate	Multi-protocol serial interface (I2C/SPI/UART, etc.)	Discovery	Peripheral buses on ECU PCBs	Bus Pirate docs. [84]
Saleae Logic software	Protocol decoding for embedded buses (UART/I2C/SPI, etc.)	Discovery	Signal capture and bus decoding on PCBs	Saleae protocol docs. [103]

Tool	Purpose	Usage Phase	Target	Reference
ChipWhisperer	Side-channel and fault injection (power/glitching)	Validation	Protected MCUs; boot/auth bypass research	ChipWhisperer getting started. [87]
BinaryAI	AI-assisted binary understanding and composition analysis	Discovery/Triage	Extracted ECU binaries	BinaryAI docs; plugin repo. [99]

Table 3.4: HPC and MCU offensive security tools

Now that we have covered ECUs from central compute down to microcontrollers, we will turn our attention to the sensor layer, where offensive questions span both RF/physics and ML/perception robustness.

Vehicle sensor tools

The final set of tools we examine in this chapter are sensor-adjacent tooling that commonly appear in connected vehicle work. Vehicle sensor systems present a distinct attack surface in which the threat is not a network-accessible service but the physical signal environment, requiring tools capable of generating, capturing, or manipulating RF, optical, acoustic, and AI-layer inputs to assess sensor integrity and the resilience of perception pipelines. The important difference here is that "sensor tools" operate at the boundary between software-layer manipulation and physical-world effects, such as adversarial patches on road signs that cause a camera-based perception system to misclassify, meaning that both attack dimensions need to be considered when assessing findings.

As we will see in *Chapter 9*, many advanced sensors mimic the architecture of MCU-based ECUs, so for those, the same penetration testing tools from the previous section apply. This section covers tools for capturing and decoding RF signals from vehicle sensor systems, replaying and injecting sub-GHz transmissions, and applying AI-augmented techniques to sensor-layer analysis.

RF signal capture, protocol analysis, and injection

Vehicle sensor systems, including tire pressure monitors, keyless entry receivers, and remote start modules, rely on RF protocols in the sub-GHz bands, typically 315 MHz and 433 MHz in automotive applications, that are often proprietary and sparsely documented. These signals

are frequently transmitted without cryptographic protection or with weak rolling-code implementations, making replay and injection attacks practical.

The tools in this section span both the discovery and exploitation phases of RF testing: passive capture and protocol decoding establish what is being transmitted and how signals are structured, while transmit-capable platforms allow a security tester to replay or inject crafted frames to test receiver validation. Software-defined radio platforms have made both roles achievable without specialized lab equipment.

We will now discuss the key tools for signal capture and protocol decoding:

- **rtl_433** [104] decodes radio transmissions from devices on ISM bands (and other frequencies). This shows up in vehicle-adjacent work for sensors and telemetry devices that transmit in common bands (for example, TPMS-like systems in certain contexts), where visibility into "what is being transmitted" is the first step before any security hypothesis can be tested.
- **RTL-SDR V3** [105] is a low-cost USB software-defined radio receiver covering approximately 500 kHz to 1.75 GHz that serves as the RF front end for software-based signal decoding. When paired with rtl_433, a security tester can passively capture TPMS sensor broadcasts from a target vehicle and automatically decode sensor identifiers, tire pressure, temperature, and battery status without any vehicle network access, enabling both sensor spoofing analysis and vehicle tracking research.
- **Universal Radio Hacker (URH)** [106] is a software tool for recording, analyzing, demodulating, and decoding radio frequency signals captured by an SDR receiver. In the sensor context, a pentester can use URH to analyze TPMS transmissions and RKE signals captured from a target vehicle, recovering the modulation parameters, packet structure, and bit encoding of proprietary ISM-band protocols without requiring prior knowledge of the specification. URH complements rtl_433, by providing an interactive protocol analysis environment for signals that rtl_433 does not decode automatically.
- **GNU Radio** [107] is an open-source signal processing framework that implements radio protocols in software using a block-based data flow model. GNU Radio serves as the runtime for vehicle-specific decoder modules such as gr-tpms and enables construction of custom receiver chains for ISM-band sensor protocols not covered by rtl_433 or URH. A security tester can build GNU Radio flowgraphs to process IQ sample captures from HackRF or RTL-SDR into decoded sensor payloads.
- **gr-tpms** [108] is a GNU Radio out-of-tree module that implements a receiver and decoder for TPMS RF transmissions operating in the 315 MHz and 433 MHz ISM bands. A security tester loads the gr-tpms flowgraph with an RTL-SDR V3 or HackRF One as the RF front end to capture TPMS sensor broadcasts from a target vehicle, recovering

the sensor identifier, tire pressure reading, temperature, and battery status from decoded packets without any vehicle network access.

- **HackRF One** [46] is a wideband SDR transceiver covering 1 MHz to 6 GHz that supports both receive and transmit operations, extending TPMS testing beyond passive capture. A security tester can use it with rtl_433 for TPMS signal reception and subsequently transmit crafted TPMS frames at the correct ISM-band frequency to test whether the target vehicle's receiver accepts replayed or spoofed sensor packets without freshness validation.
- **Yard Stick One** [109] is a sub-GHz SDR transceiver operating in the 300 MHz to 928 MHz ISM band. The Yard Stick One enables a security tester to not only receive TPMS and RKE transmissions but also to transmit replay frames at the correct sub-GHz frequency, confirming whether the target receiver enforces any form of replay protection

AI-augmented tools

Sensor-layer AI augmentation is fundamentally different from backend or ECU AI augmentation. Here, AI frequently exists in the target system (perception and fusion) as much as in the tester's toolchain. The most practical offensive tooling is therefore adversarial ML evaluation utilities:

- **ART (Adversarial Robustness Toolbox)** [110] is a Python library for ML security, hosted by the LF AI & Data foundation, and explicitly lists support for evaluating threats such as evasion, poisoning, extraction, and inference. A security tester can apply ART to automotive perception models, such as object detection or lane classification networks, to generate adversarial inputs that test whether the model can be reliably deceived under conditions representative of real-world sensor data
- **Counterfit** [111] is a generic automation layer for assessing the security of machine learning systems and integrating multiple adversarial frameworks. These tools are practically usable when the assessment scope includes ML models in perception pipelines, simulated ADAS stacks, or model-serving endpoints used in development and validation environments.
- **Foolbox** [112] is a Python library for constructing adversarial examples against machine learning models using a consistent API that abstracts over different attack methods and model frameworks including PyTorch, TensorFlow, and JAX. In an automotive context, a pentester can use Foolbox to craft adversarial inputs targeting vision-based ADAS components, such as camera-fed pedestrian detectors or traffic sign classifiers, testing whether small perturbations to input images cause misclassification

without requiring access to the model's training data. Foolbox complements ART, by providing a broader set of attack algorithms across frameworks.

- **Garak** [113] is an open-source LLM vulnerability scanner that probes language models for failure modes including prompt injection, jailbreaking, hallucination, and data extraction through a structured probe-and-detector framework. A security tester can use Garak to assess in-vehicle AI assistant implementations and cloud-connected LLM services for susceptibility to adversarial inputs that cause the model to bypass safety constraints or reveal sensitive system prompt content.

The following table summarizes representative tools for assessing vehicle sensor-adjacent attack surfaces. These tools cover RF visibility for sensor and telemetry signals, controlled SDR experimentation, and adversarial ML validation for perception, fusion, and other machine-learning components used in modern vehicle systems:

Tool	Purpose	Usage Phase	Target	Reference
HackRF / USRP	RF transmit/ receive SDR platforms for TPMS/RF labs	Discovery/ Validation	TPMS and other sensor-adjacent RF experiments	HackRF page; UHD/USRP manual [46, 45]
rtl_433	Decode ISM-band device transmissions	Discovery	ISM-band sensor/telemetry visibility	rtl_433 repo. [104]
ART	Adversarial ML evaluation toolkit (evasion, poisoning, etc.)	Validation	Perception/ fusion ML models and ML components	ART repo/ releases. [110]
Counterfeit	Automation layer for ML security testing across frameworks	Validation	ML systems and evaluation workflows	Counterfit repo; Microsoft release note. [111]

Table 3.5: Vehicle sensor tools

Note

Best practice:

Treat *sensor spoofing* and *model spoofing* as separate hypotheses. A sensor-level perturbation that is physically feasible can still be neutralized by fusion; conversely, a model weakness can be triggered without physically "spoofing" a sensor if the system has simulation, replay, or model-serving interfaces in scope.

Before wrapping up the discussion on penetration testing tools, it is noteworthy to highlight the role of AI in offensive security tooling and how it is changing the landscape on what is feasible, how deep an engagement can be, and how quickly the objectives can be completed.

How AI is changing penetration testing

AI's strongest practical impact today is concentrated where the work is dominated by text, structure, and scale: backend/API workflows, binary triage in software-heavy ECUs, and fuzz harness generation for large codebases. Burp AI is a good example at the backend layer because it is integrated into a mainstream proxy workflow and explicitly designed to accelerate understanding and reduce human error without "always-on autonomy" [14].

At the ECU software layer, VulHunt and its associated writing emphasize semantic analysis of binaries and discuss LLM integration for triage and hunting, which maps well to the reality of dependency-heavy firmware images [100]. For fuzzing, OSS-Fuzz-gen and Spark AI speak directly to the harness bottleneck: OSS-Fuzz-gen describes LLM-based fuzz target generation and benchmarking; CI Fuzz Spark's docs describe automatically generating fuzz tests [101, 102]. This leaves defenders with a concrete consequence: if an organization's vehicle software components are fuzzable in principle, AI-assisted harnessing tends to reduce the time-to-fuzz and time-to-find, which shifts offensive economics even without new exploit primitives.

AI is also changing wireless work, but more narrowly. DeepSig's OmniSIG demonstrates the "AI helps measure RF" category, such as signal detection and classification from IQ streams, where AI can reduce triage time in crowded spectrum environments [60]. However, the deeper the workflow goes into tight timing, PHY constraints, and device-specific behavior, the more AI becomes an assistive analysis layer rather than the core driver of exploitation.

Finally, in sensors and autonomy stacks, AI changes the offensive testing question itself. ART and Counterfit explicitly target ML threats such as evasion and extraction and automate adversarial evaluation workflows [110, 111]. This is a meaningful shift: it expands offensive testing from "can I impact the sensor" to "can I systematically evaluate whether the ML

perception/fusion system fails under realistic perturbations," which is closer to safety analysis than classic appsec.

Where AI remains immature is equally important. Physical extraction on MCUs, debug lock bypass, and lab-grade fault injection remain dominated by instrumentation, physical access, and device-specific skill. ChipWhisperer's own documentation emphasizes voltage/clock glitching and side-channel work which are precisely the kinds of tasks that remain stubbornly physical [87]. It stands to reason that the "AI wave" will shift the most in layers that already look like software, and the least in layers where the limiting factor is physics.

Summary

In this chapter we surveyed the offensive security tool ecosystem that supports penetration testing across the full connected vehicle stack. Rather than presenting tools in isolation, we organized them according to architectural layers and mapped them to the two operational phases that govern most real-world engagements: discovery and exploitation/validation. This perspective is important because the effectiveness of a penetration test rarely depends on a single powerful tool; instead, it emerges from the systematic orchestration of multiple tools that progressively transform an opaque system into one that is observable, measurable, and ultimately testable.

Across the vehicle backend, wireless interfaces, in-vehicle networks, ECU compute platforms, and sensor subsystems, a consistent pattern emerges. Discovery tooling focuses on visibility: enumerating exposed infrastructure, mapping protocols, extracting firmware artifacts, capturing traffic, and reconstructing workflows. Exploitation and validation tooling then builds on this visibility to test hypotheses about weaknesses by crafting inputs, manipulating protocol state, fuzzing implementations, or inducing abnormal behavior under controlled conditions. The penetration tester's workflow therefore resembles a scientific process. The tester observes, forms hypotheses, tests those hypotheses, and validates results. Tools serve as instruments that enable increasingly precise experimentation against the system.

As will become apparent in the following chapters, automotive security testing spans multiple engineering domains simultaneously. Backend testing resembles traditional cloud and API security assessments. Wireless testing introduces radio frequency constraints and physical-layer experimentation. In-vehicle network testing requires protocol reverse engineering and bus-level manipulation. ECU analysis brings embedded systems reverse engineering and firmware analysis into scope. Finally, sensor-layer work merges cybersecurity with signal processing and machine learning robustness. Few other domains require practitioners to operate comfortably across such a wide range of technologies. As a result, building a capable offensive tool arsenal is not simply about collecting utilities, but about developing the ability

to integrate them into coherent workflows that cross hardware, software, and network boundaries.

In this chapter, we also examined how artificial intelligence is beginning to reshape parts of this workflow. AI currently provides the greatest practical benefits in areas where data is structured and plentiful, such as API analysis, binary triage, and fuzz harness generation. In these contexts, AI systems act as accelerators that reduce the time required to understand complex software components and generate large volumes of test inputs. However, layers dominated by physical interfaces, such as MCU debugging, fault injection, or RF experimentation, remain constrained by instrumentation and laboratory conditions. In those environments, AI primarily assists analysis rather than replacing hands-on experimentation. Understanding where AI meaningfully augments testing and where traditional engineering techniques remain indispensable is therefore an important skill for modern penetration testers.

In the chapters that follow, we will begin applying these tools to concrete attack scenarios across the vehicle architecture. Rather than introducing new tooling, the focus will shift toward building realistic attack chains, correlating findings across layers, and demonstrating how weaknesses discovered during reconnaissance can evolve into meaningful security compromises. The toolkit presented in this chapter therefore serves as the foundation for the offensive experiments that lie ahead.

References

[1] Shodan Search Engine. [Online]. Available: `https://www.shodan.io`

[2] Censys, *"Attack Surface Management and Internet Intelligence*." [Online]. Available: `https://search.censys.io`

[3] OWASP, "*Amass Project*." [Online]. Available: `https://owasp.org/www-project-amass/`

[4] SecurityTrails, "*DNS intelligence and historical data*." [Online]. Available: `https://securitytrails.com`

[5] Robert David Graham, "*masscan*". [Online]. Available: `https://github.com/robertdavidgraham/masscan`

[6] Gitleaks, "*Gitleaks*." [Online]. Available:`https://github.com/gitleaks/gitleaks`

[7] Truffle Security, "*truffleHog*". [Online]. Available: `https://github.com/trufflesecurity/trufflehog`

[8] GitGuardian, "*GitGuardian secrets detection and remediation platform.*" [Online]. Available:https://www.gitguardian.com

[9] skylot, "*jadx: Dex to Java decompiler.*" [Online]. Available: https://github.com/skylot/jadx

[10] MobSF, "*Mobile Security Framework*". [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF

[11] topjohnwu, "*Magisk.*" [Online]. Available:https://github.com/topjohnwu/Magisk

[12] OpenSSL, "*openssl-s_client manual page*". [Online]. Available: https://www.openssl.org/docs/man3.0/man1/openssl-s_client.html

[13] Progress Telerik, "*Fiddler.*" [Online]. Available: https://www.telerik.com/fiddler

[14] PortSwigger, "*Burp AI documentation.*" [Online]. Available:https://portswigger.net/burp/documentation/desktop/burp-ai

[15] mitmproxy, "*mitmproxy documentation.*" [Online]. Available:https://docs.mitmproxy.org/stable

[16] Postman, "*Send a request with the Postman API client.*" [Online]. Available:https://learning.postman.com/docs/sending-requests/create-requests/request-basics

[17] OWASP, "*Zed Attack Proxy (ZAP).*" [Online]. Available: https://www.zaproxy.org

[18] MQTT Explorer, "*MQTT Explorer*". [Online]. Available: https://mqtt-explorer.com

[19] ticarpi, "*jwt_tool.*" [Online]. Available: https://github.com/ticarpi/jwt_tool

[20] nabla-c0d3, "*SSLyze.*" [Online]. Available: https://github.com/nabla-c0d3/sslyze

[21] TCPDUMP Group, "*Tcpdump and Libpcap.*" [Online]. Available: https://www.tcpdump.org

[22] jtpereyda, "*boofuzz.*" [Online]. Available: https://github.com/jtpereyda/boofuzz

[23] Peach Tech, "*Peach Fuzzer Professional*". [Online]. Available: https://www.peach.tech

[24] NCC Group, "*Scout Suite: Multi-Cloud Security Auditing Tool.*" [Online]. Available: https://github.com/nccgroup/ScoutSuite

[25] Prowler, "*Prowler: Open-source cloud security platform.*" [Online]. Available:https://github.com/prowler-cloud/prowler

[26] Rhino Security Labs, "*Pacu: The AWS exploitation framework.*" [Online]. Available:https://github.com/RhinoSecurityLabs/pacu

[27] Aqua Security, "*Trivy*." [Online]. Available: https://github.com/aquasecurity/trivy

[28] OWASP, "*Dependency-Check*." [Online]. Available:https://owasp.org/www-project-dependency-check/

[29] Amazon Web Services, "*AWS IoT Device Defender developer guide*." [Online]. Available:https://docs.aws.amazon.com/iot/latest/developerguide/device-defender.html

[30] Microsoft, "*Microsoft Defender for IoT documentation*." [Online]. Available:https://learn.microsoft.com/en-us/azure/defender-for-iot/

[31] Eclipse Foundation, "*Eclipse Mosquitto*." [Online]. Available:https://mosquitto.org

[32] AWS Command Line Interface Documentation. [Online]. Available: https://docs.aws.amazon.com/cli

[33] Azure Command-Line Interface (CLI) documentation. [Online]. Available: https://learn.microsoft.com/en-us/cli/azure/?view=azure-cli-latest

[34] Aircrack-ng, "*Aircrack-ng*". [Online]. Available: https://www.aircrack-ng.org

[35] Kismet Wireless, "*Kismet: Wi-Fi, Bluetooth, RF, and more*." [Online]. Available:https://www.kismetwireless.net

[36] Hak5, "*The PineAP Suite*". [Online]. Available: https://docs.hak5.org/wifi-pineapple-6th-gen-nano-tetra/getting-started/the-pineap-suite

[37] Flipper Zero, "*Flipper Zero documentation*." [Online]. Available:https://docs.flipper.net/zero

[38] hashcat, "*hashcat password recovery utility*." [Online]. Available:https://hashcat.net/hashcat

[39] Great Scott Gadgets, "*Ubertooth*". [Online]. Available: https://github.com/greatscottgadgets/ubertooth

[40] virtualabs, "*btlejack: Bluetooth Low Energy Swiss-army knife*." [Online]. Available:https://github.com/virtualabs/btlejack

[41] BlueZ Project, "*BlueZ*." [Online]. Available:https://www.bluez.org

[42] Bettercap — the Swiss army knife for network attacks and monitoring https://www.bettercap.org

[43] Mike Ryan, "*crackle*". [Online]. Available: https://github.com/mikeryan/crackle

[44] D. Mantz, J. Classen, M. Schulz, and M. Hollick, "*InternalBlue - Bluetooth Binary Patching and Experimentation Framework.*" [Online]. Available:https://github.com/seemoo-lab/internalblue

[45] Ettus Research, "*USRP Hardware Driver and USRP Manual*". [Online]. Available: https://files.ettus.com/manual

[46] Great Scott Gadgets, "*HackRF One.*" [Online]. Available:https://greatscottgadgets.com/hackrf/one

[47] Software Radio Systems, "*srsRAN Project Documentation.*" [Online]. Available:https://docs.srsran.com/projects/project

[48] OpenLTE, "*OpenLTE*". [Online]. Available: https://sourceforge.net/projects/openlte/

[49] Open5GS, "*Open5GS documentation.*" [Online]. Available:https://open5gs.org/open5gs/docs

[50] OpenBTS, "*OpenBTS.*" [Online]. Available: http://openbts.org

[51] L3Harris, "*StingRay II cellular exploitation tool.*" [Online]. Available:https://www.l3harris.com/all-capabilities/stingray-ii-cellular-exploitation-tool

[52] Osmocom, "*Osmocom open-source mobile communications projects.*" [Online]. Available:https://osmocom.org

[53] Wireshark, "*Network Protocol Analyzer.*" [Online]. Available:https://www.wireshark.org

[54] RfidResearchGroup, "*Iceman Fork - Proxmark3*". [Online]. Available: https://github.com/RfidResearchGroup/proxmark3

[55] emsec, "*ChameleonMini.*" [Online]. Available:https://github.com/emsec/ChameleonMini

[56] osqzss, "*gps-sdr-sim: Software-Defined GPS Signal Simulator.*" [Online]. Available:https://github.com/osqzss/gps-sdr-sim

[57] GNSS-SDR, "*GNSS-SDR.*" [Online]. Available:https://gnss-sdr.org

[58] riebl, "*Vanetza: Open-source implementation of the ETSI C-ITS protocol suite.*" [Online]. Available: https://github.com/riebl/vanetza

[59] scapy-radio, "*scapy-radio*". [Online]. Available: https://github.com/resurrecting-open-source-projects/scapy-radio

[60] DeepSig, "*OmniSIG*". [Online]. Available: https://www.deepsig.ai/omnisig

[61] linux-can, "*can-utils*: Linux-CAN SocketCAN user space applications." [Online]. Available:https://github.com/linux-can/can-utils

[62] cantools, "*cantools*". [Online]. Available: https://github.com/cantools/cantools

[63] Collin Kidder, "*SavvyCAN*." [Online]. Available:https://github.com/collin80/SavvyCAN

[64] dschanoeh, "*Kayak*." [Online]. Available:https://github.com/dschanoeh/Kayak

[65] RBEI-ETAS, "*BUSMASTER*". [Online]. Available: https://rbei-etas.github.io/busmaster/

[66] Vector Informatik, "*CANalyzer*". [Online]. Available: https://www.vector.com/int/en/products/products-a-z/software/canalyzer/

[67] CANToolz, "*CANToolz README*." [Online]. Available:https://github.com/CANToolz/CANToolz/blob/master/README.md

[68] Caring Caribou, "*Caring Caribou.*" [Online]. Available:https://github.com/CaringCaribou/caringcaribou

[69] Scapy, "*Automotive-specific documentation*." [Online]. Available:https://scapy.readthedocs.io/en/latest/layers/automotive.html

[70] python-can, "*python-can*". [Online]. Available: https://github.com/hardbyte/python-can

[71] python-can-isotp, "*python-can-isotp*". [Online]. Available: https://github.com/pylessard/python-can-isotp

[72] zombieCraig, "UDSim". [Online]. Available: https://github.com/zombieCraig/UDSim

[73] Technica Engineering, "*MediaConverter 100BASE-T1 BCM*." [Online]. Available:https://www.technica-engineering.com/mediaconverter-100base-t1

[74] Intrepid, "*RAD-Moon 2 | 100/1000BASE-T1 Media Converter*". [Online]. Available: https://intrepidcs.com/products/automotive-ethernet-tools/rad-moon2-100-1000base-t1-media-converter

[75] Nmap Project, "*Nmap documentation*." [Online]. Available:https://nmap.org/docs.html

[76] Wireshark, "*SOME/IP Protocol Display Filter Reference*." [Online]. Available:https://www.wireshark.org/docs/dfref/s/someip.html

[77] Rapid7, "*Metasploit Framework*." [Online]. Available:https://www.metasploit.com

[78] Ettercap, "*Ettercap*". [Online]. Available: https://www.ettercap-project.org

[79] tomac, "*Yersinia*". [Online]. Available: https://github.com/tomac/yersinia

[80] CyTAL, "*Automotive Protocol Security: SOME/IP, DoIP, UDS Fuzzing*". [Online]. Available: https://cytal.co.uk/blog/automotive-protocol-security-protecting-connected-vehicles-from-digital-threats

[81] Grand Idea Studio, "*JTAGulator*". [Online]. Available: https://grandideastudio.com/portfolio/security/jtagulator

[82] OpenOCD, "*Open On-Chip Debugger User's Guide*." [Online]. Available:https://openocd.org/doc/pdf/openocd.pdf

[83] SEGGER, "*J-Link debug probes*." [Online]. Available:https://www.segger.com/products/debug-probes/j-link

[84] Bus Pirate, "*Command Reference*". [Online]. Available: https://docs.buspirate.com/docs/command-reference

[85] Linux I2C, "*i2c-tools*". [Online]. Available: https://i2c.wiki.kernel.org/index.php/I2C_Tools

[86] minicom team, "*minicom*". [Online]. Available: https://salsa.debian.org/minicom-team/minicom

[87] NewAE Technology, "*ChipWhisperer documentation*". [Online]. Available: https://chipwhisperer.readthedocs.io/en/latest/getting-started.html

[88] NewAE Technology, "*ChipSHOUTER*." [Online]. Available:https://github.com/newaetech/ChipSHOUTER

[89] Riscure, "Inspector SCA". [Online]. Available: https://www.riscure.com/security-tools/inspector-sca/

[90] flashrom, "*flashrom classic CLI manual page*." [Online]. Available:https://flashrom.org/classic_cli_manpage.html

[91] ReFirmLabs, "*binwalk: Firmware Analysis Tool*." [Online]. Available:https://github.com/ReFirmLabs/binwalk

[92] QEMU, "*System Emulation documentation*." [Online]. Available:https://www.qemu.org/docs/master/system/index.html

[93] National Security Agency, "*Ghidra software reverse engineering framework.*" [Online].Available: https://github.com/nationalsecurityagency/ghidra

[94] Hex-Rays, "*IDA Pro.*" [Online]. Available:https://hex-rays.com/ida-pro/

[95] Frida, "*Frida dynamic instrumentation toolkit.*" [Online]. Available:https://frida.re

[96] radare2, "*radare2 reverse engineering framework.*" [Online]. Available:https://github.com/radareorg/radare2

[97] AFL++, "*AFL++ fuzzer.*" [Online]. Available:https://github.com/AFLplusplus/AFLplusplus

[98] Google, "*syzkaller.*" [Online]. Available: https://github.com/google/syzkaller

[99] BinaryAI, "*BinaryAI documents.*" [Online]. Available:https://binaryai.cn/doc/en/introduction

[100] Binarly, "*VulHunt in Depth.*" [Online]. Available: https://www.binarly.io/blog/vulhunt-in-depth-inside-the-binary-vulnerability-analysis-framework

[101] Google, "*oss-fuzz-gen.*" [Online]. Available:https://github.com/google/oss-fuzz-gen

[102] Code Intelligence, "*CI Fuzz Spark AI Test Agent.*" [Online]. Available:https://docs.code-intelligence.com/ai-test-agent/spark

[103] Saleae, "*I2C Analyzer user guide.*" [Online]. Available:https://support.saleae.com/product/user-guide/protocol-analyzers/analyzer-user-guides/using-i2c

[104] merbanan, "*rtl_433*". [Online]. Available: https://github.com/merbanan/rtl_433

[105] RTL-SDR Blog, "*RTL-SDR Blog V3 user guide.*" [Online]. Available:https://www.rtl-sdr.com/rtl-sdr-blog-v3-dongles-user-guide/

[106] Johannes Pohl, "*Universal Radio Hacker (URH)*". [Online]. Available: https://github.com/jopohl/urh

[107] GNU Radio, "*GNU Radio.*" [Online]. Available:https://www.gnuradio.org

[108] jboone, "*gr-tpms*". [Online]. Available: https://github.com/jboone/gr-tpms

[109] Great Scott Gadgets, "*Yard Stick One*". [Online]. Available: https://greatscottgadgets.com/yardstickone/

[110] Trusted-AI, "*Adversarial Robustness Toolbox (ART).*" [Online]. Available: https://github.com/Trusted-AI/adversarial-robustness-toolbox

[111] Microsoft Azure, "*Counterfit*." [Online]. Available:`https://github.com/Azure/counterfit`

[112] Bethge Lab, "*Foolbox*." [Online]. Available: `https://github.com/bethgelab/foolbox`

[113] Leon Derczynski, "*garak*". [Online]. Available: `https://github.com/leondz/garak`

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

Part 2

Attacking the External Interfaces and Vehicle Networks

This part moves from theory to practice, examining the externally reachable attack surfaces that expose modern vehicles to remote adversaries. You will learn how to probe and exploit cloud backend services, wireless communication interfaces, and in-vehicle network protocols, progressing from the outermost connectivity layers inward toward the vehicle communication backbone. Each chapter combines an analysis of common weaknesses with targeted penetration testing techniques and real-world case studies drawn from major OEM incidents. By the end of this part, you will have the practical skills to assess and challenge the security of the connectivity stack that underpins the connected vehicle.

This part has the following chapters:

- *Chapter 4, Attacking the Vehicle Backend*
- *Chapter 5, Attacking the Wireless Vehicle Systems*
- *Chapter 6, Attacking the In-Vehicle Communications*

4

Attacking the Vehicle Backend

Modern connected vehicles have expanded their reliance on backend systems considerably in the last few years as customers have grown accustomed to the convenience of direct remote access to their vehicle data and functions. Whether it is receiving **over-the-air** (**OTA**) software updates at home rather than at the dealership or remotely unlocking your car so a friend can fetch something from the backseat, the vehicle backend services continue to be an attractive selling point for modern vehicles. Not only is it good for the consumers, but **original equipment manufacturers** (**OEMs**) also enjoy the cost savings associated with reduced dealership service fees, as well as the big data collection that offers them useful insights into driver and vehicle performance.

Unfortunately, as with almost all features that add convenience through expanded connectivity, there is a hidden cost to vehicle security. Naturally, enabling remote services extends the vehicle attack surface to include the backend, and given their open nature, our focus will be particularly on the cloud-based vehicle backend architectures. The latter is essentially an **Internet of Things** (**IoT**) cloud architecture customized for vehicles.

In this chapter, we start by understanding the basics of vehicle backend IoT architectures and their most common communication protocols. We then survey common weaknesses that apply to these systems as we set the stage for exploitation techniques. Next, we will discuss some penetration testing tools and techniques that apply to these systems before we turn our attention to real-life case studies in which the vehicle backend was compromised, resulting in a vehicle impact. Finally, we will highlight some security best practices to increase the security resilience of backend architectures by employing a zero-trust approach.

The rest of this chapter is broken into the following main topics:

- Vehicle backend architecture overview
- Understanding backend connectivity protocols
- Common weaknesses in vehicle backend architectures
- Penetration testing techniques
- Case studies of real-life attacks on vehicle backends
- Best practices for securing the vehicle backend infrastructure

Vehicle backend architecture overview

Modern vehicle backend architectures typically leverage cloud platforms to connect vehicles (as edge devices) with backend services. Infrastructures like **Microsoft Azure** [1] and **Amazon Web Services** (**AWS**) [2] enable bi-directional communication for telematics data, remote vehicle commands, analytics, and integration with business systems. Part of the reason for the increased proliferation of cloud-based vehicle backends is the wide range of services that can be provided, including the following:

- **Telematics data collection and monitoring**: Continuous upload of vehicle telemetry (GPS location, speed, sensor readings, engine status, and so on) to the cloud. This supports real-time vehicle monitoring, fleet management dashboards, and predictive maintenance analytics.
- **OTA updates**: Services to deliver software updates or configuration changes from the cloud to the vehicle's **Electronic Control Units** (**ECUs**).
- **Artificial intelligence** (**AI**) **training**: Collect labeled image data to enable **machine learning** (**ML**) model training about specific driving scenarios.
- **Remote command and control**: Ability for users or fleet operators to send commands to the vehicle from a mobile app or web portal. Common commands include remote door lock/unlock, engine start/stop, honking the horn, flashing lights, climate control activation, and finding the vehicle's location.
- **Remote status updates**: Integration with companion mobile apps and third-party services enable companion apps to receive the car's status (fuel or battery level, tire pressure), as well as receive alerts, schedule service appointments, or even make feature purchases.

As evident by the list of use cases, these services create opportunities for attackers to cause severe **safety**, **financial**, **operational**, and **privacy** damages if compromised. Whether it is unlocking someone's vehicle remotely for theft purposes, tracking a victim's location, or issuing malicious software updates, security vulnerabilities in the backend can and will result

in adverse consequences. Take, for example, the collection and transmission of training data back to the cloud scenario. An attacker who can falsify such information has the potential of poisoning ML models that, in turn, can result in safety hazards. To ensure the security of these services, one must consider the security of the end-to-end solution, starting from the backend and down to the last vehicle component that is contributing to the service.

In later chapters of the book, we will study individual segments of this communication pipeline, but for now, our focus will be strictly on the backend itself. Before we get into exploitation techniques against the backend, we need to understand the main building blocks and communication services that make up a vehicle backend architecture, illustrated and described in *Figure 4.1*.

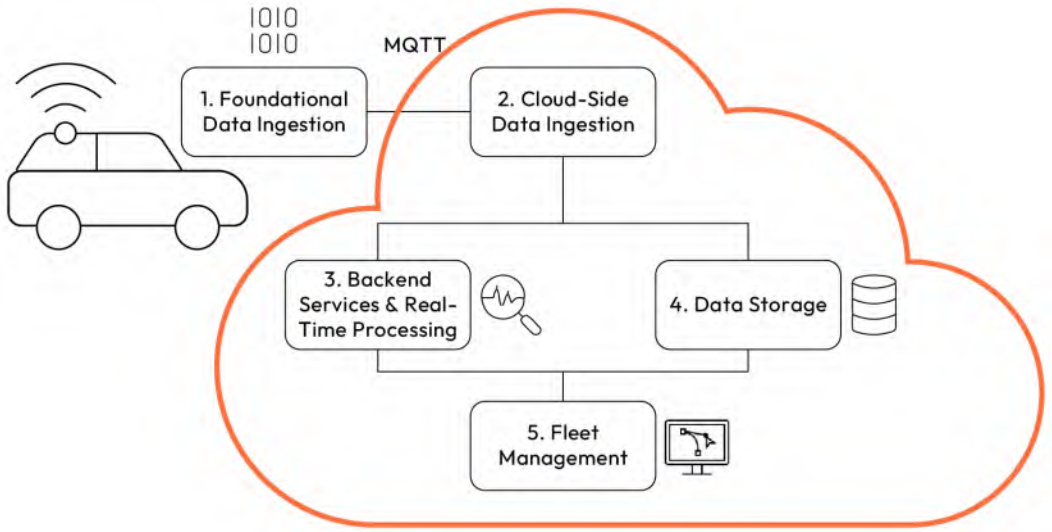


Figure 4.1: Five main building blocks of a connected vehicle backend architecture

There are five main building blocks described as follows:

1. **Foundational data ingestion - MQTT and telematics integration**: On the vehicle side, data is captured and curated using onboard telematics or gateway ECU. Typically, in-vehicle telemetry, including engine performance, sensor outputs, and **Advanced Driver Assistance Systems** (**ADAS**) status, is published through lightweight protocols like **Message Queuing Telemetry Transport (MQTT)**, chosen for its efficiency over constrained network links.
2. **Cloud-side data ingestion and event-driven orchestration**: On the backend side, a cloud-based MQTT broker receives these messages and processes them for distribution. That is, once data reaches the cloud, specialized ingestion services (in

Azure, often an IoT Hub or an MQTT-based ingestion layer [3]) verify the data's source, apply protocol rules, and forward relevant events to the system's core. Here, an event router (such as Event Grid) [4] steps in to distribute these notifications. By adopting event-driven architecture, engineers can decouple each subsystem, allowing new services to tap into the same event feed without altering the ingestion pipeline.

3. **Backend services and real-time processing**: Following ingestion, event subscribers like Azure Functions or custom backend services dissect the streaming data. Using predefined rules, they might assess conditions for alerts (e.g., tire pressure anomalies) or run data processing functions to convert the data to the desired message formats. By matching these raw signals against known thresholds or rules, the system can generate feedback to the user, such as a safety-critical event.
4. **Data storage for insights and accountability**: In addition to real-time notifications, there are use cases that depend on the persistent storage of events. The persistent storage layers include the following two:
    - **Operational databases** (e.g., Azure Cosmos DB): Store recent telematics or status updates, enabling lookups for diagnostics, usage tracking, or user-facing dashboards
    - **Data lakes / cold storage** (e.g., Azure Data Lake): Retain large volumes of historical vehicle data to support advanced analytics, predictive maintenance, and post-incident investigations
5. **Fleet management and telemetry visualization**: Once this architecture is established, fleet management portals or OEM dashboards can aggregate, and display insights alerted from real-time processing or analyzed from the stored data. Using real-time feeds, data logs, and ML-based anomaly detection, these applications help identify potential malfunctions, coordinate OTA firmware updates, and maintain an at-a-glance perspective on fleet health.

Given the above quick overview, we should be able to start seeing the value proposition of this new computing paradigm for OEMs and users alike.

Vehicle unlock use case example

In this section, we will dive deeper into specifics as we explore how a vehicle unlock use case may be realized using the AWS Connected Vehicle Reference Architecture [5] as shown in *Figure 4.2*.

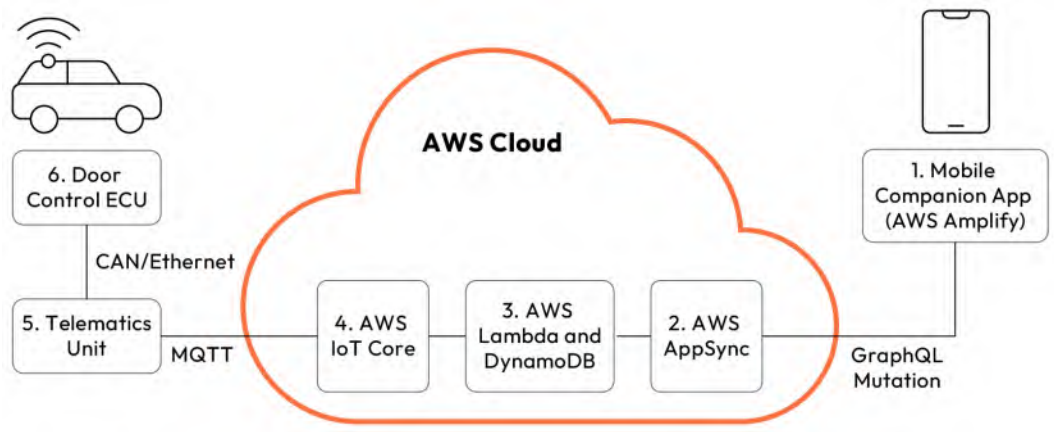


Figure 4.2: AWS connected vehicle reference architecture showing end-to-end message flow (adapted from [5])

The process moves in the following pattern:

1. The mobile companion app sends a secure remote vehicle unlock command via AWS Amplify as a GraphQL mutation to AWS AppSync
2. AWS AppSync receives the mutation and routes it to the backend processing layer
3. This command is stored in Amazon DynamoDB, triggering an AWS Lambda function that checks the vehicle's current connectivity state using the device shadow maintained in AWS IoT Core
4. If the vehicle is online, the Lambda function publishes the unlock command to AWS IoT Core with a unique request identifier and designated MQTT response topic
5. The vehicle's telematics unit, which maintains a persistent MQTT connection to AWS IoT Core, receives the command and prepares to execute the unlock operation
6. The telematics unit sends the appropriate **Controller Area Network** (**CAN**) or Ethernet message to the door control ECU, which physically executes the unlock

After executing the command, the vehicle publishes a response message to the designated MQTT response topic. This response is processed by the backend Lambda function, which updates the command status and vehicle state in DynamoDB. Finally, the updated status is pushed back to the mobile application through an AWS AppSync subscription, allowing the user to receive near real-time confirmation that the vehicle has been unlocked.

Note

Terms:

- AWS AppSync is a managed service that helps you build and run GraphQL APIs to access and update data in the cloud, while AWS Amplify is a toolkit that simplifies integrating these and other cloud services into your mobile or web applications.
- GraphQL is a query language for APIs that lets clients request the data they need, making interactions with servers more flexible compared to traditional Representational State Transfer (REST) Application Programming Interfaces (APIs).
- A mutation is a GraphQL operation sent by the client that requests the backend to perform an action or change the system state.
- A device shadow is essentially a digital representation of the corresponding vehicle in the cloud. It is a JavaScript Object Notation (JSON) document that stores the vehicle's last known state as well as any desired state updates.
- AWS Lambda function is a serverless compute service that lets you run custom code in response to events, such as GraphQL operations, without needing to manage any servers.
- DynamoDB is a fully managed NoSQL database service by AWS that stores data as key-value and document models.

Evidently, this brief introduction is not enough to enable you to perform an effective penetration test campaign against a cloud backend. There are books that teach how to perform security testing of Azure, AWS, and Google backend systems, and you are certainly encouraged to study that in further detail if that is the focus of your work.

With the vehicle backend architecture established as a foundation, the next section examines the connectivity protocols that define how vehicles, mobile applications, and backend services communicate with one another.

Understanding backend connectivity protocols

Backend services would not be possible without the connectivity protocols that link the backend to the vehicle through a communication gateway (typically called **Telematics**

Control Unit (**TCU**)). Vehicles rely on cellular networks to maintain an internet connection to the cloud using 4G/LTE/5G modems. Over the years, several protocols have been employed to enable vehicle-to-backend connectivity, which we survey in the following subsections.

HTTP/REST APIs

Many vehicle services use **Hypertext Transfer Protocol (Secure)** (**HTTP(S)**) **Application Programming Interfaces** (**APIs**), especially for interactions initiated from cloud to vehicle or vice versa through an intermediary. For instance, a vehicle might periodically send **Representational State Transfer** (**REST**)-based POST requests with batched telemetry. More commonly, mobile apps or web apps use REST APIs to talk to the cloud backend (for user login or sending a command which the cloud then forwards to the vehicle).

An API gateway in the backend routes incoming API requests to the target backend service. It can also handle caching of requests and responses to reduce the load on the backend service. Typically, it also handles authentication and authorization using **Open Authorization** (**OAuth**) and **JSON Web Tokens** (**JWT**). Moreover, a common security feature is to enable whitelisting of **Internet Protocol** (**IP**) addresses that are allowed to access the backend service. This is prudent considering that certain services should only be accessible by enrolled vehicles and devices rather than by the entire internet.

WebSockets

Some real-time services use WebSocket connections (which provide full-duplex communication over a single **Transmission Control Protocol** (**TCP**) connection). For instance, a vehicle could maintain a WebSocket to a cloud endpoint for instant push of events. WebSockets or similar protocols might be used for streaming telemetry (e.g., live telemetry dashboards).

SMS and legacy telephony

Older telematics systems sometimes fall back to **Short Message Service** (**SMS**) for certain commands, for example, when sending a vehicle wake-up message. However, this is largely a legacy protocol now, as most modern connected cars prefer IP data for all communications.

AMQP, XMPP and CoAP

Advanced Message Queuing Protocol (**AMQP**) is supported by Azure IoT Hub as an alternative to MQTT/HTTP. Some automotive systems may use AMQP or even **Extensible Messaging and Presence Protocol** (**XMPP**) historically, but MQTT has largely overtaken these due to its lightweight nature. Another protocol in IoT is **Constrained Application Protocol** (**CoAP**), though it is more common in constrained IoT and not widely seen in vehicles.

Diagnostics protocols

Remote diagnostics, including predictive diagnostics, is a trend widely adopted among OEMs. **Diagnostics over IP** (**DoIP**), standardized as ISO 13400 [6], has commonly been used in diagnostic tools and provides remote diagnostics via a gateway, as it is mature and widely supported by ECUs. However, to support more cloud-native solutions, the industry is moving toward **Service-Oriented Vehicle Diagnostics** (**SOVD**), which is a REST-style diagnostics API standardized as ISO 17978 [7]. SOVD has several benefits as it is cloud-friendly (HTTP/HTTPS), stateless, easier to secure (e.g., using OAuth 2.0 and **mutual Transport Layer Security** (**mTLS**)), and can scale across fleets.

MQTT as a de facto standard

Among the communication protocols presented above, MQTT has emerged as the de facto standard protocol for IoT and connected car communications.

MQTT is a lightweight publish/subscribe messaging protocol ideal for devices with limited bandwidth or intermittent connectivity [8,9]. The vehicle TCU runs an MQTT client that maintains a persistent mTLS connection to the cloud broker in the automotive backend, such as AWS IoT Core or Azure IoT Hub. Telemetry data is published on specific topics (vehicle ID is often part of the topic), and the vehicle can subscribe to command topics for receiving remote control commands. MQTT's design handles unreliable networks by queuing messages and supporting various **Quality of Service** (**QoS**) levels. To get a closer feel of how MQTT is used, let's consider a typical flow involving MQTT messaging:

1. First, the vehicle TCU connects to the cloud MQTT broker (e.g., vehicle-iot.cloud.com) using its client certificate
2. It publishes a message like `{ "tirePressure": 32, "fuelLevel": 50 }` to topic `vehicles/<VIN>/telemetry`
3. The cloud, upon receiving this message, processes and stores the data, possibly raising alerts if values are abnormal

In another example, when the user in a mobile app presses the **Unlock Car** button, the backend issues an MQTT publish on topic `vehicles/<VIN>/commands/unlock`. The TCU subscribed to that topic, receives the command and then performs the unlock action on the car. This entire exchange must be properly authenticated and encrypted to prevent any unauthorized or spoofed messages.

It goes without saying that all communication between the vehicle and the cloud must be secured. Industry best practices dictate using **Transport Layer Security** (**TLS**) encryption for data in transit and strong authentication of the vehicle connectivity gateway. In practice, this means mTLS is used in many platforms: typically, the TCU holds a device certificate, and the

cloud broker requires the vehicle to authenticate with that certificate, while the vehicle also verifies the cloud server's certificate. AWS, for example, mandates mTLS with X.509 certificates for vehicles connecting to AWS IoT Core. Additionally, authentication tokens or API keys may be used for certain API calls (for example, a REST endpoint might require an OAuth 2.0 access token associated with the user or vehicle).

Note

Terms:

- X.509 certificates are digital certificates that bind a public key to the identity of a device, server, or user, signed by a trusted certificate authority (CA) to enable authentication in secure communications. X.509 certificates are widely used in protocols like TLS/mTLS to enable authentication, encryption, and trust between communicating systems, e.g., between a vehicle's TCU and a cloud service.
- OAuth 2.0 access tokens are short-lived tokens issued by an authorization server that allow an application to access protected APIs on behalf of a user or service without sharing credentials.

As we have seen so far, vehicles are IoT devices on wheels, and they rely on cloud services for extended functionality. Readers interested in learning more about building secure automotive IoT applications are encouraged to explore this resource [10].

The next section examines the security weaknesses that can arise in these vehicle backend architectures.

Common weaknesses in vehicle backend architectures

Connected vehicle systems inherit many of the same vulnerabilities that plague IoT and web applications. A car's cloud backend is essentially a collection of web APIs and IoT endpoints; thus, the Open Worldwide Application Security Project (OWASP) Top 10 API Security Risks [11] and certain MITRE common weaknesses [12] are highly relevant. In addition, adopting appropriate vulnerability management approaches to provide a practical framework for triaging, tracking, and remediating vulnerabilities is becoming increasingly important for automotive organizations [13]. In recent years, numerous security incidents have shown that vehicles are vulnerable not just via physical access, but via attacks on cloud services and APIs. In fact, one study from 2023 found a 380% increase in API-related incidents in the automotive

domain compared to previous years, highlighting how backend weaknesses are becoming a prime target [14].

We present several categories of common security weaknesses in vehicle backend architectures illustrated in *Figure 4.3* and described as follows:

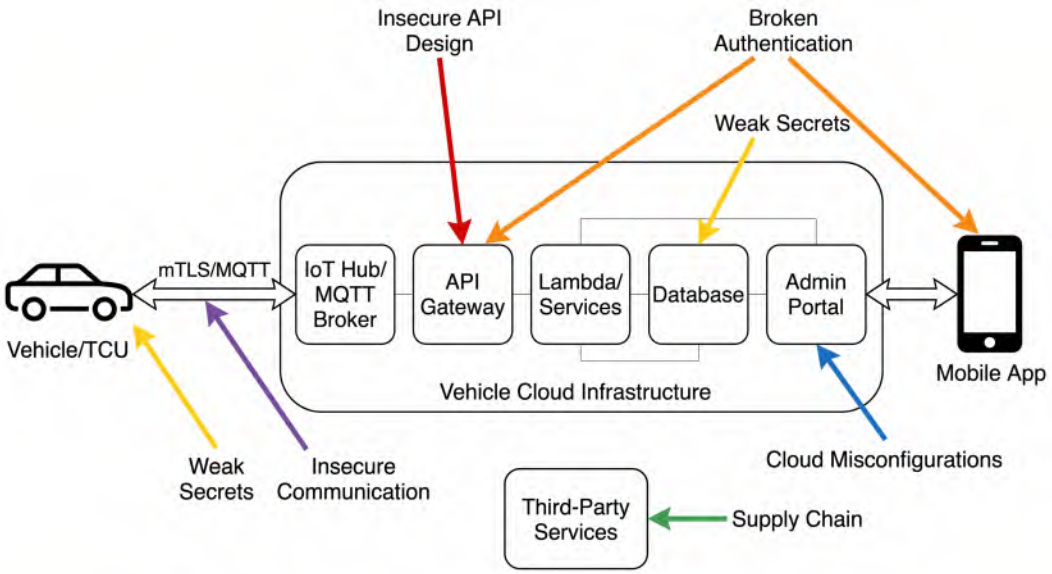


Figure 4.3: Overview of common weaknesses in the vehicle backend architecture

Insecure API design and input validation weaknesses

Many vehicle backend services are essentially web applications (for instance, portals or APIs that take parameters). If these inputs are not sanitized, classic injection vulnerabilities (**Structured Query Language** (**SQL**) injection, command injection) can occur. This is exemplified by the Spireon telematics platform case, where an SQL injection vulnerability in a cloud administration portal allowed researchers to gain admin access [15]. The OWASP Top 10 API Security Risks lists **Injection** and **Improper Assets Management** as common issues, and the automotive incidents reflect this. Unvalidated input could allow an attacker to run unauthorized code on backend servers or manipulate data. For instance, a poorly filtered parameter might let an attacker inject an SQL command to dump user credentials.

As we have demonstrated so far, connected vehicles rely on web services and APIs in the cloud (for mobile apps, web portals, etc.), making them the dominant attack surface in most architectures. These APIs typically expose endpoints tied to VINs, vehicle identifiers, user accounts, and fleet groupings. When API design fails to properly validate input or sanitize parameters, attackers may manipulate requests to access or influence vehicles beyond their

authorized scope. Improper validation of VIN fields, vehicle IDs, or account identifiers can lead to cross-vehicle data access, unauthorized remote command execution, or exposure of sensitive telemetry such as location history and diagnostic information. Additionally, inadequate rate limiting may enable large-scale VIN enumeration attacks. Business logic weaknesses are particularly critical in automotive systems, where an attacker might exploit workflow assumptions in remote command processing, charging control logic, or OTA scheduling mechanisms. Unlike conventional web services, flaws in vehicle backend APIs can propagate operational impact across entire vehicle populations.

Broken authentication and authorization controls

Vehicle backend systems manage multiple identity domains: end users, fleet operators, dealer technicians, backend services, and vehicles themselves. Authentication weaknesses may involve improperly implemented mobile app login flows, insecure token generation, insufficient multi-factor authentication, or poorly validated session management. Tokens associated with vehicle ownership or fleet roles must be tightly scoped and cryptographically protected. Improper authentication could lead to unauthorized access to vehicle APIs or services. In some past examples, APIs lacked authentication or relied on easily guessable identifiers. For example, the backend API serving the Nissan Leaf companion app was found to accept commands if the correct VIN was provided, with no password required, which is essentially a poorly authenticated API that allows attackers to remotely trigger functions like climate control [16].

Broken authorization (e.g., Insecure Direct Object References) is another facet where an API fails to properly verify that the requesting user or device has permission to access a given vehicle's data or resources. This can lead to one customer's account being able to access another account's vehicle by tweaking an identifier in requests. That is, if backend APIs rely on client-supplied vehicle identifiers without revalidating ownership or fleet assignment, attackers may retrieve data from, or issue commands to, vehicles they do not control. Moreover, dealer systems present additional complexity, as service accounts may require elevated privileges for diagnostics or firmware flashing. Without strict role segmentation, attackers could escalate from consumer-level access to dealer-level or fleet-administrator capabilities. In a large-scale connected vehicle environment, authorization weaknesses represent one of the most severe systemic risks.

Weak secrets or credential management

Automotive backend ecosystems depend on a complex hierarchy of secrets, including telematics authentication keys, OTA signing certificates, API tokens, and service account credentials. Weak secrets management practices, such as embedding hardcoded passwords or API keys in TCUs or mobile applications, reusing static credentials across services, or failing to rotate signing keys, can undermine the integrity of the entire vehicle platform.

A striking case involved the MiCODUS MV720 GPS trackers used in vehicles: researchers found the devices and their cloud API shared a default password "123456" and even a hardcoded master password, which 94% of tested devices were still using [17]. This meant an attacker could easily authenticate to the backend as any device. Hardcoded API keys or secrets in mobile apps or firmware similarly pose a risk, where if an attacker could extract those credentials, they could impersonate the respective vehicles or servers. Moreover, regarding automotive backends, there have been instances of contractors accidentally leaking credentials. For example, in 2022, a source code repository uploaded to GitHub was discovered to contain hardcoded keys that exposed access to a customer data database [18]. Because automotive systems often operate on a global scale, exposed credentials can create cascading compromise risks affecting large fleets or entire production lines.

Insecure communication

Connected vehicles continuously exchange telemetry, diagnostic data, and remote command acknowledgments with the vehicle backend infrastructure. The link between vehicle and backend (typically over cellular or Wi-Fi) is, therefore, naturally another potential attack vector, though modern encryption makes this more challenging. Communication weaknesses may involve misconfigured TLS implementations, weak cipher configurations, improper certificate validation, or the absence of mutual authentication between vehicles and backend services. If vehicle-to-backend communication is not properly encrypted and authenticated, an attacker could intercept or spoof messages. Earlier connected car systems had issues like a lack of HTTPS enforcement or not validating TLS certificates. Modern vehicles generally use TLS and often operate on private **Access Point Names** (**APNs**) (closed cellular networks), which makes interception harder, but misconfigurations (like accepting any certificate or using outdated ciphers) can undermine security.

Cloud infrastructure flaws

Vehicle backend platforms are typically built on large-scale cloud infrastructure supporting telematics ingestion, fleet dashboards, OTA orchestration, and dealer service systems. Typical flaws of any cloud backend include misconfigurations, unpatched software, weak internal access controls, and vulnerabilities in the cloud infrastructure. An example of this happened when, at one automaker, an improperly configured **Single Sign-On** (**SSO**) system allowed access to hundreds of internal applications [15]. Through this, the researchers accessed mission-critical internal tools (like build servers, **Continuous Integration/Continuous Delivery** (**CI/CD**) pipelines, internal databases, etc.) and even achieved remote code execution on some backend systems.

This highlights that a vehicle backend is not just one server; it is an ecosystem including databases with customer data, developer tools, and more. It is noteworthy that an attacker who finds a weakness in the backend's corporate IT (like an exposed admin interface or default

credentials on a cloud service) can pivot to deeper control. Because automotive backends often serve millions of vehicles simultaneously, infrastructure misconfiguration can result in systemic vulnerabilities leading to large-scale impact rather than isolated incidents.

Supply chain and third-party integration weaknesses

Automakers often rely on third-party providers for connectivity services such as telematics service providers (e.g., Sirius XM provides connected vehicle services for many brands [19]), map or traffic data providers, fleet management platforms, and even digital license plate services. If these systems have vulnerabilities or use vulnerable libraries, an attacker could exploit them to attack all vehicles or users relying on that service. Rather than attacking each car directly, an attacker can target a central service provider that many vehicles trust. It is a powerful attack vector that underscores the importance of vetting and securing any third-party connected vehicle service.

Furthermore, third-party integrations in the vehicle backend frequently rely on OAuth tokens or API keys. If proper validation is lacking or the scopes are overly permissive, attackers may pivot through trusted external systems into the core vehicle backend.

Now that we have surveyed the common weaknesses that impact vehicle backend systems, we will explore the penetration testing techniques that leverage such weaknesses to compromise vehicle systems.

Penetration testing techniques

Penetration testing of vehicle backends should be approached as a structured attack campaign that moves through reconnaissance, vulnerability assessment, and exploitation. Reconnaissance identifies exposed backend assets, APIs, trust relationships, communication paths, and third-party integrations. Vulnerability assessment then determines where authentication, authorization, input validation, secret handling, or configuration weaknesses exist. Exploitation finally demonstrates how those weaknesses can be abused to access vehicle data, issue remote commands, escalate privileges, or pivot deeper into the connected vehicle ecosystem. This workflow blends classic web and API security testing with IoT-specific techniques. Tools such as **Burp Suite** [20], **Postman** [21], **Frida** [22], and **Shodan** [23] support the cloud-facing assessment, while hardware debuggers, firmware extraction, and reverse engineering tools can support deeper investigation of trusted vehicle-side components such as TCUs and companion systems.

As we saw earlier in this chapter, the target systems span cloud services, mobile apps, and the vehicle connectivity embedded systems, so a versatile set of techniques and tools is needed. The following techniques are most effective when applied in the same sequence an attacker

would typically follow: first mapping the attack surface, then validating weaknesses, and finally demonstrating how those weaknesses can be exploited.

Cloud reconnaissance and misconfiguration exploitation

This phase supports reconnaissance by identifying exposed telematics assets, dealer portals, OTA infrastructure, staging systems, and other trust boundaries that define the backend attack surface. Cloud reconnaissance in the context of the vehicle backend focuses on identifying exposed telematics servers, fleet dashboards, OTA management platforms, and dealer portals. Security testers will often leverage **Open Source Intelligence** (**OSINT**) to find relevant cloud endpoints. Tools such as **Shodan**, **Censys** [47], and **SecurityTrails** [48] assist in discovering publicly accessible assets, such as an IoT broker open on the MQTT port or an exposed admin portal.

Subdomain enumeration tools like **Amass** [49] can help identify development or staging environments that may expose vehicle backend APIs. For example, searching for domain or subdomain names related to a specific car brand or telematics provider (common patterns like `api.oemname.com`, `telematics.brand.com`, or known platforms like siriusxm domain) can yield valuable targets. Moreover, public documentation or forums sometimes inadvertently expose API details as well. That is, a tester could search on GitHub for references to a specific OEM's API endpoints or code. The outcome of this phase is a prioritized map of reachable assets and likely entry points for deeper testing.

Communication testing and protocol analysis

This phase bridges reconnaissance and vulnerability assessment by showing how vehicles, mobile apps, dealer tools, and backend services authenticate and exchange commands, telemetry, and software updates.

Vehicle backends rely on secure communication between vehicles, mobile applications, dealer tools, and cloud services. Communication security testing evaluates TLS configurations, certificate validation logic, and mutual authentication mechanisms used in vehicle-to-backend communication. Security testers can use tools such as **Wireshark** [38], **tcpdump** [39], **sslyze** [40], and **Nmap** [41] to analyze TLS handshakes and identify weak protocol configurations. Moreover, using Wireshark, testers can decode MQTT, HTTP, etc., which can reveal if messages are encrypted and what protocols are in use. Man-in-the-middle testing can be performed using **Burp Suite**, **OWASP ZAP** [24], or **mitmproxy** [42] to determine whether mobile applications properly validate backend certificates. Testers can then assess whether replay attacks are possible against remote vehicle command APIs or OTA request endpoints.

If the vehicle uses, for instance MQTT, security testers can use an MQTT client tool (like **MQTT Explorer** [43] or **Eclipse Mosquitto** [44]) to attempt connections. Of course, this requires credentials, but testers might obtain those from the device or app. If a tester can extract a

device certificate (perhaps via hardware disassembly or an exploit in the TCU), they could try to connect as the vehicle to test the broker's behavior. In addition, testers could target MQTT and fuzz topic names or payload sizes to see if the broker does not enforce strict policies.

Testers can also use CAN interface tools (like a CANtact [45] or PCAN device [46]) if they want to see how the TCU interacts with the in-vehicle network when a backend command is received which can expose if any unsafe commands are directly passed to the CAN bus. If the TCU supports Wi-Fi or Bluetooth, those interfaces could also be entry points to pivot into the backend. For instance, some TCUs leave AT command interfaces reachable via SMS or Bluetooth, which an attacker could use to inject CAN messages or reconfigure APN settings, that is, the cellular access point name parameters that control which network gateway the TCU uses to reach the vehicle backend. Understanding these communication paths helps testers determine where trust can be broken, replayed, intercepted, or impersonated during later exploitation.

Secret discovery and credential abuse

This phase supports vulnerability assessment by identifying credentials, keys, certificates, and tokens that may allow testers to bypass intended trust boundaries or impersonate legitimate components. Automotive ecosystems rely heavily on cryptographic keys and credentials to authenticate vehicles, mobile apps, TCUs, and OTA infrastructure. Secret discovery testing focuses on identifying hard-coded passwords or API keys in mobile companion apps, embedded credentials in TCUs, and service account tokens in backend repositories.

Mobile analysis tools such as **MobSF** [29], **Frida**, **Magisk** [30], **Jadx** [31], and **APKTool** [32] are used to analyze vehicle companion apps for exposed telematics endpoints or embedded secrets. Reverse engineering may reveal API keys used to authenticate remote vehicle command services. Reverse engineering can also reveal how it computes signatures or nonces for requests, whether these algorithms are weak, and whether testers can replicate or bypass them.

Security testers who have access to a vehicle or TCU can also perform hardware testing. Hardware tools like debuggers or **Joint Test Action Group (JTAG)**/**Serial Wire Debug (SWD)** interfaces allow dumping the firmware. Firmware analysis with tools like **Binwalk** [33], **Ghidra** [34] or **IDA Pro** [35] can uncover embedded credentials or URLs. As discussed further in *Chapters 7* and *8*, firmware extraction and reverse engineering can reveal the secrets and trust assumptions that connect vehicle-side systems to the backend.

Cloud-based secrets can be evaluated by scanning repositories using tools like **truffleHog** [36] or **GitLeaks** [37], and by reviewing CI/CD configurations for exposed OTA signing keys or backend service tokens. Testers can also assess whether vehicle backend credentials are rotated

appropriately and scoped according to least privilege. It is important to consider that a leaked signing key or telematics token could enable unauthorized remote vehicle interaction at scale.

API exploitation and input manipulation

This phase moves from assessment into exploitation by validating whether exposed backend functionality can be abused through weak input validation, broken object-level authorization, or flawed business logic. API exploitation of the vehicle backend focuses on endpoints that interact with VIN-based vehicle identities, remote command interfaces, trip data, battery telemetry, and OTA update services. Since many vehicle backend interfaces are APIs, security testers should use standard web app pentesting tools. **Burp Suite** or the open-source **OWASP ZAP** is indispensable for intercepting and modifying API calls. **Postman** is useful for manually crafting and sending REST API calls with custom headers or bodies, especially once endpoints are known. Testers can configure a mobile device or emulator to route traffic through **Burp Suite**, allowing them to inspect requests from companion mobile apps or vehicle head units and backend services.

Testing often begins with enumerating telematics endpoints and analyzing how VINs, vehicle IDs, and user account identifiers are handled. Improper validation of VIN parameters may allow unauthorized access to vehicle data across accounts. **Burp Intruder** or custom **Python** scripts can be used to automate VIN enumeration attempts or test whether remote commands such as door unlock, climate control activation, or charging management are properly scoped to authorized users.

Business logic testing is particularly critical in automotive systems. For example, testers should evaluate whether remote command APIs enforce ownership checks for each request, whether fleet APIs restrict access to assigned vehicles, and whether rate limiting prevents large-scale abuse. Tools like **curl** or **Python** scripts (using the requests library) can also automate these calls for testing at scale.

Moreover, using fuzzers to provide malformed input can be revealing. For instance, a tool like **boofuzz** [25] or **Peach Fuzzer** [26] could be configured to test a REST API with random parameters to see if any HTTP 500 Internal Server Error is returned or strange behavior occurs (indicative of an injection flaw).

It is important to note that in a vehicle backend environment, a single logic flaw could potentially scale to millions of vehicles being affected. In practice, this is often the point where testers confirm whether a backend weakness can translate into unauthorized access to vehicle functions or customer data at scale.

Authentication, authorization and privilege escalation

This phase demonstrates exploitation by proving whether a low-privilege identity, compromised token, or improperly scoped account can gain access to functions or data beyond its intended role. Authentication testing in vehicle backends focuses on mobile app logins, fleet operator portals, dealer service accounts, and API-based service integrations. Intercepting proxy tools such as **Burp Suite**, **OWASP ZAP,** and **Fiddler** [27] allow security testers to analyze token issuance, session cookies, and multi-factor authentication flows. If an app uses JWT for authentication, testers might try forging tokens (e.g., if the token is not properly signed or if the backend does not verify the signature, which is a known weakness in some JWT implementations). Tools like **jwt_tool** [28] can assist in checking token vulnerabilities.

Authorization testing in automotive systems frequently revolves around VIN-based access control. Testers attempt horizontal privilege escalation by modifying VIN values in API calls to determine whether vehicle data from other customers becomes accessible. Vertical escalation scenarios may involve upgrading a consumer account to dealer-level or fleet-administrator privileges by manipulating role identifiers or permission claims.

In dealer and service networks, testers evaluate whether diagnostic endpoints, software flashing APIs, or service history databases are properly segmented. Fleet environments are examined to ensure that one fleet operator cannot access another fleet's vehicles. Because vehicle backends manage safety-critical commands and sensitive location data, even subtle authorization weaknesses can have wide-ranging operational consequences. From an attack-chain perspective, successful authorization bypass or privilege escalation can convert a narrow foothold into broad operational control over vehicle services, fleet functions, or remote command capabilities.

Supply chain and integration security testing

This phase expands the exploitation model beyond the OEM backend itself by testing whether trusted third-party services or software dependencies can serve as a path into the broader vehicle ecosystem. The topic of supply chain security has gained significant attention in recent years due to high-profile breaches, including compromises in the npm ecosystem, where trusted packages and maintainer accounts were abused to distribute malicious code across a large downstream user base [60].

Automotive backends are exposed to the typical supply chain issues that plague other industries as they depend on numerous third-party services, including map or traffic data providers, telematics service providers, and fleet management platforms. Supply chain testing is critical in finding whether vulnerable libraries could impact vehicle backend systems. Tools such as **OWASP Dependency-Check** [50] and **Trivy** [51] can help identify vulnerable components in the vehicle backend.

Security testers can also test the OAuth flows between mobile apps and backend services and API integrations with dealer management systems. **Burp Suite** and **Postman** can be used to manipulate access scopes and callback parameters to determine whether external integrations are securely implemented. It is noteworthy that backend security is not limited to OEM-owned infrastructure, but includes every trusted integration that can influence vehicles, users, or backend decisions.

Taken together, these techniques give penetration testers a structured path from identifying exposed backend assets, to validating weaknesses in trust and implementation, to demonstrating exploitability and operational impact. In other words, the backend assessment should not be viewed as a collection of disconnected tests, but as a campaign that progresses through reconnaissance, vulnerability assessment, and exploitation.

Now that we have surveyed the common penetration testing techniques, we will look at some real-life case studies in which some of these techniques were applied to achieve a successful breach of a vehicle backend and its supporting vehicle systems.

Case studies of real-life attacks on vehicle backends

In recent years, multiple high-profile incidents have demonstrated attacks on the vehicle backend. In this section, we present three such case studies. Each case study outlines how one or more vulnerabilities were discovered, how the attack was executed, and its impact on vehicles. These real-world examples underscore the hypothesis that connected vehicles share IoT backend weaknesses, and they provide lessons for engineers in what can go wrong.

Case study 1: Exploiting insecure vehicle backend API design

We start with this relatively simple attack, which demonstrates how insecure backend design can translate directly into real-world physical and privacy risks [16].

In a nutshell, researchers discovered that the Nissan vehicle backend API relied solely on the VIN sent by the NissanConnect EV App to grant access. This allowed an attacker to submit web requests containing only the target vehicle's VIN in order to remotely interact with Nissan Leaf vehicles worldwide. An attacker would gain unauthorized access to telematics data and driving history, as well as remote control of non-safety physical functions such as climate control. Worse, attackers could exploit this weakness at scale by systematically enumerating VINs to affect multiple vehicles. While this vulnerability did not impact essential driving functions like steering or braking, it exposed Nissan Leaf owners to privacy risks and potential battery depletion attacks.

> **Note**
>
> Source: `https://www.troyhunt.com/controlling-vehicle-features-of-nissan/` [16]

The overall attack flow is illustrated in *Figure 4.4*.

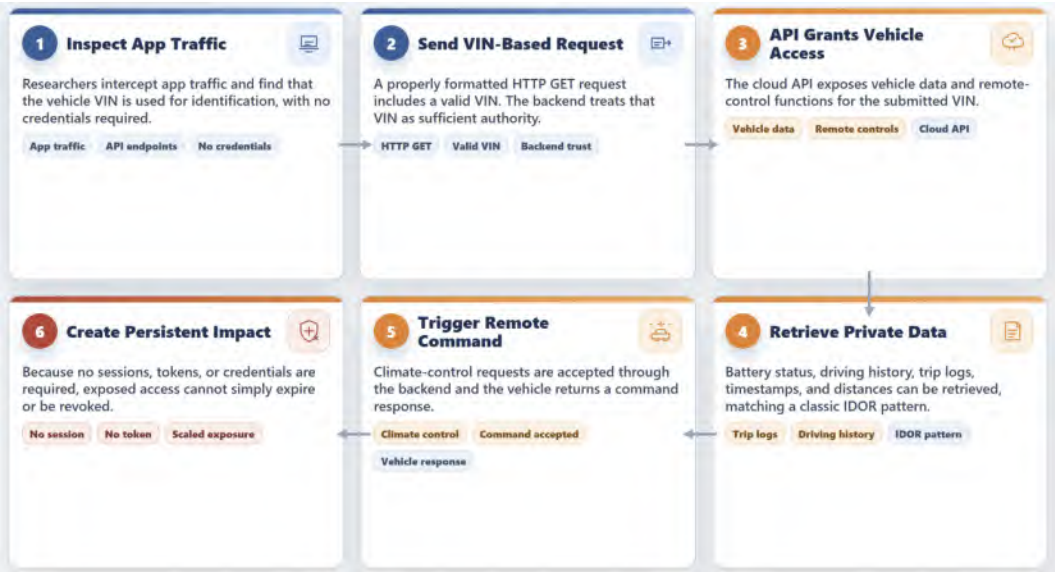


Figure 4.4: Nissan Leaf backend API exploitation flow

Discovery and reconnaissance

Researchers began by analyzing NissanConnect EV App's communication flows. They intercepted the traffic using **Fiddler**, which allowed them to inspect the requests sent from the app to the vehicle backend. During the inspection, they discovered that sensitive operations, like retrieving battery status and driving history or toggling the climate control, required only the VIN as input as a form of authentication. There were no login credentials, session cookies, or OAuth tokens needed to prove legitimate ownership. Since VINs are easily obtainable (visible on dashboards or even found in online photos or listings), attackers could target any Nissan Leaf by feeding that VIN into the NissanConnect EV backend.

Initial access

Initial access did not require breaching any systems, bypassing authentication, or exploiting software vulnerabilities in the traditional sense. The backend APIs were publicly accessible and

functioned exactly as designed, just using the VIN as the identifier or as a form of authentication without any additional security controls.

Thus, any attacker with a web browser, command-line tool, or simple script could send properly formatted HTTP GET requests directly to Nissan's backend servers. Because the API implicitly trusted VIN, an attacker providing a valid one would immediately be granted access to the corresponding vehicle data and control functions. Moreover, the attack was not limited to a single geography; researchers confirmed that regional NissanConnect implementations in locations such as the UK, Norway, and Canada were similarly affected, even though endpoint URLs varied slightly.

Exploitation

After confirming that commands were accepted based on VIN alone, the researchers systematically tested multiple endpoints to see which actions were possible without ever presenting user credentials. The results revealed broad functionality:

- **Battery status queries**: The researchers could obtain current battery charge levels, timestamps of the last update, and other operational details.
- **Climate control activation**: Requests to power on the air conditioning, fans or heated seats were processed immediately. In real-world conditions, repeatedly toggling climate control could drain the battery.
- **Driving history**: The API provided logs of previous trips, exposing timestamps and distances that revealed the owner's usage patterns.

To verify that the vulnerability extended beyond their own vehicle, the researchers used **Burp Suite**'s Intruder feature to iterate requests across a range of VINs against the battery status endpoint, confirming that a valid response was returned from a vehicle they did not own. It is noteworthy that none of these requests included any session token or authentication credential; the VIN alone was sufficient to identify and access any registered vehicle on the backend.

Even though essential driving functions like steering or braking were not impacted, these capabilities were still significant enough to compromise privacy and potentially inconvenience owners by depleting the battery.

Persistence

This attack required no interaction with the vehicle itself but instead exploited an insecure vehicle backend design. Thus, as long as the vehicle backend APIs were exposed, the attack allowed for persistent access. Because all requests, legitimate or malicious, were routed through the same NissanConnect EV cloud infrastructure, the backend had no mechanism to distinguish between an actual vehicle owner and an attacker. When a VIN-based request

arrived, the backend obediently forwarded commands to the target vehicle, resulting in real-world actions such as turning on the heater or retrieving trip data.

Since there were no sessions, tokens, or credentials involved, there was nothing the backend could do to expire or revoke access. As long as the APIs remained unsecured, an attacker could re-issue requests indefinitely using the same or different VINs to maintain control. For example, an attacker could easily script periodic battery checks or repeatedly trigger climate control functions.

Impact

The impact of the vulnerability included both physical vehicle usability and personal privacy. The researchers explained that by repeatedly activating climate control while a vehicle was parked, an attacker could significantly drain the battery, leaving the vehicle inoperable and potentially stranding the owner. While not immediately life-threatening, this represented a real and practical disruption.

More critically, the researchers highlighted how the exposure of driving history enabled profiling of vehicle owners, revealing daily routines, travel patterns, and likely periods when the owner was away from home, resulting in the violation of the owners' privacy.

Ultimately, this attack demonstrated how insecure backend APIs can turn connected-vehicle features into a large-scale surveillance and disruption vector, even without compromising core safety systems.

Case study 2: Remote vehicle takeover via vehicle backend vulnerability

In September 2024, security researchers publicly disclosed a critical vulnerability in Kia's dealer portal infrastructure that enabled remote takeover of any hardware-equipped Kia vehicle using nothing more than a license plate number [52]. The issue had been identified and privately disclosed to Kia earlier in 2024. By chaining together multiple weaknesses, including API misconfigurations, overly permissive dealer privileges, and third-party VIN lookups, the security researchers demonstrated how common web application flaws, when embedded in connected vehicle ecosystems, can escalate into real-world physical security risks at a massive scale.

In simple steps, an attacker could retrieve a vehicle's VIN from a license plate, register a fake dealer account, obtain a privileged session token, demote the legitimate owner, assign themselves as the new primary owner, and gain access to remote vehicle functions. Once assigned as the new owner, the attacker could unlock doors, start or stop engines, and track vehicle locations.

The attack exposed not only vehicle control capabilities but also large volumes of **Personally Identifiable Information** (**PII**), including names, phone numbers, email addresses, and home addresses.

> **Note**
>
> Source: `https://samcurry.net/hacking-kia` [52]

The sequence of attack steps is depicted in *Figure 4.5*.

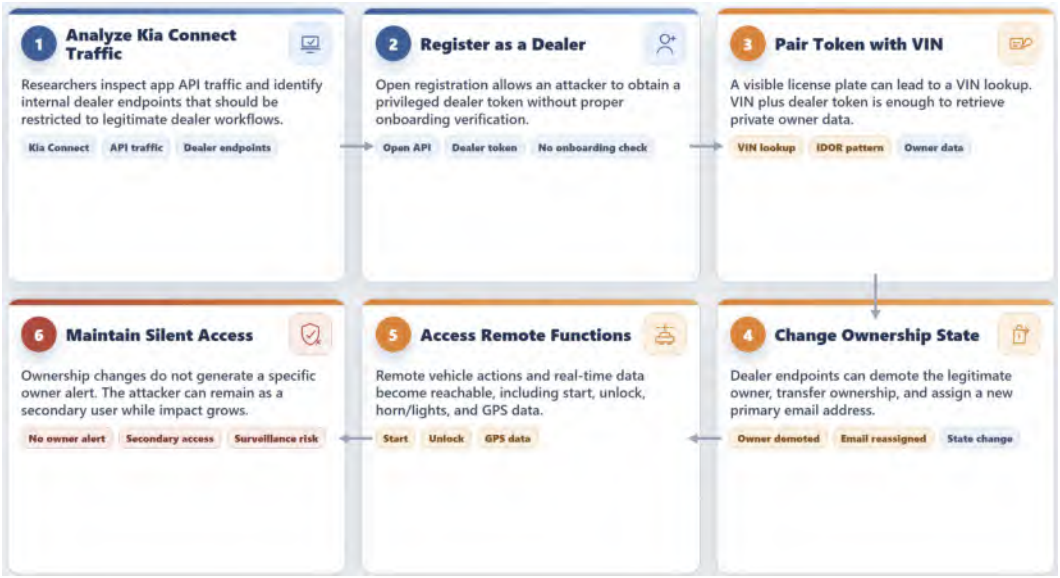


Figure 4.5: Kia Dealer backend API exploitation flow

Discovery and reconnaissance

The researchers initially began by analyzing Kia's standard owner-facing website and mobile application. During their investigation, they observed how newly purchased vehicles were activated and linked to customer accounts. This then shifted their focus toward Kia's dealership infrastructure, specifically the domain `kiaconnect.kdealer.com`, which was used internally by dealerships to register vehicles to customers.

Next, the researchers analyzed the dealer portal's client-side JavaScript and intercepted API traffic using tools like **BurpSuite**, which allowed them to identify internal endpoints and study authentication token behavior. Through this process, they uncovered references to internal "employee-only" API endpoints such as:

- `/dec/dlr/dvl`
- `/dec/dlr/rvp`
- `/ownr/dicve`
- `/prof/authUser`

These endpoints handled dealer vehicle lookup and owner data retrieval (`dvl`), revocation of vehicle permissions (`rvp`), vehicle ownership assignment (`dicve`), and authentication (`authUser`). The researchers found that the backend automatically accepted any request accompanied by a valid dealer token. Thus, this attack was possible due to a fundamental weakness in the trust model rather than exploiting a standalone vulnerability.

Initial access

The initial access breakthrough occurred when the researchers discovered that they could register for a dealer account directly on the dealer portal. That is, the dealer portal accepted registration requests that followed the same HTTP structure used for standard vehicle owner account creation. Critically, there was no onboarding process or verification to confirm that the registrant was an authorized Kia dealership. As a result, the researchers were able to create a fake dealer account and gain access to dealer-level functionality without being a legitimate dealer.

Thus, after registration, the researchers authenticated via the `/prof/authUser` endpoint and received a valid session token. This token was treated by the backend as a legitimate dealer credential and granted access to privileged dealer-level API functionality. Thus, this privileged session token was effectively a master key to gain access to Kia's vehicle backend systems. No additional authentication or, e.g., server-side verification of whether requests originated from the actual dealership networks was performed. Basically, if the token was valid, the backend trusted the request.

Exploitation

Using a valid dealer token and a target vehicle VIN, the researchers were able to proceed with the attack. If an attacker does not know the VIN but is able to read the publicly visible license plate of the target vehicle, they can use a third-party license plate–to–VIN lookup service, such as **PlateToVin.com** [53]. Using such a service, the researchers showed that they could convert a license plate number into the corresponding VIN.

Armed with the target vehicle VIN, the researchers could now proceed with the following steps. First, they queried the `/dec/dlr/dvl` endpoint to retrieve sensitive information about the target vehicle and its registered owner using the VIN. This included the vehicle owner's name, phone number, email address, and sometimes physical address. The system exhibited

classic **Insecure Direct Object Reference** (**IDOR**) behavior. That is, possession of a valid VIN was sufficient to retrieve private owner data, so-called **IDOR with read access**.

Next, the researchers used the `/dec/dlr/rvp` endpoint to demote or remove the legitimate owner's permissions from the vehicle account. The system did not require any confirmation or additional verification from the original owner nor generate any notification about the change, allowing the ownership modification to occur silently.

Finally, through the `/ownr/dicve` endpoint, the researchers were able to assign their own email address as the primary owner of the vehicle and thus become the legitimate new owner of the vehicle according to the system. However, as the original owner would not be able to access their vehicle anymore, they would immediately be able to detect the attack. To avoid detection, the researchers showed that they could also add themselves as a secondary "invisible" user, where the original owner should still have access to the vehicle. It is worth noting that with the possession of a valid VIN, the system also allowed modification of vehicle ownership without proper authorization checks, so called **IDOR with write capability**.

The following is an illustrative example of an API request to transfer vehicle ownership. Note that the actual endpoint paths and parameter names used by the researchers differ from this example:

```
POST /api/vehicle/ownership/transfer HTTP/1.1
Host: api.owners.kia.com
Authorization: Bearer [DEALER_TOKEN]

{

  "vin": "KNDPMCAL3E7137076",

  "new_owner_email": "attacker@example.com"

}
```

Once the vehicle ownership was reassigned to the security researchers, they gained access to Kia Connect remote functionality. They could execute the following actions:

- Lock or unlock doors
- Start or stop the engine
- Honk the horn and flash the headlights
- Access GPS location data
- Potentially access camera functionality in supported models

Persistence

After adding themselves as the primary or secondary owner, the researchers were able to achieve persistent access to vehicle functions, as they would now be the "legitimate" owner of the target vehicle as per the vehicle backend.

As mentioned, the system did not generate any specific alerts or push notifications to legitimate owners when the ownership changed. As a result, the researchers noted that an attacker could silently monitor a vehicle's location or issue remote commands over time without the victim's knowledge. Because dealer tokens granted access to all backend dealer API endpoints, an attacker in possession of a valid token could interact with any registered vehicle in the system without restriction.

Another point to highlight is that poor backend security design also contributed to persistence. That is, dealer accounts were treated as inherently trusted actors within the ecosystem with the dealer token serving more or less as a master key which provided access to privileged backend services. Moreover, there were no additional checks or validation of legitimate vehicle ownership for high-risk requests such as vehicle ownership revocations or assignments.

Impact

The impact of this vulnerability was severe and affected any hardware-equipped Kia vehicle manufactured after 2013, spanning model years 2014 through 2025 across a wide range of models. From a physical security perspective, the researchers demonstrated that they could remotely control critical vehicle functions, including unlocking doors, and starting engines. This leads to concrete risks for vehicle owners including vehicle theft, stalking, or potentially harassment or other forms of physical harm.

From a privacy perspective, the researchers emphasized that attackers could collect large volumes of PII, including vehicle owners' names, phone numbers, email addresses, home addresses, and GPS tracking data. Some models also exposed interfaces that could potentially provide access to camera functionality.

The Kia dealer portal incident illustrates how weak identity validation, over-privileged tokens, IDOR vulnerabilities, and excessive backend trust relationships can allow simple web application flaws to turn into attacks with large-scale impacts.

Case study 3: Remote vehicle control through backend compromise

This case study documents a year-long security research effort targeting Mercedes-Benz's connected vehicle ecosystem, resulting in full remote control of vehicle functions. Unlike the two previous case studies that focused on the backend APIs, this attack required a complete end-to-end analysis starting from vehicle electronics to internal backend services. Security

researchers disclosed these findings at Black Hat USA 2020 [54], covering vulnerabilities across both on-vehicle hardware components and cloud-hosted backend services.

The researchers began with hardware reverse engineering of the "Hermes" TCU. They extracted firmware, uncovered embedded credentials, and gained access to the vehicle's private mobile carrier network. Further, by chaining together approximately 19 distinct vulnerabilities, including hardcoded cryptographic material, weak backend trust assumptions, and a **Server Side Request Forgery** (**SSRF**) flaw in a social media integration plugin, they were able to impersonate trusted internal devices and invoke remote vehicle commands. Ultimately, they demonstrated the ability to unlock doors, control windows and lights, and remotely start engines without physical access, by interacting with the Hermes TCU.

Note

Sources:

- `https://i.blackhat.com/USA-20/Thursday/us-20-Yan-Security-Research-On-Mercedes-Benz-From-Hardware-To-Car-Control.pdf` [54]
- `https://www.youtube.com/watch?v=bBUdMyCP3rY` [54]

The attack flow steps for this case study are shown in *Figure 4.6*.

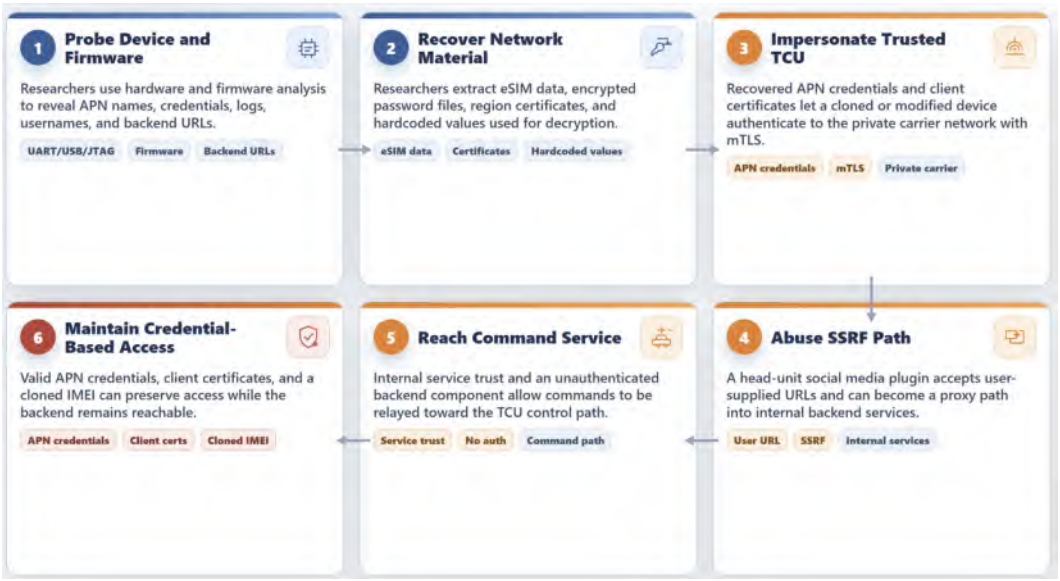


Figure 4.6: Remote vehicle control: backend exploitation flow

Discovery and reconnaissance

The research began with mapping the vehicle's attack surface. Three primary components were identified:

- NTG 5.5 Head Unit (Windows CE 7 Automotive ARM)
- OBD-2 interface
- Hermes TCU (Linux ARM)

While the head unit (infotainment system) and the OBD-2 interface were also examined, the Hermes TCU quickly emerged as the most critical target due to its integrated 4G connectivity, embedded Linux software, and responsibility for handling remote services such as door unlocking and engine start.

To safely conduct their analysis, the researchers constructed a dedicated test bench that replicated the vehicle environment. This included connecting the TCU with a head unit and spoofing heartbeat messages on the CAN bus to avoid triggering anti-theft lockout mechanisms. This controlled lab setup allowed hardware probing and firmware experimentation without permanently bricking the unit.

Next, the security researchers began their analysis of the Hermes TCU by exploring standard embedded debugging interfaces. Through the **Universal Asynchronous Receiver-Transmitter** (**UART**) interface, they accessed system logs that revealed usernames and backend URLs. Using the **Universal Serial Bus** (**USB**) interface, they issued the `AT^SETMODE` command to switch the device into a debug mode that exposed additional communication ports. By changing the USB mode, they were then able to send further AT commands to the application interface, which allowed them to extract sensitive application configurations. These application configurations revealed critical carrier details such as APN names, usernames and passwords to connect to the private vehicle network.

> **Note**
>
> **Terms:**
>
> APNs are unique network identifiers used by cellular devices to connect to private mobile networks rather than the public internet. By assigning a specific APN, a vehicle's telematics unit routes data through a dedicated path to the backend infrastructure. This setup effectively isolates and protects sensitive traffic from broad internet exposure, reducing the attack surface and allowing operators to apply granular security policies, such as IP filtering or firewall rules, at the mobile carrier or private gateway level.

Furthermore, the researchers used tools like **OpenOCD** [55] with an FT2232 adapter to identify the JTAG pins and operate the debug interface. However, while they successfully used the JTAG interface to read or write the 4G module's processor memory, they found it too unstable for a full firmware dump due to lack of the specific NAND driver for the JTAG interface and interference from the system's watchdog.

Note

Terms:

- Universal Asynchronous Receiver-Transmitter (UART) interface is a serial communication interface commonly used in embedded systems to access console output, send commands, and perform debugging over a simple hardware connection.
- Universal Serial Bus (USB) interface is a hardware communication interface used to connect devices for data transfer and power supply, and in embedded systems it is often used for firmware access, configuration exchange, and debugging purposes.
- AT set mode command is a device command that changes the operational state of the hardware to enable additional debugging communication channels accessible via USB.
- Joint Test Action Group (JTAG) interface is a hardware debugging interface that provides low-level access to a device's processor and memory for testing, programming, and firmware analysis.

Therefore, the researchers proceeded with the physical disassembly of the Hermes TCU. Using a **Ball Grid Array** (**BGA**) rework station, they desoldered the NAND flash memory chips from the board. The researchers then built custom **Printed Circuit Board** (**PCB**) adapters that can connect directly to the raw memory, which allowed them to dump the firmware using the tool **Proman NAND reader**. By analyzing the dump, they then reconstructed the partition tables and filesystems, resulting in full access to OEM app binaries, configuration files, encrypted certificates, and scripts.

> **Note**
>
> **Terms:**
>
> **Ball Grid Array** (**BGA**) rework station is a hardware tool used to remove, replace, or reattach integrated circuits mounted with ball-shaped solder connections on printed circuit boards.

However, as the researchers proceeded with reverse-engineering of the app binaries, they encountered a major obstacle. The executables would not run in emulators, and references within the code were missing or wrong. They realized that the firmware dump had a bit-flipping problem, which seemed to stem from a phenomenon in NAND flash that is usually corrected by **Error Correction Code** (**ECC**) in the NAND controller hardware.

During the initial reconstruction of the filesystem, the researchers had skipped ECC and therefore these errors remained in the extracted files. Further, if the researchers tried to write back any modified data to the NAND, e.g., a modified command line to gain a root shell, it would become corrupted because the NAND hardware ECC algorithm would make the "correct data" become "wrong" due to bit-flips. Therefore, the researchers first had to recover the ECC algorithm to be able to extract a correct firmware dump without bit-flips.

They then modified the boot parameters and command-line configurations to be able to obtain an interactive root shell on the device. Because secure boot only checks the boot procedure, the researchers were able to modify the system files freely to install this root shell backdoor. They then flashed the new raw data back to the NAND flash. Using the root shell, the researchers found an engineer mode program with access to the CAN bus, which allowed them to send signals to, e.g., lock or unlock the doors.

Initial access

The researchers further analyzed the Hermes TCU and found that it contained an **embedded Subscriber Identity Module** (**eSIM**). By physically extracting and transplanting the eSIM into a 4G development board/router and configuring it to use the same APN names the researchers extracted before via USB, they were able to replicate the vehicle's cellular identity in their own hardware. To avoid network blocking, they ensured that the **International Mobile Equipment Identity** (**IMEI**) was set to the same as the TCU.

This step allowed the researchers to connect directly to the same private internal carrier network used by legitimate vehicles. From the perspective of the backend, their lab setup now appeared indistinguishable from a real car. However, the carrier network has a security policy in which devices are isolated from each other, so scanning for other connected vehicles is not possible.

Instead, the researchers used the backend URLs extracted before via UART, to explore how they can connect to the vehicle backend. Connecting to the vehicle backend requires HTTPS with mTLS authentication and therefore client certificates are required.

Encrypted certificates were extracted from the firmware dump and further analyzed. The certificates were stored in a **Public-Key Cryptography Standards #12 (PKCS#12)** container called a PFX file, which was password-protected. The password to access the PFX file was stored in the `passwd` file, which in turn was encrypted using the AES-256-CBC algorithm.

> **Note**
>
> **Terms:**
>
> **Public-Key Cryptography Standards #12 (PKCS#12)** is a cryptographic container format used to bundle a private key with its corresponding certificate and related certificates into a single, password-protected file for secure transport and storage.

The researchers found that HERMES will load the PFX file and the `passwd` file during booting, and then the application will load the corresponding region certificate based on the market area. The researchers further found that a system library could decrypt the `passwd` file using the AES-256-CBC algorithm and a hardcoded AES-256 key and **Initialization Vector (IV)** value. So, the researchers used **OpenSSL** [56] and the hardcoded values to decrypt the `passwd` file to get the PFX file password, which allowed them to access the client certificates.

With the extracted APN credentials and decrypted client certificates, they could authenticate backend services as a trusted device. At this stage, they effectively impersonated a legitimate Hermes TCU within the vehicle backend infrastructure.

Exploitation

Once positioned inside the private vehicle infrastructure network, the researchers shifted their focus toward backend vulnerabilities. First, they identified a social media plugin in the head unit as a potential attack vector. This component accepted user-supplied URLs for fetching profile images or data. Due to improper validation of these URLs, this led to a classic **Server-Side Request Forgery (SSRF)** vulnerability. The researchers could craft the URLs in such a way as to force the backend server to initiate HTTP requests to arbitrary destinations such as the server's local files or intranet servers.

Note

Terms:

Server-Side Request Forgery (**SSRF**) is a web application vulnerability where an attacker tricks a server into making network requests on its behalf. Typically, an HTTP endpoint accepts a URL (or similar parameter) from a user or device, and the server then "fetches" that URL internally. If the server does not strictly validate or restrict where it is allowed to connect, an attacker can point the server to internal resources, such as databases, internal REST APIs, or other sensitive systems, which would otherwise never be exposed to the outside internet.

Because these requests originated from the backend infrastructure, they could access internal services that were not directly exposed to the public internet. That is, the SSRF vulnerability effectively turned this vulnerable backend component into a proxy, where an attacker could reach trusted internal systems. As an example, the researchers showed how they could read specific data from the **Telematics Service Provider** (**TSP**) server's local files, e.g., the local `/etc/passwd` file.

Furthermore, using SSRF to pivot laterally, the researchers scanned and interacted with the vehicle backend services. Critically, several of these internal services lacked additional authentication. They implicitly trusted any request originating from the internal network.

Among these services, the researchers identified a critical backend component, namely the **remote vehicle command control** endpoint. This endpoint exposed functionality for commands such as locking/unlocking doors, opening/closing the roof, turning lights on/off, and starting/stopping the engine (on supported vehicles). Triggering this endpoint would then relay the request to the corresponding TCU to perform the actual car action (i.e., locking/unlocking, opening/closing, starting/stopping the engine, etc.).

This attack chain demonstrates a breakdown in the backend trust model, where internal network location was treated as sufficient proof of legitimacy. That is, no mutual authentication or token validation was enforced between the services, which allowed the researchers to trigger internal services once they had gained access to the internal backend infrastructure.

In summary, exploitation required chaining together the following:

- Firmware reverse engineering
- Hardcoded AES-256-CBC key extraction
- Client certificate decryption

- APN credentials discovery
- eSIM transplantation and IMEI replication
- SSRF exploitation in a social media plugin
- Abuse of unauthenticated internal services

Each individual issue might have seemed limited in scope but put together in a full attack chain enabled full remote vehicle command execution.

Persistence

Persistence in this case was achieved by maintaining reliable access to the vehicle backend infrastructure. After extracting the APN credentials, decrypting the client certificates, and transplanting the eSIM into their own 4G hardware, the researchers were able to impersonate a legitimate Hermes TCU on the private carrier network. From the vehicle backend's perspective, the researchers' setup appeared indistinguishable from an authentic TCU.

As long as they retained valid APN credentials, the client certificates for mTLS authentication, and a functioning eSIM with the expected IMEI identity, they could repeatedly reconnect to the vehicle backend. The SSRF vulnerability in the social media plugin then acted as the pivot point. By exploiting it, they could relay requests to backend services that lacked proper authentication and trigger remote command endpoints.

Persistence therefore depended on two conditions: continued access to the private network as a trusted device, and the presence of the SSRF flaw. As long as the backend trust model remained unchanged, they could repeatedly re-establish access and trigger remote vehicle commands. This persistence was architectural rather than device-based, relying on backend design weaknesses rather than implanted malware in the vehicle.

Impact

Successful exploitation of the combined vulnerabilities allowed full remote control of critical vehicle functions without physical access. The researchers demonstrated capabilities such as the following:

- Locking and unlocking doors
- Operating roof and lights
- Remotely starting engines

Furthermore, the researchers stated that the attack had potential impact on an estimated two million connected Mercedes-Benz vehicles in the China market [54]. The security researchers followed the responsible disclosure procedure, and Mercedes-Benz initiated an incident response process. As a result, immediate exposure was contained within 48 hours through

service shutdown and network-level controls, with a full architectural fix introduced over the following weeks.

Having examined how insecure backend design, overly permissive API trust models, and hardcoded credentials can be exploited to achieve remote vehicle control at scale, the following section outlines best practices for securing the vehicle backend infrastructure against these and similar attack vectors.

Best practices for securing the vehicle backend infrastructure

The case studies we sampled here are a small fraction of the published real-life attacks against vehicle backends. Surveying the majority of those cases, we can derive several security best practices that must be considered. For penetration testers, looking for the absence of these controls can be an effective method when trying to identify vulnerabilities and weaknesses in those systems.

Strong authentication and access control

First, it is imperative to ensure that all endpoints and messages are authenticated. Vehicles should authenticate to the cloud using unique credentials (preferably device certificates or unique cryptographic keys) and mutual authentication (e.g., mTLS). Do not rely on security through obscurity. For example, backend APIs should never rely on simple identifiers such as a VIN or email address as sole proof of identity; instead, always enforce user/session authentication and vehicle authentication (e.g., token tied to a user and vehicle). Furthermore, the design of APIs should prioritize security by avoiding the exposure of sensitive operations through public-facing endpoints.

Also, it is important to restrict access rights to the bare minimum necessary to perform the job function following the **principle of least privilege**. Moreover, **Role-Based Access Control** (**RBAC**) should be implemented to ensure only authorized users can access and perform specific actions as defined by their role. In addition, a more explicit trust model, e.g., employing a **Zero-Trust Architecture** (**ZTA**), is required, where each request is validated (e.g., per-resource ownership checks) rather than simply trusting all requests that originate from within the backend infrastructure. In practice, this means using protocols like OAuth 2.0 for user-to-cloud authentication (so that mobile app commands carry a valid access token) and validating that the specific token's user is indeed authorized for the target vehicle to prevent IDOR attacks. For internal admin portals, best practices include using **multi-factor authentication** and IP whitelisting to reduce the risk of compromised credentials being abused and to limit overall access to such portals.

Securing communication

All communication between vehicles and backend must be secured using modern protocols (such as TLS 1.2+). Manufacturers should use appropriate **encryption and authentication for any vehicle- to backend communication**. This includes protecting MQTT topics with TLS and optionally payload encryption for sensitive fields. The vehicle should validate the backend server's certificate (to prevent man-in-the-middle attacks), and the backend should reject any unencrypted or unsigned data from vehicles. In addition, sensitive data at rest stored in the vehicle backend (like personal information or location history) should be encrypted and access controlled. Best practices include avoiding proprietary or weak cryptography; instead, use proven standards and libraries (e.g., use AES for any on-device storage encryption, and TLS/DTLS for transport). Moreover, key management is equally important. It is imperative to regularly rotate and update certificates and cryptographic keys, as well as to establish an expiration and renewal process for vehicle certificates so that a compromised certificate eventually becomes invalid. Additionally, enforce **perfect forward secrecy** on TLS connections to ensure that past communications remain protected even if long-term keys are compromised.

Input validation and secure API development

The backend services must rigorously validate all inputs and commands against expected formats and ranges, especially those coming from vehicles or mobile apps. This helps prevent common web vulnerabilities like SQL injection, command injection, SSRF, or buffer overflows in services that parse vehicle data. In addition, make sure to sanitize inputs such as data used in file paths or system commands on backend servers, and enforce proper authorization checks on every request. In other words, the backend should never assume that just because a message came over the "trusted" channel from a vehicle or app, it is automatically legitimate; i.e., always ***verify the sender's permissions and the message content***. For instance, if a request says, "*unlock vehicle X*," the backend should confirm that the requester is authenticated as the owner of vehicle X (and perhaps add context checks, like rate limiting such commands or detecting anomalies such as a request to unlock a car from an unusual location or time).

Moreover, follow secure development practices for all cloud services and in-vehicle software that parse external input, and use secure coding practices and established security testing approaches. For example, fuzz testing of APIs can help uncover injection or parsing vulnerabilities before deployment. Adhere to OWASP recommendations and perform code reviews with security in mind. Employing strict checks and using frameworks that auto-sanitize inputs (where possible) is key. Additionally, disable any debug interfaces and protect admin functions in production systems as many issues arise from leftover admin APIs that are not meant for public use.

Eliminating default credentials and protecting secrets

Never use universal default passwords or tokens for devices. Each vehicle or device should have a unique secret or certificate provisioned securely, e.g., during manufacturing via a **Hardware Security Module** (**HSM**). If a default credential is unavoidable initially, make sure to force a change on first use. Regularly audit that no development or contractor credentials are embedded in code or configuration files. Apply rigorous credential management using vaults and avoid plaintext keys in source code. Employ automated scanning (e.g., using tools like **GitGuardian** [57]) to catch any credentials before code release.

Also, segment duties so that even if one device's credential is compromised, it cannot be used to access others or escalate privileges on the backend. That is, each vehicle should only have access to its own data/commands. In cloud infrastructure, follow the **principle of least privilege** for service accounts and API keys. This means that a compromise of one backend service should not automatically grant full admin rights to the whole system.

Segmentation and network security

It is imperative to architect the backend with segmentation between critical components. For instance, the interface that communicates with vehicles (IoT Hub, MQTT broker) should be isolated from the business logic and databases via firewalls or **Virtual Private Cloud** (**VPC**) separation. That way, a compromise of the external interface on the backend does not directly expose internal data.

Within the vehicle, similarly, isolate the telematics module from safety-critical CAN networks (the cloud should never directly send commands to control, e.g., braking functionality of the vehicle). Currently, cloud providers offer **security groups** (cloud-native network access control rules) and **Web Application Firewall** (**WAF**) services; use them to filter out malicious requests such as SQL injection attempts or known bad user agents. Finally, **rate limiting** should also be enforced to restrict excessive request volumes and reduce the risk of abuse and **Denial-of-Service** (**DoS**) attacks.

Monitoring, detection, and incident response

Establish a **Vehicle Security Operations Center** (**VSOC**) or integrate vehicle backend logs into existing **Security Operations Center** (**SOC**) monitoring. Every sensitive action (e.g., unlock command, firmware update push, etc.) should generate logs with context (who, what, when, which vehicle) that are monitored for suspicious activity.

In addition, it is suggested to use anomaly detection on both the vehicle side (via an **Intrusion Detection System** (**IDS**) agent in the TCU, if possible) and backend side (via AI-driven API monitoring services) to catch deviations like an account issuing commands to many vehicles or

a vehicle sending malformed data potentially indicating compromise. That is, best practices include continuously monitoring API usage for unusual activity.

Protecting sensitive data and privacy

The backend often stores personal data (user profiles, vehicle location history, etc.) and secrets (encryption keys, private keys associated with certificates, etc.). This data must be protected at rest and in use. Use database encryption or at least encrypt sensitive fields. Passwords should be hashed and salted, while private keys should be stored in HSM-backed secure key vaults rather than in a database. Implement strict access controls in databases so that even if an SQL injection is attempted, the attacker cannot easily dump everything.

Monitor large or unusual data queries and trigger alerts for such anomalous activities with appropriate responses. If an attacker does breach the backend, having data encrypted can mitigate the damage (attackers cannot easily use what they steal). **Privacy-by-design principles** should be followed: only collect data you need, anonymize or pseudonymize where possible, and adhere to relevant regulations such as the **General Data Protection Regulation** (**GDPR**) [58]. The goal is to ensure that even if an attack occurs, the **impact on users' personal data is minimized**.

Supply chain and third-party management

Conduct security assessments of any third-party telematics or cloud service integrated into the vehicle backend. OEMs should require strong security measures contractually. For example, if using a third-party service from a telematics partner, insist on regular penetration tests and vulnerability disclosures. Keep an eye on vulnerabilities in open-source components used in the backend (e.g., the log4j vulnerability [59] affected millions of servers) and ensure you have a process to quickly patch exploitable vulnerabilities in cloud systems just as you would in vehicles.

Defense-in-depth principle

By following the above-mentioned best practices, automotive companies can significantly reduce the attack surface of cloud backends. As a general approach, the **defense-in-depth principle** should be applied when designing cybersecurity for the vehicle backend. Thus, a secure vehicle backend architecture would have multiple gates an attacker must get through: even if one protective layer falters (say, an unknown input validation bug), other controls (authentication, monitoring) can mitigate the damage. The goal is to prevent the kinds of trivial exploits seen in our case studies. Engineers should internalize lessons such as "*never trust data from the client (vehicle or app) without verification*" and "*assume that any exposed service will be found and probed by attackers.*"

Summary

In recent years, many prominent vehicle security breaches have demonstrated that the cloud backend and associated APIs are an attractive target for attackers aiming to establish a foothold in the vehicle. In this chapter, we examined how security weaknesses common in IoT systems, such as broken authentication, poor encryption, and vulnerable APIs, also manifest in automotive backend architectures. Through an overview of typical connected vehicle backend designs on Azure and AWS, we saw that vehicles are essentially IoT devices on wheels, relying on cloud services to extend their functionality.

We then surveyed how design and implementation flaws can creep in, from trusting user-supplied identifiers like VIN, to leaving an admin portal exposed, to shipping devices with default passwords. Real-world case studies confirmed that these weaknesses are not just theoretical. In each case, attackers (in this instance, researchers) were able to exploit the same kinds of issues that plague general IT systems, resulting in the ability to locate, unlock, or even disable vehicles remotely.

For penetration testers, probing companion mobile applications and web services is a must when testing for backend-related security vulnerabilities. For defensive security engineers, the weaknesses presented, and the corresponding security best practices, should provide plenty of useful takeaways to build more resilient backend services that can effectively reduce the vehicle attack surface. With the backend fortified, both the drivers on the road and the engineers behind the scenes can have greater confidence in the overall security of modern vehicles.

Now that we have covered the vehicle backend, it is time to take a step deeper into the vehicle E/E architecture and examine wireless vehicle systems in the next chapter.

References

[1] Microsoft Azure Architecture Center, "*Automotive connected fleets – Reference Architecture*," Microsoft Docs, 2022. [Online]. Available: `https://learn.microsoft.com/en-us/industry/mobility/architecture/automotive-connected-fleets-content`

[2] A. Givens and J. L. Snow, "*Building and Modernizing Connected Vehicle Platforms with AWS IoT*," AWS IoT Blog, Jun. 2023. [Online]. Available: `https://aws.amazon.com/blogs/iot/modernizing-connected-vehicle-platforms-with-aws-iot/`

[3] Microsoft Azure Blog, "*Microsoft Connected Vehicle Platform: trends and investment areas*," 2020. [Online]. Available: `https://azure.microsoft.com/en-us/blog/microsoft-connected-vehicle-platform-trends-and-investment-areas/`

[4] Azure Architecture Center, "*Automotive messaging, data, and analytics – Architecture,*" Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/industry/mobility/architecture/automotive-messaging-data-analytics-content

[5] AWS, "*AWS Connected Vehicle Solution – Reference Architecture Diagram.*" [Online]. Available: https://docs.aws.amazon.com/pdfs/architecture-diagrams/latest/aws-connected-vehicle/aws-connected-vehicle.pdf

[6] ISO, "*ISO 13400-2:2019 Road vehicles — Diagnostic communication over Internet Protocol (DoIP) — Part 2: Transport protocol and network layer services*," ISO, Geneva, Switzerland, 2019. [Online]. Available: https://www.iso.org/standard/74785.html

[7] ISO, "*ISO 17978-1 Road vehicles — Service-oriented vehicle diagnostics (SOVD) — Part 1: General information, definitions, rules, and basic principles*," ISO, Geneva, Switzerland. [Online]. Available: https://www.iso.org/standard/85133.html

[8] HiveMQ, "*MQTT Packets: A Comprehensive Guide*," HiveMQ Blog. [Online]. Available: https://www.hivemq.com/blog/mqtt-packets-comprehensive-guide/

[9] S. Lakshminarayana, A. Praseed, and P. S. Thilagam, "*Securing the IoT Application Layer From an MQTT Protocol Perspective: Challenges and Research Prospects*," IEEE Communications Surveys and Tutorials, vol. 26, no. 4, pp. 2510-2546, 2024. [Online]. Available: https://doi.org/10.1109/COMST.2024.3372630

[10] D. K. Oka, S. Nadahalli, J. Yost, and R. P. Bojanki, "*Building Secure Automotive IoT Applications: Developing Robust IoT Solutions for Next-Gen Automotive Software*," Packt Publishing, Birmingham, UK, 2024, ISBN: 978-1835465509.

[11] OWASP, "*OWASP API Security Top 10*," OWASP Foundation. [Online]. Available: https://owasp.org/www-project-api-security/

[12] MITRE, "*Common Weakness Enumeration (CWE)*," MITRE. [Online]. Available: https://cwe.mitre.org/

[13] D. K. Oka and R. S. Vadamalu, "*Considerations for Vulnerability Management in the Automotive Industry*," SAE Technical Paper 2026-26-0626, 2026. [Online]. Available: https://doi.org/10.4271/2026-26-0626`

[14] Upstream Security, "*2023 Global Automotive Cybersecurity Report*," Upstream Security, 2023. [Online]. Available: https://upstream.auto/reports/2023report/

[15] S. Curry et al., "*Web Hackers vs. The Auto Industry: Critical Vulnerabilities in Ferrari, BMW, Rolls Royce, Porsche, and More,*" samcurry.net, Jan. 2023. [Online]. Available: https://samcurry.net/web-hackers-vs-the-auto-industry

[16] T. Hunt, "*Controlling vehicle features of Nissan LEAFs across the globe via vulnerable APIs,*" TroyHunt.com, Feb. 2016. [Online]. Available: https://www.troyhunt.com/controlling-vehicle-features-of-nissan/

[17] P. Umbelino, "*Critical Vulnerabilities Discovered in Popular Automotive GPS Tracking Device (MiCODUS MV720)*," BitSight Research, Jul. 19, 2022. [Online]. Available: https://www.bitsight.com/resources/critical-vulnerabilities-discovered-popular-automotive-gps-tracking-device-micodus-mv720

[18] J. Coker, "*Toyota Reveals Data Leak of 300,000 Customers*," Infosecurity Magazine, Oct. 11, 2022. [Online]. Available: https://www.infosecurity-magazine.com/news/toyota-data-leak-customers/

[19] E. Kovacs, "*Several Car Brands Exposed to Hacking by Flaw in Sirius XM Connected Vehicle Service*," SecurityWeek, Dec. 2022. [Online]. Available: https://www.securityweek.com/several-car-brands-exposed-hacking-flaw-sirius-xm-connected-vehicle-service

[20] PortSwigger, "*Burp Suite – Web Security Testing Software.*" [Online]. Available: https://portswigger.net/burp

[21] Postman Inc., "*Postman API Platform*." [Online]. Available: https://www.postman.com/

[22] Frida, "*Frida – Dynamic Instrumentation Toolkit*." [Online]. Available:https://frida.re/

[23] Shodan, "*Shodan – Search Engine for Internet-Connected Devices*." [Online]. Available: https://www.shodan.io/

[24] OWASP, "*OWASP ZAP – Zed Attack Proxy,*" OWASP Foundation. [Online]. Available: https://www.zaproxy.org/

[25] J. Pereyda, "*boofuzz: Network Protocol Fuzzing for Humans*," GitHub. [Online]. Available: https://github.com/jtpereyda/boofuzz

[26] GitLab, "*Protocol Fuzzer Community Edition (Peach Fuzzer CE),*" GitLab. [Online]. Available: https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce

[27] Telerik, "*Fiddler – Web Debugging Proxy*." [Online]. Available: https://www.telerik.com/fiddler

[28] ticarpi, "*jwt_tool: A Toolkit for Testing, Tweaking and Cracking JSON Web Tokens*," GitHub. [Online]. Available: https://github.com/ticarpi/jwt_tool

[29] MobSF, "*Mobile Security Framework (MobSF)*," GitHub. [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF

[30] topjohnwu, "*Magisk: The Magic Mask for Android*," GitHub. [Online]. Available: https://github.com/topjohnwu/Magisk

[31] skylot, "*jadx: Dex to Java Decompiler*," GitHub. [Online]. Available: https://github.com/skylot/jadx

[32] iBotPeaches, "*Apktool: A Tool for Reverse Engineering Android APK Files*," GitHub. [Online]. Available: https://github.com/iBotPeaches/Apktool

[33] ReFirmLabs, "*binwalk: Firmware Analysis Tool*," GitHub. [Online]. Available: https://github.com/ReFirmLabs/binwalk

[34] National Security Agency, "*Ghidra: Software Reverse Engineering Framework*," GitHub. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[35] Hex-Rays, "*IDA Pro – Interactive Disassembler*." [Online]. Available: https://hex-rays.com/ida-pro/

[36] Truffle Security Co., "*TruffleHog: Find, Verify, and Analyze Leaked Credentials*," GitHub. [Online]. Available: https://github.com/trufflesecurity/trufflehog

[37] Gitleaks, "*Gitleaks: Find Secrets with Gitleaks*," GitHub. [Online]. Available: https://github.com/gitleaks/gitleaks

[38] Wireshark Foundation, "*Wireshark – Network Protocol Analyzer*." [Online]. Available: https://www.wireshark.org/

[39] TCPDUMP Group, "*tcpdump – Packet Analyzer*." [Online]. Available: https://www.tcpdump.org/

[40] nabla-c0d3, "*SSLyze: Fast and Powerful SSL/TLS Scanning Library*," GitHub. [Online]. Available: https://github.com/nabla-c0d3/sslyze

[41] Nmap Project, "*Nmap – Network Mapper.*" [Online]. Available: https://nmap.org/

[42] mitmproxy contributors, "*mitmproxy: An Interactive TLS-capable Intercepting HTTP Proxy*," GitHub. [Online]. Available: https://github.com/mitmproxy/mitmproxy

[43] T. Nordquist, "*MQTT Explorer: An All-round MQTT Client*," GitHub. [Online]. Available: https://github.com/thomasnordquist/MQTT-Explorer

[44] Eclipse Foundation, "*Eclipse Mosquitto – An Open Source MQTT Broker*." [Online]. Available: https://mosquitto.org/

[45] Linklayer Labs, "*CANtact – USB CAN Bus Interface*." [Online]. Available: https://cantact.io/

[46] PEAK-System Technik GmbH, "PCAN-USB – CAN Interface for USB." [Online]. Available: https://www.peak-system.com/products/hardware/external-pc-interfaces/pcan-usb/

[47] Censys, "*Censys Internet Intelligence Platform*." [Online]. Available: https://censys.com/

[48] SecurityTrails, "*SecurityTrails – DNS Intelligence and Threat Hunting*." [Online]. Available: https://securitytrails.com/

[49] OWASP Amass, "*OWASP Amass: In-depth Attack Surface Mapping and Asset Discovery*," GitHub. [Online]. Available: https://github.com/owasp-amass/amass

[50] OWASP, "*OWASP Dependency-Check – Software Composition Analysis,*" OWASP Foundation. [Online]. Available: https://owasp.org/www-project-dependency-check/

[51] Aqua Security, "*Trivy: Find Vulnerabilities, Misconfigurations, Secrets and SBOM in Containers*," GitHub. [Online]. Available: https://github.com/aquasecurity/trivy

[52] S. Curry, N. Rivera, J. Rhinehart, and I. Carroll, "*Hacking Kia: Remotely Controlling Cars With Just a License Plate*," samcurry.net, Sep. 2024. [Online]. Available: https://samcurry.net/hacking-kia

[53] PlateToVIN, "*PlateToVIN: Convert US License Plate to VIN Numbers via API*." [Online]. Available: https://platetovin.com/

[54] M. Yan, J. Li, and G. Harpak, "*Security Research on Mercedes-Benz: From Hardware to Car Control*," Black Hat USA 2020, Aug. 2020. [Online]. Available: https://i.blackhat.com/USA-20/Thursday/us-20-Yan-Security-Research-On-Mercedes-Benz-From-Hardware-To-Car-Control.pdf, https://www.youtube.com/watch?v=bBUdMyCP3rY

[55] OpenOCD, "*Open On-Chip Debugger*." [Online]. Available: https://openocd.org/

[56] OpenSSL Software Foundation, "*OpenSSL Cryptography and SSL/TLS Toolkit*." [Online]. Available: https://www.openssl.org/

[57] GitGuardian, "*Secrets Security and NHI Governance*." [Online]. Available: https://www.gitguardian.com/

[58] European Parliament and Council, "*Regulation (EU) 2016/679 (General Data Protection Regulation),*" Official Journal of the European Union, Apr. 2016. [Online]. Available: `https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng`

[59] NIST, "*CVE-2021-44228 Detail – Log4Shell (Apache Log4j),*" National Vulnerability Database. [Online]. Available: `https://nvd.nist.gov/vuln/detail/cve-2021-44228`

[60] X. René-Corail, "*Our plan for a more secure npm supply chain,*" GitHub Blog, Sep. 22, 2025. [Online]. Available: `https://github.blog/security/supply-chain-security/our-plan-for-a-more-secure-npm-supply-chain/`

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

5

Attacking the Wireless Vehicle Systems

There is no doubt that the holy grail of automotive attacks are the ones that can be launched remotely and can cause major consequences, including safety and financial impact as well as disruption of operation. A fundamental enabler of remote attacks is the compromise of one or more of the vehicle's wireless networking interfaces. Understandably, **original equipment manufacturers** (**OEMs**) are most worried about the possibility of remote attacks since these are the ones with the greatest potential of scaling across many vehicles.

The upward trend of expanding connectivity services in vehicles spanning Cellular, Wi-Fi, Bluetooth, **Near-Field Communication** (**NFC**), and regular **Radio Frequency** (**RF**) channels is further expanding the prospect of such attacks. Protecting the vehicle's wireless protocols is a critical piece in any multi-layer defense strategy; therefore, the focus of this chapter is on understanding the common weaknesses and vulnerabilities that these protocols are subjected to.

We will start by surveying the various wireless connectivity protocols that a typical vehicle supports then explore the common weaknesses and vulnerabilities that such protocols exhibit. Then we will explore the various tools and techniques that can be leveraged to compromise such protocols with the objective of establishing a foothold within the vehicle. While doing so, we will point to several real-life attack examples and provide best practices to eliminate unnecessary connectivity interfaces and make the ones that are mandatory as resilient to cyber-attacks as possible.

In the rest of this chapter, we will cover the following main topics:

- Overview of wireless vehicle systems
- Understanding wireless vehicle systems protocols
- Common weaknesses in wireless vehicle systems
- Penetration testing techniques for wireless vehicle systems
- Case studies of real-life attacks on wireless vehicle systems
- Best practices for securing wireless vehicle systems

Overview of wireless vehicle systems

There was a time when the words "wireless" and "vehicle" were not normally mentioned in the same sentence. However, in recent years, the number of wireless systems integrated into vehicles has seen a sharp increase. From wireless **Tire Pressure Monitoring Systems (TPMS)**, **Ultra-Wideband (UWB)**, and **Vehicle-to-Everything (V2X)** communication to more common Wi-Fi, Bluetooth and **Global Navigation Satellite System** (**GNSS**), the range of vehicle wireless interfaces is greater than most people expect. In fact, wireless connectivity has become a cornerstone of modern vehicles, enabling everything from remote keyless entry to vehicle summons features in which vehicle owners can retrieve their cars from a parking structure using nothing but their mobile phones.

As we have seen in the previous chapter, connected vehicles utilize a variety of wireless interfaces to communicate with internal and external systems. To support various use cases, wireless communication goes from external-facing systems on vehicles such as a telematics unit or an in-vehicle infotainment system to internal in-vehicle network systems such as gateways and **Electronic Control Units (ECUs)**.

Although these wireless interfaces provide convenience and functionality, they also create numerous entry points for attackers, thus expanding the attack surface. Wireless attacks target the various wireless interfaces used by vehicles, including Wi-Fi, Bluetooth, cellular networks, and other RF communications. In many cases, attacking these interfaces serves as an initial access tactic allowing an adversary to establish a foothold within one of the vehicle's electronic components before mounting more advanced attacks to achieve their objective, as illustrated in *Figure 5.1*:

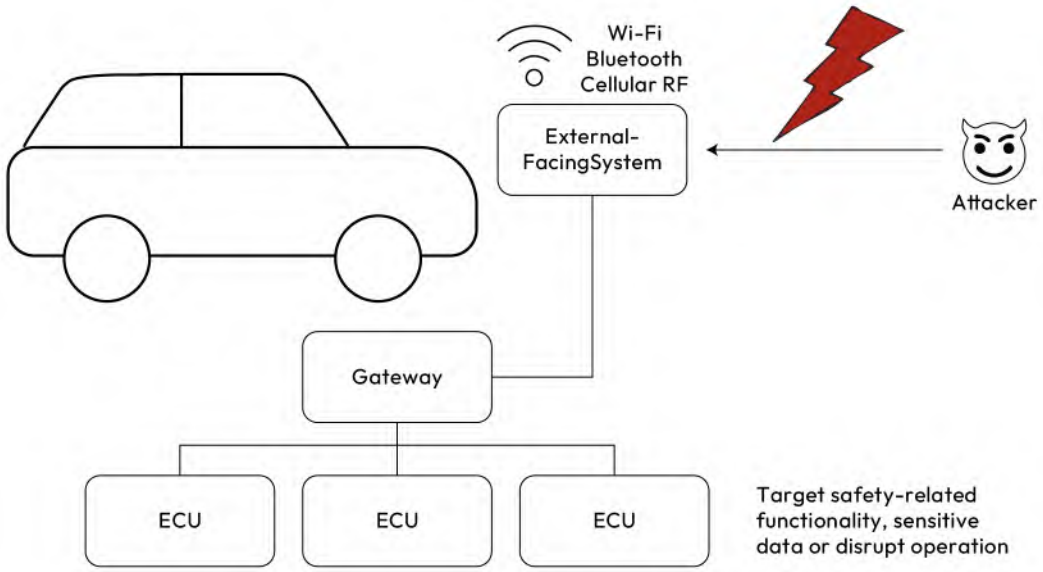


Figure 5.1: Example flow of attack: from wireless access to targeting in-vehicle ECUs

These wireless attacks on connected vehicles can have far-reaching consequences, both in terms of **data privacy**, **financial damage**, **operations**, and **vehicle safety**. Compromising a vehicle's wireless communication systems may expose sensitive information such as location data, user credentials, or proprietary vehicle diagnostics. In addition, attacks targeting systems such as keyless entry could allow attackers to steal vehicles. Worse, successful wireless attacks could allow an attacker to gain remote control of critical systems, such as brakes, steering, or powertrain, disabling such systems to prevent normal vehicle operation or potentially even disrupting the systems, leading to life-threatening scenarios.

Recent security research has shown that the wireless attack surface is a primary target for both vehicle theft and for establishing a foothold into the vehicle internal network when constructing multi-chain attacks. In the case studies section in this chapter, we will get a closer look at some of these real-life attacks, but let's first survey the various wireless attack surfaces relevant to the modern connected vehicle.

Understanding wireless vehicle systems protocols

In this section, we introduce the common wireless interfaces and protocols used in modern vehicle systems. While these wireless protocols enable various valuable features that improve user experience, they also expand the attack surface serving as rich entry points for remote attackers. Understanding what each interface is used for, where it fits in the vehicle

architecture, and which higher-level services typically run over it is a prerequisite to understanding how these interfaces become vulnerable to attack.

An overview of common wireless interfaces used in vehicle systems is given in *Table 5.1*, followed by a brief discussion of each protocol family:

Interface	Access range	Frequency	Technology	Applications	Security relevance
Wi-Fi	Up to 100 m	2.4 GHz, 5 GHz, 6 GHz	IEEE 802.11a/b/g/n/ac/ax	updates, Infotainment	Externally reachable IP interface; expands attack surface; requires strong authentication, encryption, and network segmentation.
Bluetooth	~10 m	2.4 GHz	Bluetooth Classic, BLE	Hands-free calling, audio streaming, sensors, digital key support	Susceptible to pairing abuse, unauthorized GATT access, and relay attacks; security depends on proper bonding and access control.

Interface	Access range	Frequency	Technology	Applications	Security relevance
Cellular	Up to several kilometers (macro cells)	Various bands (700 MHz–3.5 GHz, region-dependent)	GSM, UMTS, LTE, 5G NR	Telematics, emergency calls, OTA updates	Primary backend interface; relies on TLS, mTLS, and token-based authorization; misconfiguration can enable large-scale remote attacks.
RF	Varies (10 m–30 m)	Various (Sub-GHz and ISM bands (e.g., 315/433/868/915 MHz))	Proprietary RF protocols in ISM bands	Key fobs, TPMS	Historically exploited for replay and relay attacks; limited cryptography and proprietary designs pose a high theft risk.
UWB	Varies (10 m–30 m)	3.1–10.6 GHz range	IEEE 802.15.4z	Digital Key, secure ranging	Provides strong protection against relay attacks via distance measurement; risks include ranging manipulation and multi-radio integration flaws.

Interface	Access range	Frequency	Technology	Applications	Security relevance
NFC	~4 cm	13.56 MHz	ISO/IEC 14443, ISO/IEC 18092	Vehicle unlock, backup access	Short range reduces exposure, but cloning and fallback misuse can bypass stronger access mechanisms if not controlled
GNSS	Global (kilometers)	L1 (1.575 GHz), L2 (1.227 GHz), L5 (1.176 GHz)	GPS, GLONASS, Galileo, BeiDou, NavIC, QZSS	Navigation, tracking	Vulnerable to spoofing and jamming; impacts location-based features and can affect safety and forensic integrity.
V2X	Up to 1 km	5.9 GHz (DSRC/C-V2X)	IEEE 802.11p, LTE-V2X	Cooperative driving, safety messaging	Safety-critical interface; requires strong message authentication and misbehavior detection to prevent spoofing or false alerts.

Table 5.1: Overview of Wireless Interfaces

Wi-Fi

Wi-Fi hotspots provide internet access for passengers and are often connected to vehicles through one of the high-performance ECUs. Beyond OEM-provided Wi-Fi access, aftermarket

Wi-Fi dongles may also be connected through the **On-Board Diagnostics (OBD)** port to provide connectivity or telemetry features. In addition to hotspots, individual ECUs may offer Wi-Fi connectivity to communicate with a user's mobile phone, for instance during remote parking or service workflows. Some in-vehicle systems also rely on dedicated Wi-Fi links for internal communication rather than traditional wired in-vehicle networks. As a result, Wi-Fi can appear in the vehicle both as an external-facing interface and as an internal transport mechanism.

Vehicle Wi-Fi dongles have varying levels of security assurance, making them a desirable target for adversaries. Since they are already plugged into the OBD port, an exploitable vulnerability in such a dongle places the attacker directly on the vehicle internal network, which is quite useful if the attacker's objective is to reach embedded ECUs or sensors on the **Controller Area Network** (**CAN**) bus.

Examples of protocols running on top of Wi-Fi include **Transport Layer Security** (**TLS**), **Hypertext Transfer Protocol (Secure)** (**HTTP(S)**), WebSockets, **Message Queuing Telemetry Transport** (**MQTT**), **Diagnostics over IP** (**DoIP**), and **Service-Oriented Vehicle Diagnostics** (**SOVD**), which are also briefly introduced in *Chapter 4*.

Bluetooth

Bluetooth is widely used in vehicles to connect smartphones and other nearby devices for hands-free communication, media streaming, data synchronization, and short-range service interaction. In automotive systems, Bluetooth may support both user-facing features and background communication between vehicle and companion devices. Bluetooth Classic is commonly used for audio and phone integration, while **Bluetooth Low Energy (BLE)** is often used for lower-power, intermittent exchanges such as status updates, sensor data, and digital key discovery.

Attackers frequently abuse vulnerabilities or insecure configurations within the Bluetooth software stack to inject their own code and data before finding another way to escalate privileges and perform even more intrusive attacks against other vehicle components.

Examples of common Bluetooth profiles for automotive use cases include **Hands-Free Profile** (**HFP**), **Advanced Audio Distribution Profile** (**A2DP**), and **Phone Book Access Profile** (**PBAP**). Moreover, **BLE**, which is a low-power wireless technology designed for short-range, intermittent data exchange, is used for such applications as **Generic Attribute Profile** (**GATT**) for vehicle status and sensor data. In addition, the **Car Connectivity Consortium** (**CCC**) Digital Key protocol uses a combination of BLE, NFC and UWB to improve security for vehicle access.

Cellular

Cellular connectivity is one of the primary mechanisms that link the vehicle to external cloud services. It is commonly used for telematics, emergency calling, fleet connectivity, remote diagnostics, software updates, and mobile app features such as remote unlock or climate control. In most architectures, this connectivity is provided through a telematics or connectivity control unit that acts as the vehicle's main long-range communication path.

In recent years, telematics units have shifted to **embedded Subscriber Identity Module (eSIM)** support, which is a digital version of the traditional SIM card used in mobile devices to authenticate and connect to cellular networks. Unlike a physical SIM card, the eSIM is embedded directly into the cellular modem circuitry and supports 4G/5G networks [1]. Critical flaws in SIM/eSIM design have been shown to enable numerous attacks such as impersonation, man-in-the-middle and traffic eavesdropping. Protocols running over cellular networks are often similar to those used over Wi-Fi, including TLS, HTTP(S), WebSockets, MQTT, DoIP, and SOVD.

Radio frequency (RF)

Remote Keyless Entry (RKE) systems use RF signals to wirelessly lock and unlock vehicle doors, while **Passive Keyless Entry and Start (PKES)** systems rely on proximity-based communication to support hands-free access and engine start. Immobilizer systems use low-frequency RF communication as part of the vehicle's start authorization process, and Tire Pressure Monitoring Systems (TPMS) use RF sensors to transmit tire pressure data to the vehicle. Many of these automotive RF systems operate in license-free **Industrial, Scientific, and Medical** (**ISM**) bands and often rely on vendor-specific protocols rather than broadly standardized stacks. RF channels are frequently exploited for vehicle theft as well as establishing initial access into ECUs that support RF based communication.

Ultra-wideband (UWB)

UWB uses short-duration, wide-band RF pulses to enable highly accurate distance and direction measurement between the vehicle and an external device. In automotive systems, UWB is primarily used for secure vehicle access, most notably in modern digital key implementations where precise ranging is required to prevent relay attacks. Unlike traditional RF-based keyless entry, UWB allows the vehicle to verify that the key is physically near the vehicle and in a specific location relative to it. UWB is commonly combined with BLE and NFC for device discovery, authentication, and fallback access mechanisms. From a cybersecurity perspective, while UWB significantly raises the bar against relay-based theft, it remains susceptible to attacks such as distance-measurement manipulation, protocol implementation flaws, key provisioning abuse, and weaknesses in multi-radio handover logic.

Near field communication (NFC)

NFC has been gaining popularity in automotive systems because it enables simple, short-range interactions between the vehicle and an external device. Common use cases include device pairing, credential provisioning, and digital key workflows such as tap-to-unlock or fallback access. Because NFC requires close physical proximity, it is often used for intentional user-present actions rather than continuous wireless communication. In many digital key architectures, NFC works alongside BLE and UWB as part of a broader multi-radio access design. NFC systems face several potential attacks, such as eavesdropping, where an adversary intercepts communication using an antenna beyond NFC's typical 10 cm range. It is also susceptible to relay attacks that leverage multiple devices to extend the NFC range, enabling unauthorized transactions.

Global navigation satellite system (GNSS)

GNSS is used in automotive systems for navigation and, increasingly, for localization and time synchronization. In modern vehicles, GNSS data may be consumed not only by infotainment navigation systems but also by telematics, fleet services, and advanced driver assistance or autonomous driving functions. Although GNSS is often thought of as a navigation feature, it can also serve as a supporting input to other vehicle systems that depend on accurate position or timing information.

GNSS systems are vulnerable to various attacks that exploit the low power and open nature of their signals. Key threats include jamming, which involves overpowering GNSS signals with noise, causing service disruption such as **Denial of Service (DoS)** attacks, and spoofing, where adversaries transmit fake signals to mislead receivers, causing a shift in both time and location. Since many vehicles rely on GNSS to synchronize internal ECUs to a common time source, spoofing can have serious impacts on vehicle safety if the time source becomes unreliable.

Vehicle-to-everything communication (V2X)

The purpose of **V2X communication** is to enhance road safety, traffic efficiency, and driving convenience by enabling vehicles to exchange information with other vehicles, roadside infrastructure, pedestrians, and network services. This includes **Vehicle-to-Vehicle** (**V2V**), **Vehicle-to-Infrastructure** (**V2I**), **Vehicle-to-Pedestrians** (**V2P**), and **Vehicle-to-Networks** (**V2N**) communication. V2X systems are designed to support low-latency exchange of traffic, hazard, and situational awareness data, making them especially relevant to safety-oriented and cooperative driving functions.

Naturally, cybersecurity has been a concern from the inception of V2X protocols due to the potential for invasion of user privacy as well as the potential for spoofing of **Basic Safety Messages (BSMs)**, which can have hazardous consequences. Passive attacks against V2X involve eavesdropping on communication channels to collect sensitive data like vehicle

location and behavior, threatening user privacy, and enabling tracking. Active attacks, on the other hand, involve injecting false messages, spoofing identities, or disrupting communications, potentially causing accidents, traffic congestion, or system malfunctions.

Taken together, these wireless interfaces support a wide range of vehicle functions, from convenience features, and telematics to access control and cooperative driving. With this protocol landscape in mind, we can now examine the recurring weaknesses that tend to appear across these wireless technologies and the trust assumptions they introduce into the vehicle architecture.

Common weaknesses in wireless vehicle systems

As we have seen earlier, wireless connectivity has become a defining characteristic of modern vehicles. Features such as remote services, over-the-air updates, smartphone integration, and cooperative driving systems all depend on continuous communication with external networks. While these capabilities deliver significant functional and commercial benefits, they also expand the vehicle's exposure to remote attacks in ways that were largely absent from earlier automotive architectures.

This section examines common weaknesses found in wireless vehicle systems. Rather than focusing on individual vulnerabilities, we analyze recurring patterns that emerge across technologies, software, and systems. These weaknesses stem from architectural assumptions, legacy design choices, and the challenges of integrating consumer-grade wireless technologies into safety-critical environments. The goal of this section is to provide a clear understanding of how and why wireless vehicle systems can be attacked, setting the stage for practical penetration testing techniques in the next section.

Figure 5.2 provides an overview of common weaknesses found in wireless vehicle systems:



Figure 5.2: Common weaknesses in wireless vehicle systems

Expanding wireless attack surface

Perhaps the first and most fundamental weakness of vehicle wireless interfaces is that they expand the attack surface of the vehicle. While removing wireless access is no longer an option, at the very minimum reducing the attack surface to the minimum access possible is still desirable.

An overview of the expanding wireless attack surface is shown in *Figure 5.3*:

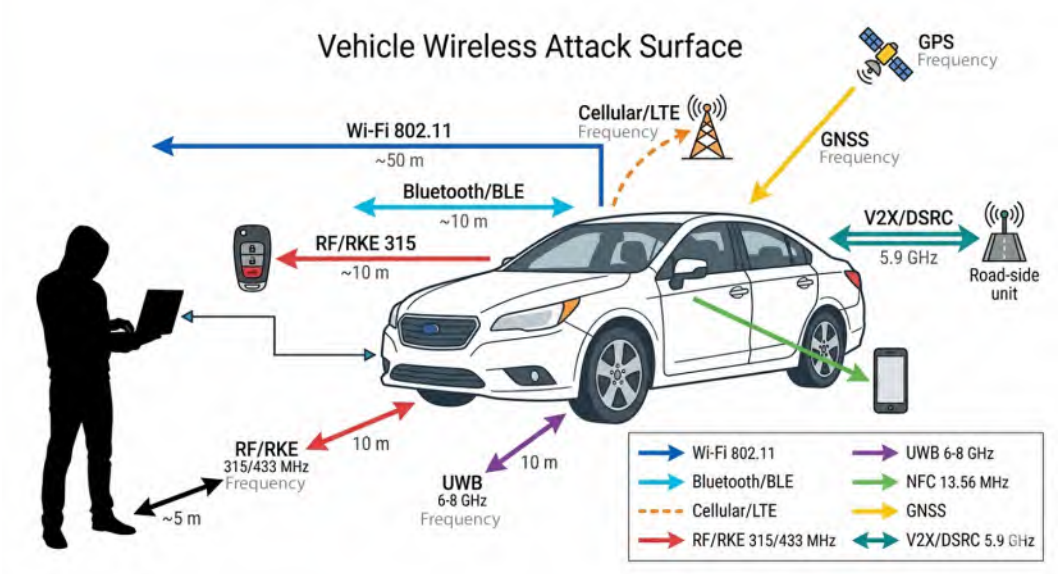


Figure 5.3: Expanding wireless attack surface on modern vehicles

Across Wi-Fi, Bluetooth, cellular, NFC, UWB, V2X, and RF-based access systems, a set of recurring weaknesses emerges: excessive trust in external input, weak isolation between wireless domains and internal vehicle systems, insecure fallback behavior, poor credential or key management, and insufficient validation of wireless data before it influences higher-privilege functions.

Let's examine how these weaknesses apply to each wireless interface:

- In **Wi-Fi-enabled** vehicle systems, weaknesses frequently arise from misconfigured access points, outdated encryption schemes, shared credentials, or unintended exposure of internal services once a client is connected. Because Wi-Fi often bridges external devices into higher-bandwidth vehicle platforms, poor segmentation can allow a compromise of a convenience feature to expand into broader system access.
- **Bluetooth-related** weaknesses often stem from insecure pairing workflows, overly persistent trust relationships, excessive privileges granted to paired devices, and broad exposure of services beyond what is functionally necessary [2]. In practice, profile implementations, trust caching, and insufficient access control over Classic Bluetooth or BLE services can allow a temporary pairing event to become a long-lived attack path. For example, vehicles may expose hands-free contacts, messages, media, diagnostic, or proprietary services over Bluetooth, sometimes without authentication or by abusing weak pairing mechanisms. Relevant Bluetooth profiles that support these types of use

cases include **Hands-Free Profile** (**HFP**), **Phone Book Access Profile** (**PBAP**), **Message Access Profile** (**MAP**), **Audio/Video Remote Control Profile** (**AVRCP**), **Serial Port Profile** (**SPP**), and **Object Push Profile** (**OPP**). Once paired, due to trust caching, it is generally possible to achieve persistent access, making Bluetooth an attractive entry point.

- **RF-based** access systems frequently suffer from weaknesses in replay resistance, cryptographic key management, synchronization logic, and protocol design assumptions about proximity and authenticity. Because these systems directly influence vehicle entry and start authorization, even small design weaknesses can have outsized consequences.
- **UWB-related** weaknesses are often found not only in the ranging protocol itself but in the surrounding system design. Common issues include insufficient validation of distance measurements, unsafe trust transfer between radios, weak fallback handling, and implementation flaws in the logic that combines UWB with BLE, NFC, or legacy keyless-entry mechanisms.
- **NFC-related** weaknesses usually appear in credential provisioning workflows, trust bootstrapping, backend authorization, or the way NFC-derived trust is extended into higher-privilege vehicle functions [3]. Even when the radio interface itself is limited by short range, the operations triggered by NFC events may still have broad security impact if not carefully constrained.
- **GNSS** weaknesses arise from overreliance on externally received position and timing data without sufficient validation, correlation, or fallback logic. When GNSS data is trusted too broadly, manipulation of location or time can indirectly affect navigation, synchronization, geofencing, and other dependent vehicle functions.
- **V2X-related** weaknesses commonly stem from protocol-level trust assumptions, limited opportunities for strong session-based authentication, challenges in validating broadcast safety messages, and the operational difficulty of revocation and misbehavior handling at scale. Because these systems support safety-relevant decisions, weak message validation or excessive trust in received data can create disproportionate risk.

Cellular module architecture and trust boundaries

Cellular telematics units are frequently treated as communication peripherals, but in reality, they are advanced computing platforms. A typical 4G or 5G module includes its own processor, memory, storage, and operating system, often running embedded Linux. It may communicate with a gateway ECU or be directly connected to the in-vehicle network.

One of the fundamental weaknesses of cellular interfaces lies in the implicit trust placed in these modules. Once integrated, they are often granted broad access to vehicle networks while being reachable from external cellular infrastructure. This consequently breaks down the trust boundary between external networks and internal vehicle systems, making the cellular module a high-value target as an entry point for remote attackers.

Network isolation and private APN failures

Private **Access Point Names (APNs)** are a way to create isolated, dedicated networks for specific groups of devices where security policies can be more strictly enforced. The fundamental goal is to ensure that only authorized devices and users have access to these protected network segments, thus reducing exposure to external threats. For example, they can isolate traffic used by a vehicle fleet from the public internet by routing data through operator-controlled networks. However, improper configuration of operator networks and private APNs can defeat the purpose of an APN, leading to certain services on wireless vehicle systems becoming accessible over the public internet.

In other instances, networks may lack proper isolation between devices, allowing vehicles connected to the same APN to discover and communicate with each other directly. This lack of segmentation enables lateral movement once a single device within the network is compromised. An attacker who gains access to the APN used by a vehicle fleet, e.g., through a compromised SIM or rogue device, can scan for exposed services and exploit multiple vehicles without interacting with the public internet at all.

AT command interfaces and unsafe control paths

Attention (AT) commands [4], standardized in 3GPP TS 27.007 [5], were originally designed to control dial-up modems and have since been extended by module manufacturers with proprietary commands, some of which expose system-level functions without adequate input validation. They are text-based instructions that allow software to configure radio parameters, manage **Short Message Service** (**SMS**) messages, and control data sessions. Because AT commands are standardized to an extent by bodies like 3GPP, most cellular modules share a core command set but may include vendor-specific extensions. In many vehicle systems, AT commands are exposed through multiple pathways, including **Universal Asynchronous Receiver/Transmitter** (**UART**), **Universal Serial Bus** (**USB**), Bluetooth, and network sockets.

Their primary weakness lies in the lack of sufficient access control and input validation. AT command parsers may accept commands from untrusted sources. For example, in some devices, specially crafted over-the-air SMS messages can embed AT commands, potentially allowing attackers to change system settings or trigger harmful actions. Moreover, poor parsing logic and a lack of sanitization further increase the risk of command injection and unintended behavior.

Software vulnerabilities in wireless-facing components

Wireless-facing software is a common source of critical vulnerabilities. Network daemons, protocol parsers, and message handlers process untrusted input at high privilege levels and are often written in memory-unsafe languages such as the C programming language. Buffer overflows, out-of-bounds writes, and similar vulnerabilities continue to be discovered. Furthermore, **Open-Source Software (OSS)** is commonly used for components such as communication stacks (e.g., BlueZ) and networking services (e.g., dnsmasq, ConnMann). Because secure development practices and testing approaches can vary among OSS projects, vehicle systems inherit the same OSS weaknesses that affect the broader automotive supply chain.

Note

Terms:

- A network daemon in an embedded Linux system is a background process that manages network-related services or functionalities. These daemons enable communication between devices, handle network protocols, or provide specific services like remote access, file sharing, or time synchronization.
- A buffer overflow is a type of software vulnerability that occurs when a program writes more data to a buffer (a fixed-sized block of memory) than it can hold. This excess data can overwrite adjacent memory locations, leading to unintended behavior, including crashes, data corruption, or execution of malicious code.
- An out-of-bounds write is software vulnerability where a program writes data beyond the allocated bounds of a memory object (e.g., array, struct), modifying memory that should not be accessed. This vulnerability can lead to program crashes, corruption of data, or exploitation by attackers to execute arbitrary code or escalate privileges.

When such vulnerabilities are accessible remotely, they enable **zero-click attacks** that require no user interaction. Using various exploitation techniques, attackers can bypass modern mitigations and gain persistent control of the system.

Note

Terms:

A zero-click attack is an attack that exploits a security vulnerability in software or hardware without requiring any interaction from the victim. Unlike typical IT exploits that often require a user to click a link, open an attachment, or perform some action, zero-click attacks trigger automatically, often when a maliciously crafted input is received by the vulnerable system.

Weak credentials and hard-coded secrets

A recurring theme in wireless vehicle systems is the reliance on static authentication mechanisms. For example, default passwords, shared credentials, and hardcoded cryptographic keys are commonly found in cellular modules, infotainment systems, and diagnostic services. These credentials are designed to protect shell access, secure communication, firmware update mechanisms, or APIs. One example weakness is where the wireless vehicle system contained hardcoded **File Transfer Protocol (FTP)** credentials to access a remote update server. Abusing the FTP credentials allows an attacker to further target the update servers to upload malicious software. To make matters worse, there are instances where firmware update functionality on the wireless vehicle systems lacks proper **signature verification**, allowing attackers to install malicious software by simply being able to deliver the unsigned or improperly signed package to the vehicle.

Another common weakness is the reuse of the same credentials across fleets, which significantly amplifies the impact of compromise. That is, once a single password or key is recovered from one vehicle, for example through firmware analysis, reverse engineering, or documentation leaks, it can often be applied universally to the entire fleet.

Network services and debug interfaces

Wireless subsystems frequently expose additional network services beyond their core functionality. For example, services such as Telnet, **Secure Shell (SSH)**, HTTP administration interfaces, and debugging daemons such as **Android Debug Bridge (ADB)** are often enabled to support development and remote maintenance. If these services are left enabled on production systems, they introduce persistent remote entry points.

Additionally, misconfigurations can augment this weakness. Services may listen on all network interfaces rather than being restricted to localhost. This exposes the services to external threats. Moreover, it is important to note that this weakness is often combined with the **weak credentials and hardcoded secrets** mentioned above. That is, these services often use

common passwords such as `oelinux123` or may even lack authentication altogether. Once accessed, these services often provide high-privilege control over the underlying system and can serve as a launchpad for further attacks. For example, an attacker could install backdoors, install malware, or extract sensitive data.

RF systems and rolling code cryptography

RKE systems often rely on rolling codes to prevent replay attacks. In principle, a rolling code mechanism works by having both the key fob and the receiver (in the car) generate a sequence of codes that are valid only once; after a code is used, both sides move on to the next value. This approach prevents replay attacks where an attacker captures and resends an earlier valid code to unlock or start the vehicle.

However, in certain RKE implementations, static or global cryptographic keys are embedded into all devices manufactured by a given vendor. If an attacker can reverse-engineer the system and obtain these global keys, they can generate valid rolling codes on demand. Once these keys are compromised, the rolling code mechanism ceases to provide meaningful protection, allowing an attacker to trigger remote unlocking at will.

Wireless protocol design limitations

Over the years, wireless protocols have experienced various design flaws. For example, early versions of Bluetooth used PIN-based pairing (often 4 digits), making it susceptible to brute force attacks. Also, early Wi-Fi suffered from a cryptographic design flaw in **Wired Equivalent Privacy** (**WEP**), which enabled attackers to calculate the encryption key from information obtained in captured wireless traffic [6]. On the other hand, V2X communication protocols may minimize authentication overhead as they prioritize low latency and availability to support safety-critical use cases. This makes the protocol susceptible to spoofing, denial-of-service, and message suppression attacks.

For example, the unicast acknowledgment mechanism in IEEE 802.11p [7] **Outside the Context of a Basic Service Set** (**OCB**) mode introduces a **head-of-line blocking** condition that can halt all outbound transmissions from a vehicle or RSU, including broadcast safety messages, without requiring any vulnerability in the implementation [8]. When building wireless attack chains, it pays off to analyze the wireless protocols in use to determine if there are fundamental design flaws or implementation errors that are worthy of exploitation.

Note

Terms:

- Outside the Context of a Basic Service Set (OCB) is a mode of operation in IEEE 802.11p where devices communicate without forming a traditional Wi-Fi network. There is no access point, no association, and no authentication

handshake, allowing devices to transmit immediately. This enables low-latency communication but removes many structural security mechanisms.

- Head-of-line blocking is a condition where the first frame in a transmission queue cannot be delivered, preventing all subsequent frames from being transmitted, regardless of priority.

Now that we have a shared understanding of common weaknesses in wireless vehicle systems, the next section explores relevant penetration testing techniques that can be used to assess and validate the security of each wireless interface.

Penetration testing techniques for wireless vehicle systems

Automotive wireless attacks are not theoretical but are rather grounded in systematic reconnaissance, careful analysis of trust boundaries, and the creative chaining of multiple flaws into reliable attack paths. This section introduces concrete penetration testing activities that leverage some of the weaknesses described in the previous section.

It presents the tools, techniques, and workflows used by security researchers and professional testers to evaluate wireless vehicle systems under realistic threat models. Our emphasis is placed on remote and low-interaction attacks, reflecting the most critical risks to vehicle fleets at scale. By detailing what testers look for, how they exercise each wireless interface, and how individual findings are chained into broader compromise scenarios, this section equips both defenders and testers with a practical understanding of wireless vehicle penetration testing.

Wireless attack surface discovery and validation

Penetration testing of wireless vehicle systems begins with systematic discovery and validation of all radio interfaces and wireless protocols present in the vehicle. This phase focuses on determining what is actually exposed, how it behaves under adversarial conditions, and whether its integration matches architectural intent. The output of this phase is an evidence-based inventory of reachable radios, exposed services, pairing or association behavior, observed message formats, and trust relationships between the wireless interface and internal vehicle systems.

- **Wi-Fi testing** typically starts with passive and active discovery using tools such as **Kismet** [9], **airodump-ng** [10], and **iw** [11]. Using such tools, penetration testers can identify access points, encryption modes, hidden **Service Set Identifier** (**SSID**), and client behavior. In practice, testers capture beacon and probe traffic, determine

whether the vehicle advertises or automatically joins known networks, and then validate whether association to the network exposes IP services or diagnostic paths. Once connectivity is established, tools like **Nmap** [12], **masscan** [13], and **netcat** [14] are used to enumerate reachable services and internal network topology. One area of focus is on whether Wi-Fi connectivity bridges directly into internal Ethernet or CAN-adjacent networks.

- **Bluetooth testing** involves scanning for Classic and BLE devices using tools such as **Ubertooth** [15], **bluetoothctl**, **btmgmt**, **hcitool**, and **gatttool**, as well as more advanced frameworks like **Btlejack** [16] and **BlueZ** [17] debugging utilities. Penetration testers analyze pairing workflows, service discovery, and attribute permissions. For BLE, this includes enumerating GATT services, reading and writing characteristics where permitted, and checking whether sensitive operations require pairing, bonding, or application-layer authorization.
- **RF-based RKE systems** are examined using software-defined radios such as **HackRF** [18], **Yard Stick One** [19], or **USRP** [20] combined with tools like **GNU Radio** [21] and **Universal Radio Hacker** [22]. Penetration testers use these types of tools to capture key fob transmissions. The first step is usually passive capture, so testers can determine frequency, modulation, packet structure, timing, and whether messages change predictably between button presses. After that, testers will analyze modulation and protocol behavior, and evaluate replay resistance, rolling code synchronization, and cryptographic robustness. The focus here is not only on replay attacks but also on relay feasibility, desynchronization, and identifying whether the same key is reused across multiple authentication events or systems.
- **UWB testing** extends RF analysis into time-of-flight measurement and ranging validation. Specialized UWB development kits and protocol analyzers are used to observe ranging exchanges and validate distance calculations. The practical test is to compare reported distance against controlled physical distance, added delay, and fallback behavior when UWB ranging fails or becomes unreliable. The focus of testing is on whether UWB can be manipulated, delayed, or bypassed, and whether fallback to legacy RKE introduces exploitable downgrade paths.
- **NFC interfaces** are tested using tools such as **Proxmark3** [23], [24] and **Chameleon** Mini [25] devices to emulate cards, replay exchanges, and probe authentication workflows. Rather than focusing solely on cryptographic strength, penetration testers can examine how NFC-derived trust is propagated into vehicle systems. The practical question is whether an NFC event merely unlocks a local workflow or whether it grants reusable trust to backend, Bluetooth, UWB, or in-vehicle services.

- **GNSS testing** is performed using GNSS simulators or **Software-Defined Radio** (**SDR**)-based spoofing tools to inject controlled location and time signals. Testers typically begin with low-impact simulation in a controlled environment, gradually varying position, velocity, and time to identify which vehicle features consume GNSS data and whether they reject implausible changes. Penetration testers then observe how navigation systems, security features, and dependent services respond to the anomalous data injected. The focus is on detecting unsafe assumptions or trust correlations between systems.
- **V2X attack surface discovery** involves capturing and analyzing broadcast messages using dedicated V2X radios, SDRs, and protocol analyzers. Penetration testers evaluate message authenticity, replay resistance, and the target system's behavior under message flooding or selective suppression. Because V2X systems often lack traditional session semantics due to low latency communication, the focus of testing is on protocol resilience and detecting abnormal behavior.

Another tool worth mentioning is **Wireshark** [26], which can be used for post-capture analysis. After wireless traffic has been captured and converted into Packet Capture (PCAP) format (except for technologies like NFC and GNSS that are not directly packet-based), Wireshark enables decoding and inspection of communication to determine which protocols, profiles, or network services are in use and assess the exposed attack surface.

For all wireless interfaces, this phase focuses on understanding how the system communicates by observing real traffic rather than relying on assumed architecture. By validating exposure, trust boundaries, and integration paths, penetration testers can establish the foundation for deeper exploitation activities described in the following subsections.

Cellular module compromise and privilege analysis

Cellular telematics units should be considered and tested as independent computing systems. Penetration testers first attempt to gain logical access by targeting exposed services over the cellular interface, often discovering HTTP management portals, SSH daemons, or proprietary update services. If access to these services requires authentication, an analysis of whether weak or hardcoded credentials are used is conducted as described in the next subsection. Additionally, testing to brute-force the password may also be performed.

The next step is firmware extraction. Penetration testers obtain firmware images through vendor update servers, diagnostic interfaces, or physical access. Tools such as **binwalk** [27], **firmware-mod-kit** [28], and **strings** are used to analyze filesystem contents, identify running services, and locate configuration files. Once access is obtained, testers assess how the cellular module communicates with the vehicle's internal networks to explore whether it has direct access to traffic over CAN or Ethernet.

The key goal is to determine what a compromised cellular module can reach: only its own management plane, backend-facing services, internal Ethernet, CAN-adjacent gateways, diagnostic functions, or higher-privilege vehicle domains.

Private APN lateral movement and network abuse

First, penetration testers try to gain access to the private APN using compromised SIM cards or rogue cellular endpoints such as a private **femtocell**. Once they have access to the private APN, penetration testers can assess whether the connected vehicles in the same APN are isolated from each other or if lateral communication is possible. They use network discovery tools such as **Nmap** to enumerate peer devices, identify shared infrastructure, and scan for exposed services across the private APN. The core test is whether the APN behaves like an isolated access network or like a flat private network where one compromised device can discover and communicate with others.

If lateral communication is possible, testers evaluate whether an initial compromise can be used to reach other vehicles, fleet services, or backend management systems. This helps determine whether the APN limits blast radius or whether a single exposed device could become a pivot point into the larger connected vehicle environment.

AT command injection and control channel abuse

AT command interfaces are tested by identifying all pathways through which commands can be injected. This includes UART interfaces, USB, Bluetooth, and other networking services such as SMS processing. Tools such as **minicom** [29], **screen**, and custom Python scripts (using **pySerial**) are used to interact with AT interfaces directly to inject commands. Testers should first identify whether the AT channel is directly reachable, indirectly reachable through another parser, or reachable only after local access. This distinction matters because a locally exposed AT interface supports privilege escalation, while a remotely reachable AT path can become an initial access vector.

For SMS-based testing, penetration testers simulate attacker-controlled messages using mobile test equipment or controlled lab environments using tools like **OpenBTS** [30] or **Osmocom** [31]. They evaluate whether incoming messages are authenticated, sanitized, or blindly forwarded to the modem. SMS-based testing frequently uncovers vulnerabilities such as command chaining, where multiple commands can be executed in a single message, buffer overflows in SMS command parsers, and logic errors where commands are processed in unintended or insecure states.

Successful exploitation during this activity demonstrates how a low-level modem control channel can become a remote attack vector when messages are insufficiently authenticated, parsed, or isolated from vehicle-facing functions.

Finding vulnerabilities in wireless-facing software

Wireless-facing software components are prime targets because they parse attacker-controlled input received over Wi-Fi, Bluetooth, cellular, RF gateways, V2X radios, or update channels. If penetration testers have access to the source code, they can use **static analysis** tools to examine the software and identify potential vulnerabilities. Further, **Software Composition Analysis** (**SCA**) tools are typically used to identify known vulnerabilities in OSS components. These tools can perform automated scanning of source code or binaries and check them against vulnerability databases such as the **National Vulnerability Database** (**NVD**).

Moreover, penetration testers can perform **fuzz testing** or **fuzzing** using tools such as **AFL++** [32], **libFuzzer** [33], and custom harnesses tailored to proprietary protocols to identify unknown vulnerabilities. Using such tools, malformed or unexpected inputs are sent to the software, targeting network daemons, message parsers, and update functionality, to observe whether it crashes or causes abnormal behavior. A strong fuzzing workflow starts by identifying the parser or service that handles wireless input, building a harness or replay setup that can feed messages into that component, and monitoring for crashes, memory corruption, hangs, or unsafe state transitions.

If specific vulnerabilities are detected, for example, a buffer overflow vulnerability, they can be further analyzed to evaluate the potential for exploitation and impact. If it is within the scope of the penetration testing assignment, testers may also develop exploits for specific vulnerabilities to demonstrate the potential impact of attacks.

This activity connects vulnerability discovery to exploitation by showing whether malformed wireless input can move beyond a crash and become code execution, privilege escalation, denial of service, or a reliable foothold for the next stage of the attack chain.

Authentication weaknesses and secret extraction

Testing authentication mechanisms focuses on identifying static credentials, shared secrets, and improper trust assumptions. Default passwords are tested against exposed services, while firmware analysis is used to extract hardcoded credentials and cryptographic material. Penetration testers examine configuration files, binaries, and initialization scripts to identify hardcoded secrets. The practical workflow is to collect candidate secrets from firmware, configuration files, mobile applications, update packages, logs, and diagnostic scripts, then test whether those secrets authenticate to local services, cloud endpoints, or vehicle-side interfaces.

Reverse engineering tools such as **Ghidra** [34], **IDA Pro** [35], and **Radare2** [36] are used to analyze authentication logic when credentials are not stored in plaintext. In many cases, credentials are obfuscated, and included in the binary rather than securely stored using secure

hardware, which allows testers to recover the credentials. Penetration testers should also assess whether recovered secrets can be reused across vehicles, confirming fleet-wide impact.

This activity often reveals whether authentication failures stem from architectural mistakes, such as shared fleet-wide credentials, excessive trust in the modem, or secrets reused across vehicles and backend services.

Exploiting network services and debug functionalities

Network services exposed by wireless subsystems are tested using standard service enumeration and exploitation techniques. Penetration testers can use tools such as **Nmap**, **netcat**, **curl** [37], and **Metasploit** [38] to identify misconfigured services and outdated software versions. Debug interfaces such as Telnet, SSH, and ADB, are evaluated for authentication bypasses and privilege escalation paths. Moreover, penetration testers can potentially use the secrets extracted from the **previous authentication weaknesses and secret extraction** activity to gain access to these services.

The method is straightforward but important: identify listening services, fingerprint versions, test authentication, check whether credentials recovered earlier work, and then determine the privilege level and network reachability obtained after login or exploitation. Penetration testers should pay particular attention to services bound to all interfaces rather than localhost, as these frequently expose root-level access unintentionally to remote incoming communication.

This activity is useful because development artifacts often provide the shortest path from wireless access to command execution, especially when debug services run with elevated privileges or are reachable from non-debug interfaces.

RF and rolling code attacks

RF-based systems are tested using SDRs to capture, analyze, and replay wireless signals. Tools such as **GNU Radio** and **Universal Radio Hacker** are used to identify the operating frequency, modulation scheme, packet structure, timing behavior, and whether the protocol uses static codes, rolling codes, challenge-response, or another proprietary mechanism.

The practical workflow begins with passive capture of multiple transmissions from the same key fob or sensor. Testers compare captures to determine which fields remain constant, which fields change, and whether sequence counters, synchronization windows, or identifiers can be inferred. Replay testing is then performed in a controlled environment to determine whether previously captured messages are accepted, rejected, or cause desynchronization.

The goal is to determine whether the implementation relies on static secrets, predictable rolling behavior, weak synchronization logic, or reusable cryptographic material. Where

permitted by scope, testers may also analyze the key fob or RF receiver firmware to determine whether secrets can be extracted or whether firmware update mechanisms can be abused.

Wireless protocol-level attacks

Wireless protocol-level attacks focus on weaknesses in the communication model itself rather than bugs in a specific implementation. In this subsection, the focus is on V2X communication because V2X systems rely on low-latency broadcast messages that must be processed quickly by nearby participants.

For V2X systems, penetration testers capture baseline traffic, identify message types, and then evaluate whether spoofed, replayed, delayed, flooded, or selectively suppressed messages influence the target system. This involves transmitting controlled test messages with dedicated V2X radios, SDRs, or simulation equipment while observing whether the receiving system accepts the message, rejects it, flags it as abnormal, or changes vehicle or driver-facing behavior.

These tests assess whether protocol design assumptions allow attackers to disrupt safety message flow, impersonate infrastructure, inject misleading information, or degrade availability. Rather than exploiting implementation bugs, protocol-level testing reveals whether the system remains resilient when the wireless environment itself becomes adversarial.

Now that we have surveyed common penetration testing techniques, the following case studies examine real-life attacks in which these techniques were applied to successfully compromise wireless vehicle systems.

Case studies of real-life attacks on wireless vehicle systems

The following case studies represent the work of security researchers that took months to produce positive results. Dedicating a few pages per case study will certainly fail to capture all aspects of the attack and how the researchers achieved their success. For that reason, we will intentionally focus on aspects of the attacks that are closely related to wireless communication while also presenting other aspects that are pertinent to each case study. The reader is highly encouraged to study the reference material to gain greater insight into how attacks were carried out in each case.

Case study 1: Cloning rolling code remote keyless entry key fobs via global master key exploitation

Security researchers exposed critical vulnerabilities in the RKE systems deployed across millions of Volkswagen Group vehicles manufactured between 1995 and 2016 [39]. These

systems use rolling code schemes in which each button press transmits a code derived from a shared cryptographic secret. Reverse engineering of ECU firmware revealed that all vehicles within a given model line share a small set of global master keys rather than per-vehicle secrets.

As a result, an attacker who captures a single RF transmission from the owner's key fob can recover the counter, clone the remote control, and gain unauthorized access to the vehicle. The researchers demonstrated a complete hardware setup costing approximately $40 using an RTL-SDR USB receiver and an inexpensive RF transmitter module.

Note

Source: `https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_garcia.pdf` [39]

Figure 5.4 presents the flow of the attack steps for this case study:

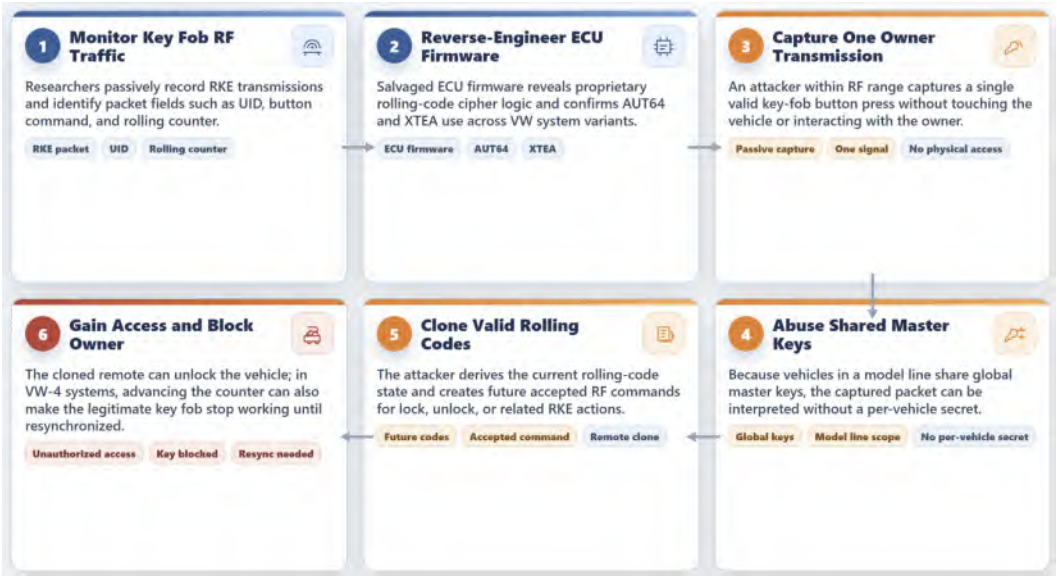


Figure 5.4: Remote keyless entry: RF rolling code exploitation flow

Discovery and reconnaissance

The research began with passive monitoring of RF transmissions from VW Group key fobs. The researchers used various SDR tools, including **HackRF**, **USRP**, and **RTL-SDR** [40], USB receivers, to record rolling code transmissions at the frequencies used across model variants: 433.92 MHz for the oldest systems (VW-1) and 434.4 MHz for vehicles produced from

approximately 2004 onwards (VW-2 through VW-4). Analysis of the captured RF packets revealed a consistent structure comprising a **Unique Identifier** (**UID**), the pressed button code, and a rolling counter value.

Using standard automotive programming tools, the researchers dumped the firmware from salvaged ECUs and performed **static analysis** of the firmware images, searching for constants and structural patterns associated with known symmetric ciphers. This process allowed them to locate and recover the proprietary cipher implementations used to encrypt each rolling code packet. The ciphers were identified as AUT64, used in VW-1 through VW-3 variants covering vehicles from approximately 1995 to 2009, and XTEA, used in the VW-4 variant covering later models.

Critically, the analysis confirmed that the cryptographic master keys were not generated per vehicle but were shared globally across all vehicles using the same system, meaning that keys extracted from a single ECU could be applied to any other vehicle in the same model line.

Initial access

Initial access in this case study relied entirely on passive eavesdropping of RF transmissions. Because RKE systems broadcast key fob signals openly over the air without any authentication, pairing mechanism, or channel encryption, any RF receiver operating within the relevant frequency range can capture these transmissions. The researchers needed to intercept only a single button press from the owner's key fob to obtain a valid rolling code packet containing the UID and counter value.

> **Note**
>
> The unidirectional and unauthenticated nature of RF-based rolling code systems makes passive interception straightforward for any attacker within RF receiving range. It is worth noting that proximity to the vehicle itself is not required at the capture stage. An attacker only needs to be within receiving range of the key fob at the moment the owner presses a button, regardless of where the vehicle is parked.

Exploitation

With a captured rolling code packet, the researchers decrypted the payload using the global master key recovered during firmware analysis. The AUT64 and XTEA cipher implementations shared the same fundamental weakness: all vehicles of a given model line used identical master keys, so decryption required no per-vehicle secret material. Decryption yielded the UID and the current counter value. The researchers then generated new valid rolling codes by incrementing the counter and re-encrypting the result with the same master key. These generated codes were then transmitted over the RF channel using the SDR setup, and the

target vehicle accepted them as legitimate commands. The researchers were able to lock, unlock, and issue other commands remotely without any physical access to the vehicle.

Persistence

A cloned remote derived from a single intercepted signal operates indefinitely, provided the vehicle's counter does not advance beyond the attacker's known state. It is important to note that the researchers identified a specific persistence-related consequence in VW-4 systems: if an attacker transmits at least two valid rolling codes with counter values sufficiently ahead of the vehicle's current expected value, the original key fob is blocked. Thus, the vehicle rejects any rolling code arriving with a counter more than two steps behind the expected value, resulting in the owner's remote becoming inoperable.

Recovery requires a manual resynchronization procedure as described in the vehicle owner's manual, a process that is not straightforward for most users. In practice, this means the attack can simultaneously grant the attacker persistent wireless access and deny the legitimate owner the use of their key fob.

Impact

This RKE vulnerability affected vehicles produced across multiple decades and model lines. The researchers showed that the practical consequence was that unauthorized vehicle access was achievable with minimal off-the-shelf equipment, no prior physical contact with the vehicle, and no user interaction. In VW-4 variants, this attack additionally enabled a denial-of-service condition against the legitimate key fob.

The root cause was a design decision to share cryptographic master keys globally across an entire model line rather than provisioning per-vehicle secrets. This choice eliminated any incremental security benefit against an attacker: once a global key is recovered, every vehicle using the same scheme is equally compromised.

Case study 2: 4G module hacking — attacking the vehicle's external gateway

This case study examines how researchers targeted automotive 4G modules as externally reachable computing platforms to achieve their objective of influencing the vehicle's internal network [41], [42]. These modules are typically embedded in telematics units to enable remote services such as vehicle status monitoring, over-the-air updates, remote unlocking, and emergency services.

The researchers first collected and analyzed 4G modules and firmware images to understand their hardware, operating systems, exposed services, credentials, and update mechanisms. They then used cellular network access, private APN exposure, weak service authentication, and SMS-accessible AT command paths to demonstrate how a remote attacker could gain

control of the telematics module. Once the module was compromised, they showed how firmware update weaknesses, hardcoded credentials, and insufficient network isolation could turn a single-device compromise into a broader fleet-level risk.

Note

Sources:

- `https://www.youtube.com/watch?v=OORUkEsannA` [41]
- `https://i.blackhat.com/USA-19/Wednesday/us-19-Shupeng-All-The-4G-Modules-Could-Be-Hacked.pdf` [42]

The attack steps sequence is shown in *Figure 5.5*:

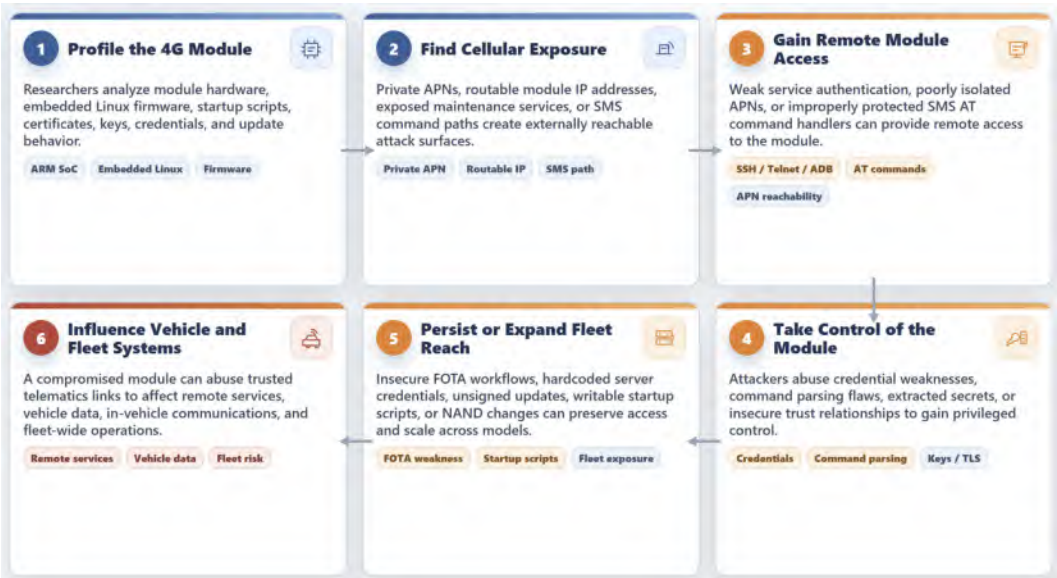


Figure 5.5: 4G module: external gateway exploitation flow

Discovery and reconnaissance

During discovery, the researchers first identified the 4G modules used across different vehicle platforms and treated each module as a standalone embedded system worthy of full hardware and software analysis. They demonstrated that 4G modules across vendors and vehicle platforms share strikingly similar architectures and software stacks.

The researchers collected and analyzed a large number of 4G modules extracted from vehicles, infotainment systems, and telematics units. Hardware analysis revealed a consistent structure among the 4G modules: typically, an ARM-based **System-on-Chip** (**SoC**) integrating baseband functionality, application processing, and NAND flash storage. On the software side, these modules hosted embedded Linux with standard userland tools, making them easier to analyze than modern smartphones.

Next, firmware was obtained through multiple channels: OEM update packages, vendor flashing tools, or physical NAND extraction from scrapped or test vehicles. Once extracted, the researchers used **binwalk** to identify the filesystem structure and startup scripts, and used **UBI reader** [43] to unpack the filesystem contents, revealing hardcoded credentials and cryptographic keys embedded in the firmware.

In parallel, the researchers analyzed operator networks and private APNs used by automotive OEMs. They also collected user manuals and developer documentation to analyze supported AT commands and configuration options, laying the groundwork for further exploitation.

> **Note**
>
> It is important to note that poorly segmented APNs allow attackers with a valid SIM to scan and communicate with other vehicles on the same network.

Initial access

Initial access to a vehicle's 4G module did not require physical proximity to the car. Instead, several remote vectors were explored to gain initial access. One method involved **cellular network manipulation**. The researchers deployed a fake base station using BladeRF [44] and YateBTS [45]. By forcing a downgrade to 2G or manipulating cell reselection parameters, nearby vehicle modules connect to the researchers' controlled base station, granting full visibility and control over network traffic.

Moreover, the researchers found **exposed network services** on the modules they tested. In addition, the researchers confirmed that on certain operator networks without firewalling or IP isolation, the modules' services were directly reachable from the outside. The researchers used the fast-scanning tool **masscan** and found several modules with open listening ports including port 23 for Telnet, port 5555 for ADB, SSH services with empty or weak passwords, and web management interfaces protected only by weak credentials.

The researchers also identified **SMS-based AT command handling** as an entry point and explored it further during exploitation.

Note

Some automotive modules allow control commands to be received via SMS for diagnostics or provisioning. If improperly authenticated, knowledge of the vehicle's phone number alone may be sufficient to trigger command execution remotely.

Exploitation

Once connected, the researchers escalated privileges and gained full control of the 4G module. First, the security researchers found that many modules used **default** or **hardcoded credentials** for remote management services. For example, the default password `oelinux123` was found across multiple modules and allowed immediate shell access with no further exploitation needed.

In one telematics unit example, the researchers identified a process listening on an internal port that was intended for inter-process communication but was bound to all interfaces rather than localhost, making it externally reachable. This process used a **Public Key Infrastructure (PKI)**-based protocol combining RSA certificate verification and AES encryption. Through firmware analysis, the researchers identified three weaknesses in this implementation: the AES key was hardcoded directly in the binary, the RSA private key was stored on the filesystem, and the password protecting that private key was also hardcoded in the binary.

By recovering the RSA private key and deriving the corresponding public key, the researchers were able to bypass TLS authentication on the port. With authentication bypassed, they sent commands through the port to activate a hidden Telnet service on the telematics unit. To log in to the Telnet service required a root password. The researchers extracted the `/etc/passwd` file from the firmware and ran the password cracking tool **hashcat** [46] using powerful GPUs to crack the password in one day. The password was complex, including uppercase and lowercase letters, numbers, and special characters. Having obtained root shell access, the researchers were able to inject CAN-related messages from the telematics unit to in-vehicle ECUs, demonstrating control over vehicle functions such as door locks and engine start.

Moreover, the researchers demonstrated that AT commands could be sent to the 4G module via SMS. Sending AT commands via SMS required a password, which the researchers obtained through reverse-engineering of the firmware, where it was found to be hardcoded. Furthermore, they demonstrated that certain SMS-triggered AT command handlers lacked proper input sanitization, allowing them to append system-level commands to legitimate modem instructions. When processed by the firmware, these injected commands executed with root privileges, effectively resulting in **remote code execution**.

Note

Remote code execution via SMS-triggered AT command injection gives an attacker arbitrary control over the module without requiring a persistent network connection, including the potential ability to modify telematics configurations, access sensitive data, or manipulate communication with in-vehicle ECUs.

Persistence

Persistence is achieved by compromising the update and boot mechanisms of the 4G module, ensuring long-term control even after reboots or resets. The researchers discovered hardcoded FTP credentials embedded in the module firmware and used them to access manufacturer **Firmware Over-the-Air (FOTA)** servers directly. Then, the researchers showed that they could replace legitimate firmware with **backdoored versions**, since the FTP account also had write access.

Furthermore, firmware updates were not protected by signature verification, meaning any device of that brand would silently install the backdoored firmware. They demonstrated that persistent access could thereby be gained across all modules of a given brand. At the device level, the researchers also demonstrated that NAND flash contents and init scripts such as `init.rc` could be modified directly to embed **persistent backdoors**.

Impact

The researchers demonstrated that a compromised 4G module could be used to **remotely** control vehicle functions. In one example, they showed that CAN-related commands could be sent from the compromised module to in-vehicle ECUs to control functions including door locks and engine start. At scale, the researchers showed that **poorly isolated private APNs** made it possible to reach a large number of vehicles on the same network simultaneously. Combined with the FOTA findings, they demonstrated that a single compromised FOTA server could be used to deploy backdoored firmware across all modules of a given brand in less than one day.

Note

The weaknesses demonstrated in this research, such as default passwords, development services left open in production, and insecure FOTA servers, are not unique to one vendor. Because these patterns are shared across entire product lines and OEM fleets, a single exploit has the potential to affect a large number of vehicles simultaneously, creating zombie fleets as the researchers described.

From an automotive cybersecurity perspective, this research illustrates that 4G modules should not be treated as peripheral connectivity components. Because they run a full Linux environment, hold cryptographic keys, and communicate directly with in-vehicle ECUs, they are safety-relevant systems that require the same security rigor applied to other critical vehicle components.

Case study 3: Zero-click remote code execution on infotainment system

This case study focuses on sophisticated attack techniques and delivery methods to achieve remote code execution on an infotainment system. This Tesla infotainment attack [47], [48], widely known as TBone, represents one of the most illustrative examples of modern automotive wireless exploitation. The attack is a **zero-click remote code execution exploit** targeting Tesla's Linux-based infotainment system over Wi-Fi. By chaining together multiple memory corruption vulnerabilities in ConnMan (network connection manager), researchers were able to gain full control of the infotainment domain without any physical access to the vehicle.

What made this demonstration particularly interesting was not only the technical sophistication of the exploit chain, but also its delivery method. The researchers used a drone equipped with a Wi-Fi dongle and antenna to deploy the attack from the air, successfully compromising a vehicle from a distance of roughly 100 meters. While the exploit did not provide direct control over safety-critical driving functions, it clearly demonstrated how weaknesses in peripheral systems can still result in serious security consequences.

Note

Sources:

- `https://kunnamon.io/tbone/` [47]
- `https://www.youtube.com/watch?v=krSj81thN0w&ab_channel=secwestnet` [48]

Figure 5.6 presents the flow of the attack steps for this case study:

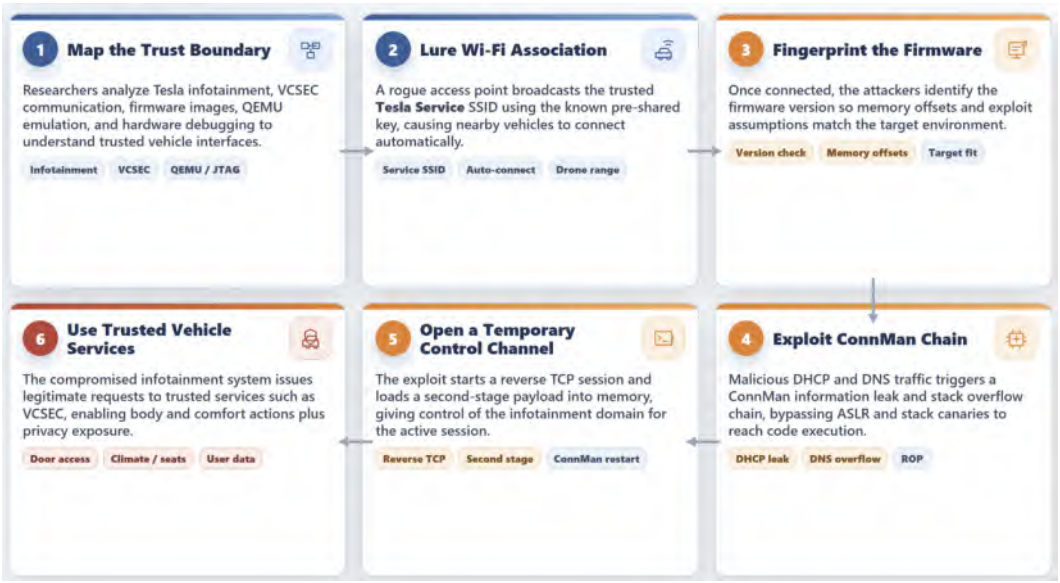


Figure 5.6: Tesla TBone: Wi-Fi to infotainment remote code execution flow

Discovery and reconnaissance

The attack began with an in-depth study of Tesla's electronic and software ecosystem, with particular attention given to the infotainment system and the **Vehicle Control Security (VCSEC)** module. These two components were selected as together they form a critical boundary between externally exposed interfaces and internal vehicle functions. The infotainment system represents one of the most complex and network-accessible domains in the vehicle, while VCSEC acts as a security gateway responsible for authenticating and authorizing many body-related actions, including door locks and other access-controlled features.

A key focus of the reconnaissance was understanding how the infotainment system communicates with VCSEC. Although the infotainment unit cannot directly control safety-critical driving functions, it sends authenticated requests to VCSEC for actions such as door access and comfort features. This made the infotainment system a desirable target, since compromising it allows an attacker to issue legitimate commands to in-vehicle ECUs through VCSEC.

To facilitate deep analysis, the researchers acquired salvaged Tesla hardware, allowing them to study both systems outside the constraints of a live vehicle. The researchers used **JTAG** to dump firmware from VCSEC-related components and subjected the firmware to extensive **static analysis** to understand trust relationships, message flows, and privilege boundaries.

The firmware was successfully emulated using **KunnaEmu** [47], a custom emulator developed by researchers. The researchers then performed fuzzing of the firmware running on the emulator.

In parallel, the researchers also analyzed the infotainment system, which was a normal Linux system, to identify potential attack vectors and understand existing protection mechanisms. The infotainment system was emulated using **QEMU** [49] running on an Intel NUC. Through this process, the researchers mapped out the system's attack surface and identified ConnMan as a particularly attractive target. ConnMan is a third-party application responsible for managing network connectivity, including Wi-Fi, **Dynamic Host Configuration Protocol (DHCP)**, and **Domain Name System (DNS)**, and is written in memory-unsafe C code. Crucially, it is accessible over Wi-Fi and handles complex network data that an attacker could control. The researchers then performed fuzzing of ConnMan using **AFL** [50].

Note

Terms:

- Dynamic Host Configuration Protocol (DHCP) is a network protocol used to automatically assign IP addresses and network configuration parameters to devices.
- Domain Name System (DNS) is a protocol that translates human-readable domain names into IP addresses.

During reconnaissance, the security researcher team also discovered a critical configuration weakness: European Tesla vehicles were programmed to automatically connect to a predefined Wi-Fi network named `Tesla Service`, intended to support diagnostics and servicing. The pre-shared key for this SSID was embedded directly in the firmware, and further investigation revealed that it had already been publicly shared online. This combination of automatic trust and exposed credentials set the stage for reliable remote access.

By the end of the reconnaissance phase, the researchers had identified not only a vulnerable service in ConnMan, but also a clear attack path from wireless access to infotainment compromise, with potential downstream implications for VCSEC-mediated vehicle functions. This combination of exposed wireless trust, memory-unsafe software, and cross-domain interaction laid the groundwork for the fully remote exploit demonstrated in the subsequent phases of the attack.

Initial access

Initial access relied not on exploiting a vulnerability in isolation, but on abusing the vehicle's normal networking behavior. By configuring a malicious Wi-Fi access point broadcasting the `Tesla Service` SSID and using the known pre-shared key, the researchers could reliably cause a nearby Tesla to connect to their network.

In the TBone demonstration, this rogue access point was mounted on a drone equipped with a Wi-Fi dongle and antenna, allowing the researchers to approach a vehicle covertly and initiate the connection from the air. As soon as the car entered the rogue network, the researchers could communicate directly with the infotainment system.

It is important to note that this initial foothold did not immediately grant control over vehicle functions or directly provide access to internal vehicle buses. However, it allowed the researchers to enter the infotainment trust zone, from which they could interact with system services that legitimately communicate with other vehicle components, including VCSEC. From an architectural perspective, this was a critical step: the infotainment system acts as a bridge between external interfaces and internal authorization mechanisms, making its compromise particularly dangerous.

Once connected over Wi-Fi, the researchers used the initial wireless network communication to fingerprint the firmware version. This step was essential, as precise knowledge of the firmware version allowed them to calculate accurate memory offsets, a prerequisite for bypassing modern exploitation mitigations later in the attack chain.

Exploitation

The exploitation phase was built around a carefully constructed, multi-stage vulnerability chain designed to bypass modern Linux exploit mitigations such as **Address Space Layout Randomization** (**ASLR**) and **stack canaries**. Because of these defenses, a straightforward buffer overflow is insufficient to execute arbitrary code. Instead, the researchers used **Return-Oriented Programming** (**ROP**), a technique that chains together small instruction sequences already present in memory to perform malicious actions without injecting new code. The researchers prepared a complex, multi-stage attack chain to exploit flaws in ConnMan to achieve reliable remote code execution.

Note

Terms:

- Address Space Layout Randomization (ASLR) is a memory protection technique that randomizes the locations of code, stack, heap, and libraries in a process's

address space, making it harder for attackers to predict memory addresses during exploitation.

- Stack canaries are special guard values placed on the stack before return addresses to detect stack-based buffer overflows; if the value is altered, the program can detect corruption and terminate execution.
- Return-Oriented Programming (ROP) is an exploitation technique that reuses small instruction sequences already present in a program's memory to execute malicious logic, commonly used to bypass modern protections such as ASLR and stack canaries.

The first stage exploited a stack information leak in ConnMan's DHCP client implementation (CVE-2021-26676 [51]). By crafting malicious DHCP responses, the researchers were able to leak sensitive memory information, including stack addresses and the base address of shared libraries such as libc. This information was used to defeat ASLR and provided a precise map of the process memory layout required for the next stage of the attack.

With this knowledge, the researchers triggered a second vulnerability: a stack overflow in ConnMan's DNS reply parsing logic, specifically within the function handling uncompressed DNS responses (CVE-2021-26675 [52]). The researchers used **scapy** [53] to craft and transmit malicious DNS and DHCP packets. Because they already knew the exact stack layout, they could jump over the stack canary and use a controlled partial write to redirect the execution flow into a carefully constructed ROP chain. This enabled reliable code execution within the infotainment process, achieved entirely over Wi-Fi and without user interaction, a so-called **zero click remote code execution**.

Once code execution was established, the exploit initiated a reverse TCP connection back to a researcher-controlled server, which delivered a second-stage payload into memory. At this point, the researchers had control over the infotainment system and could interact with higher-level vehicle services.

Note

While VCSEC itself was not directly exploited, the compromised infotainment system could now issue legitimate, authenticated requests to VCSEC for permitted operations, highlighting how an infotainment compromise can indirectly influence security-critical components through authorized interfaces.

Persistence

While the researchers do not describe any persistence mechanism that survives reboots or firmware updates, it is important to note that one operational detail significantly aided exploitation: ConnMan automatically restarts whenever it crashes. This behavior allowed the researchers to repeatedly try to trigger the vulnerability until exploitation succeeded, making the attack highly reliable despite the complexity of the exploit chain.

Once the second-stage payload was running, the researchers maintained control through an active command-and-control channel for the duration of the session. While this persistence was temporary, it was sufficient to fully demonstrate the impact of the compromise.

Note

In real-world threat scenarios, persistence could plausibly be extended by modifying user-space components or abusing update mechanisms, although such techniques were not publicly demonstrated in this case.

Impact

Although the infotainment system is separated from safety-critical driving systems by an internal gateway, the consequences of a successful compromise were still noteworthy. The researchers demonstrated that they were able to shut down the firewall, open the charge port, and remotely operate the vehicle's falcon wing doors. The researchers also stated that this level of access would allow them to control a wide range of vehicle functions, including adjusting seat positions, manipulating climate controls, adjusting music volume, and other functions accessible through the infotainment console.

Many of these actions were mediated through VCSEC, which enforces authorization for body-related functions. Importantly, the attack did not bypass VCSEC's security model; instead, it abused the fact that the infotainment system is a trusted client of VCSEC. Once the infotainment system was compromised, it could issue valid requests that VCSEC was designed to accept, illustrating how the security of authorization modules is fundamentally dependent on the integrity of their upstream clients.

Note

Beyond functional control, the attack raised serious privacy concerns. Access to the infotainment domain can expose location history, user accounts, paired mobile devices, and potentially credentials or tokens used by other vehicle services. The demonstration also underscored a broader architectural risk shared across the automotive industry: even when safety-critical systems are isolated, vulnerabilities in wireless-exposed

> third-party components like ConnMan can still lead to real-world security, privacy, and safety implications.
>
> Ultimately, the TBone attack serves as a powerful reminder that in connected vehicles, no system should be considered truly "non-critical." Externally facing peripherals such as infotainment systems and third-party software components like ConnMan, the network connection manager, cannot be treated as low risk. Moreover, when these systems are trusted by security-critical modules like VCSEC, a compromise of such a system becomes a powerful steppingstone that can transform a remote wireless vulnerability into serious control over the vehicle's behavior.

Now that we have covered these real-life case studies, we will proceed with reviewing best practices that automotive organizations can adopt to reduce the risk of introducing vulnerabilities in their wireless vehicle systems.

Best practices for securing wireless vehicle systems

Securing wireless vehicle systems requires more than addressing individual vulnerabilities as they are discovered. The complexity and longevity of modern vehicles demand a principled approach that remains effective even as technologies, attackers, and usage patterns evolve.

This section presents best practices for securing wireless vehicle systems by taking into consideration common weaknesses and examples from the case studies discussed. Rather than discussing isolated fixes, we focus on architectural and organizational practices that systematically reduce the recurring weaknesses discussed earlier, including expanded attack surface, weak isolation, exposed services, unsafe control paths, hardcoded secrets, protocol limitations, and vulnerable wireless-facing software.

The objective is to shift the reader's perspective from reactive vulnerability mitigation to proactive security engineering. When applied consistently, these principles enable vehicle platforms to remain resilient in adversarial wireless environments throughout their operational lifetimes.

Treating all wireless interfaces as untrusted entry points

Every wireless interface in a vehicle, whether high-bandwidth IP-based technologies such as Wi-Fi and cellular, short-range communication like Bluetooth and NFC, low-level RF systems such as RKE and UWB, or broadcast technologies like GNSS and V2X, must be treated as an untrusted source of input. It is important to note that physical proximity, limited range, or specialized hardware alone should never be assumed to provide meaningful security guarantees.

In practice, this means that every wireless interface should have an explicit owner, threat model, exposed-service inventory, and documented trust boundary. Interfaces that are enabled only for manufacturing, diagnostics, or service operations should not remain reachable in normal operational mode. This principle is especially important for interfaces that are traditionally perceived as low risk, such as NFC or sub-GHz key systems. These interfaces frequently act as trust anchors for higher-privilege operations, including vehicle access, pairing, and digital key provisioning. Once trust is extended beyond the initial radio exchange, weaknesses can propagate into internal vehicle networks and backend systems. Thus, the security architecture must assume that any wireless signal can be observed, replayed, modified, delayed, jammed, or emulated by an attacker with sufficient motivation and equipment.

Enforcing strong isolation between wireless domains and vehicle systems

Wireless interfaces should never provide implicit access to internal vehicle networks or control domains. Each wireless technology must be isolated using explicit trust boundaries, protocol gateways, and message validation layers. This applies equally to IP-based interfaces such as Wi-Fi and cellular, and to non-IP systems such as RKE, UWB, and V2X. Moreover, proper network isolation, including segmentation into secure zones and the use of firewalls, is crucial to limiting exposure, especially for devices accessible via public networks or poorly configured private APNs. This isolation should be enforced technically rather than assumed architecturally. For example, a compromise of a Wi-Fi access point, Bluetooth service, cellular modem, or V2X receiver should not automatically provide access to internal Ethernet, CAN, diagnostic services, or safety-relevant control domains.

Isolation is particularly critical when multiple wireless technologies interact. For example, UWB ranging results may be combined with RKE logic, NFC events may trigger digital key provisioning, and GNSS data may influence navigation or time-dependent security decisions. In each case, data crossing between domains must be validated, bounded, and contextualized rather than trusted implicitly.

Thus, wireless subsystems should not be trusted merely because they are embedded within the vehicle. The design should enforce strong isolation between the wireless domain and the internal vehicle network, with every cross-domain message explicitly authorized and validated.

Designing authentication and authorization for adversarial environments

Authentication mechanisms across all wireless interfaces must be designed with active attackers in mind. Static credentials, shared secrets, and implicit trust relationships

fundamentally lead to considerable risks potentially for all vehicles across a fleet. This applies not only to IP-based services but also to RF protocols, digital key systems, and broadcast authentication schemes. To address weaknesses related to hardcoded secrets and fleet-wide credential reuse, authentication should rely on per-vehicle and per-component identities rather than shared passwords or global keys.

Per-vehicle identity, hardware-backed key storage, and cryptographically strong challenge-response mechanisms are essential. For systems such as RKE and UWB, cryptographic designs must resist replay, relay, and downgrade attacks. For V2X and GNSS-dependent systems, authentication and integrity mechanisms must account for the absence of reliable point-to-point trust.

Authorization must be equally rigorous. It determines the permitted actions and accessible resources after authentication. Successful authentication should grant only the minimum privileges required, with trust constrained by time, context, and operational conditions. Credentials should also be managed as lifecycle assets. This includes secure provisioning, hardware-backed storage, rotation, revocation, and clear recovery procedures when keys or certificates are suspected to be compromised.

Protecting credentials, keys, and provisioning workflows

Credentials and cryptographic keys used by wireless vehicle systems should never be shared broadly across vehicles, services, or suppliers. Default passwords, hardcoded secrets, and globally reused keys can turn a single compromise into a fleet-wide issue.

Keys should be generated uniquely, stored in hardware-backed storage where possible, and scoped to the minimum set of functions required. Provisioning workflows for digital keys, cellular credentials, backend certificates, and update credentials should be authenticated, auditable, and designed to support revocation.

This practice directly reduces the impact of secret extraction, firmware analysis, documentation leaks, and backend credential abuse by limiting how far a recovered credential can be reused.

Eliminating unnecessary services, legacy control paths and unsafe fallback mechanisms

Many wireless vulnerabilities arise not necessarily from modern protocols, but from legacy mechanisms retained for backward compatibility, manufacturing, diagnostics, or service convenience. For example, AT command interfaces, SMS-based control paths, insecure pairing modes, and protocol downgrade behavior are examples that could lead to weaknesses in otherwise robust designs. In addition, legacy and maintenance services that are no longer needed, such as Telnet, SSH, or ADB, should be removed or disabled in production systems to

reduce the attack surface. These services are often enabled by default and would give attackers a trivial entry point into the wireless vehicle system. Production builds should therefore be reviewed for exposed Telnet, SSH, HTTP administration interfaces, ADB, engineering endpoints, test credentials, and diagnostic services bound to external interfaces.

It is worth mentioning that fallback mechanisms deserve particular scrutiny. UWB systems that silently revert to legacy RKE behavior, Bluetooth stacks that accept unauthenticated profile access, or Wi-Fi systems that expose diagnostic services during error states all create exploitable conditions. Where possible, a fallback path should be secured to the same level as the primary mechanism; otherwise, it should not be included. Fallback behavior should also be observable and auditable. A system should fail closed when a secure wireless path is unavailable, especially when the fallback path affects vehicle access, digital key behavior, diagnostics, or update functionality. It is important to remember that unnecessary services and backward compatibility must never override security requirements, especially in systems that control vehicle access or safety-relevant behavior.

Assuming partial compromise and designing for containment

Wireless security architectures must assume that some components will eventually be compromised. Therefore, by following a defense-in-depth approach and applying segmentation, privilege separation, and narrow communication interfaces, a successful attack against a Wi-Fi stack, Bluetooth service, cellular modem, RF receiver, or V2X component should not enable unrestricted access to other connected systems or safety-critical vehicle functions.

This type of containment requires strong process isolation, privilege separation, and network segmentation. Furthermore, wireless subsystems should operate with minimal privileges and communicate with vehicle systems through narrow, well-defined interfaces. This principle is essential for limiting the impact of attacks against external-facing systems. Containment should also apply within the wireless subsystem itself. Network daemons, protocol parsers, update clients, and message handlers should not run with unnecessary privileges to prevent a compromise of one process from providing unrestricted control over the entire module.

By taking into consideration that successful attacks can occur and by designing for containment in such cases rather than aiming for perfect prevention of attacks, the systems can remain resilient even when individual defenses fail.

Validating all wireless input semantically, not just syntactically

Input validation is a critical defense against command injection and other attacks. All inputs, including those from interfaces like AT commands and SMS, must be thoroughly validated and sanitized before processing. Proper sanitization ensures that malicious commands or data do

not compromise the device or its operations. This is especially important for AT command interfaces, SMS handlers, Bluetooth profiles, diagnostic messages, and proprietary RF or cellular control paths, where syntactically valid input may still represent an unsafe operation.

Moreover, any input data received over wireless interfaces must further be validated beyond basic protocol correctness. For example, plausibility checks, rate limiting, temporal validation, and cross-sensor correlation are essential, particularly for broadcast and sensor-dependent systems such as GNSS and V2X. For example, GNSS data should be evaluated against inertial sensors and map constraints, and V2X messages should be assessed for consistency with local observations. Furthermore, RF-based access systems should validate timing, sequence, and context rather than relying solely on cryptographic correctness.

In other words, semantic validation reduces the effectiveness of attacks where the adversary can provide technically valid but operationally unsafe input to the target system.

Hardening wireless-facing software and third-party components

Wireless-facing software should be treated as high-risk code because it processes input received from external networks and radio interfaces. Components such as Bluetooth stacks, Wi-Fi services, cellular modem daemons, DNS and DHCP services, V2X message handlers, and update clients should be designed and tested with the assumption that attackers can directly influence the data they process. Where possible, memory-safe languages should be preferred for new wireless-facing components. When memory-unsafe languages such as C or C++ are required, teams should apply strict compiler hardening, static analysis, fuzz testing, dependency scanning, and code review focused on parser and message-handling logic.

Third-party and open-source components should be inventoried, monitored, and updated throughout the vehicle lifecycle. This includes maintaining a Software Bill of Materials (SBOM), tracking known vulnerabilities, and ensuring that vulnerable wireless-facing dependencies can be patched without weakening update authenticity or introducing new trust assumptions.

Building security into the development lifecycle

Wireless vehicle security is not a one-time engineering task. Radios, protocols, backend services, third-party components, and attacker tooling evolve over time. Therefore, secure design must be supported by security activities early in the development lifecycle, including threat modeling, secure implementation, cybersecurity monitoring, secure update mechanisms, continuous testing, and long-term cryptographic agility.

Development teams should follow secure software development practices, including the use of memory-safe languages, performing code reviews, and vulnerability scanning. All wireless-

facing software should be subjected to regular fuzzing, penetration testing, and regression testing, particularly after updates or configuration changes. It is important to also conduct exhaustive security testing of third-party software and OSS components before integration. These activities are essential for identifying and addressing security weaknesses during the development lifecycle. Because many wireless-facing components process attacker-controlled input, fuzzing should be treated as a baseline activity for parsers, protocol handlers, network daemons, update clients, and message-processing services.

Moreover, vehicles must be capable of receiving security updates for wireless components throughout their operational lifetime without introducing new trust assumptions. Employing cryptographic signatures ensures that only authenticated updates are installed, while secure protocols like HTTPS protect updates during distribution. Additionally, restricting access to firmware servers and monitoring them for unauthorized changes is critical to prevent tampering and ensure the authenticity of updates. Collaboration across the automotive industry will also be critical, especially for cybersecurity monitoring by fostering the sharing of vulnerability information. Lifecycle security is especially critical for long-lived vehicles operating in rapidly changing wireless ecosystems, where new threats and attacks continuously evolve.

Summary

Wireless vehicle systems significantly expand the overall attack surface of modern vehicles by introducing multiple external communication interfaces. Over the years, numerous common vulnerabilities have been identified in wireless vehicle systems. These include hard-coded credentials, shared or static cryptographic keys, weak or missing authentication, exposed debug or diagnostic services, and unnecessary network services left enabled in production systems. Some weaknesses also arise from limitations or design flaws in the wireless protocols themselves, which can be exploited even when implementations are correct according to the protocol.

Based on these observations, this chapter introduced common penetration testing techniques used to assess the security of wireless vehicle systems. We discussed practical approaches for discovering exposed interfaces, enumerating services, and identifying misconfigurations or protocol-level weaknesses. Several tools were presented to demonstrate how testers can scan for vulnerabilities, analyze wireless traffic, and actively test different communication technologies in a controlled and systematic manner.

Moreover, to better understand how these techniques can be applied in practice, the chapter also reviewed several real-world case studies in which wireless vehicle systems were successfully compromised. These case studies illustrate how seemingly minor weaknesses can

be chained together, enabling attackers to gain unauthorized access, escalate privileges, or interfere with vehicle functionality.

Finally, the chapter outlined best practices for mitigating common wireless security issues. These include reducing the attack surface by disabling unnecessary services, enforcing strong authentication and authorization, avoiding hard-coded secrets, and following a secure-by-design approach throughout the development lifecycle. By applying these principles early and consistently, manufacturers and developers can significantly improve the resilience of wireless vehicle systems against real-world attacks.

Now that we have concluded our study of the wireless vehicle interfaces, we will move on to examine communications security throughout the in-vehicle network in the next chapter.

References

[1] Quectel Wireless Solutions, "*Quectel EC21 Series LTE Standard Module Specification*," version 2.4, Mar. 2024. [Online]. Available: `https://quectel.com/content/uploads/2024/03/Quectel_EC21_Series_LTE_Standard_Specification_V2.4-1.pdf`

[2] D. K. Oka, T. Furue, L. Langenhop, and T. Nishimura, "*Survey of Vehicle IoT Bluetooth Devices*," in 2014 IEEE 7th International Conference on Service-Oriented Computing and Applications (SOCA), Matsue, Japan, Nov. 2014, pp. 260-264. [Online]. Available: `https://doi.org/10.1109/SOCA.2014.20`

[3] TapTrack, "*NFC Relay Attacks*," Dec. 5, 2016. [Online]. Available: `https://taptrack.com/nfc-relay-attacks/`

[4] C. Ernst, "*AT Commands Guide for GSM*," emnify Blog, Jan. 18, 2022. [Online]. Available: `https://www.emnify.com/blog/at-commands-for-cellular-modules`

[5] 3GPP TS 27.007, "*AT Command Set for User Equipment (UE)*," 3rd Generation Partnership Project, Release 18, 2024. [Online]. Available: `https://www.3gpp.org/ftp/Specs/archive/27_series/27.007/`

[6] E. Tews, R.-P. Weinmann, and A. Pyshkin, "*Breaking 104 Bit WEP in Less Than 60 Seconds*," in Information Security Applications: 8th International Workshop, WISA 2007, Jeju Island, Korea, Aug. 2007, vol. 4867, pp. 188-202. [Online]. Available: `https://doi.org/10.1007/978-3-540-77535-5_14`

[7] IEEE 802.11p-2010, "*IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Amendment 6: Wireless Access in Vehicular Environments*," IEEE, New York, NY, USA, 2010.

[8] F. Klingler, F. Dressler, and C. Sommer, "*IEEE 802.11p Unicast Considered Harmful*," in 2015 IEEE Vehicular Networking Conference (VNC), Kyoto, Japan, Dec. 2015, pp. 76-83. [Online]. Available: https://doi.org/10.1109/VNC.2015.7385550

[9] Kismet Wireless, "*Kismet: Wi-Fi, Bluetooth, RF, and more.*" [Online]. Available: https://www.kismetwireless.net

[10] Aircrack-ng, "*Aircrack-ng*." [Online]. Available: https://www.aircrack-ng.org

[11] iw, "*Linux Wireless Configuration Utility.*" [Online]. Available: https://wireless.docs.kernel.org/en/latest/en/users/documentation/iw.html

[12] G. Lyon, "*Nmap Network Scanner,*" open-source network exploration tool. [Online]. Available: https://nmap.org/

[13] R. D. Graham, "*masscan: Mass IP Port Scanner,*" GitHub repository. [Online]. Available: https://github.com/robertdavidgraham/masscan

[14] Netcat, "*TCP/IP Swiss Army Knife. [Online]*." Available: https://nc110.sourceforge.io/

[15] Great Scott Gadgets, "*Ubertooth*". [Online]. Available: https://github.com/greatscottgadgets/ubertooth

[16] virtualabs, "*btlejack: Bluetooth Low Energy Swiss-army knife*." [Online]. Available: https://github.com/virtualabs/btlejack

[17] BlueZ Project, "BlueZ." [Online]. Available: https://www.bluez.org

[18] Great Scott Gadgets, "*HackRF One - Software Defined Radio*," open-source SDR platform. [Online]. Available: https://greatscottgadgets.com/hackrf/

[19] Great Scott Gadgets, "*Yard Stick One - Sub-1 GHz RF Transceiver*," open-source RF tool. [Online]. Available: https://greatscottgadgets.com/yardstickone/

[20] Ettus Research, "*USRP Software Defined Radio Products*," SDR hardware platform. [Online]. Available: https://www.ettus.com/

[21] GNU Radio, *"GNU Radio - the Free and Open Software Radio Ecosystem*," open-source software radio toolkit. [Online]. Available: https://www.gnuradio.org/

[22] J. Pohl, "*Universal Radio Hacker: Investigate Wireless Protocols Like a Boss*," GitHub repository. [Online]. Available: https://github.com/jopohl/urh

[23] PentestMag, "*RF Exploitation in Automotive Cybersecurity: KeyFob Hacking and Proxmark3*," Oct. 21, 2024. [Online]. Available: https://pentestmag.com/rf-exploitation-in-automotive-cybersecurity-keyfob-hacking-and-proxmark3/

[24] RfidResearchGroup, "*Iceman Fork - Proxmark3*," GitHub repository. [Online]. Available: https://github.com/rfidresearchgroup/proxmark3

[25] emsec, "*ChameleonMini - A Freely Programmable, Portable Tool for NFC Security Analysis*," GitHub repository. [Online]. Available: https://github.com/emsec/ChameleonMini

[26] Wireshark Foundation, "*Wireshark - Go Deep*," network protocol analyzer. [Online]. Available: https://www.wireshark.org/

[27] ReFirmLabs, "*Binwalk - Firmware Analysis Tool*," GitHub repository. [Online]. Available: https://github.com/ReFirmLabs/binwalk

[28] rampageX, "*firmware-mod-kit: Tools for Modifying Firmware*," GitHub repository. [Online]. Available: https://github.com/rampageX/firmware-mod-kit

[29] minicom team, "*minicom*". [Online]. Available: https://salsa.debian.org/minicom-team/minicom

[30] OpenBTS, "*OpenBTS*." [Online]. Available: http://openbts.org

[31] Osmocom, "*Osmocom open-source mobile communications projects*." [Online]. Available: https://osmocom.org

[32] AFLplusplus Team, "*AFL++ - American Fuzzy Lop Plus Plus*," GitHub repository. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus

[33] LLVM Project, "*libFuzzer - A Library for Coverage-Guided Fuzz Testing.*" [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[34] National Security Agency, "*Ghidra Software Reverse Engineering Framework*," GitHub repository. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[35] Hex-Rays, "*IDA Pro - The Disassembler and Debugger.*" [Online]. Available: https://hex-rays.com/ida-pro/

[36] radareorg, "*Radare2 - Unix-Like Reverse Engineering Framework*," GitHub repository. [Online]. Available: https://github.com/radareorg/radare2

[37] curl, "*Command Line Tool and Library for Transferring Data with URLs.*" [Online]. Available: https://curl.se/

[38] Rapid7, "*Metasploit Framework*," penetration testing framework. [Online]. Available: https://www.metasploit.com/

[39] F. D. Garcia, D. Oswald, T. Kasper, and P. Pavlides, "*Lock It and Still Lose It - On the (In)Security of Automotive Remote Keyless Entry Systems*," in Proc. 25th USENIX Security

Symposium, Austin, TX, USA, Aug. 2016, pp. 929-944. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_garcia.pdf

[40] RTL-SDR.com, "*RTL-SDR Blog*," software-defined radio using Realtek RTL2832U chipset. [Online]. Available: https://www.rtl-sdr.com/

[41] S. Gao, Z. Huang, H. Xie, and Y. Zhang, "*All the 4G Modules Could Be Hacked*," DEF CON 27, Las Vegas, NV, USA, Aug. 2019. [Online]. Available: https://www.youtube.com/watch?v=OORUkEsannA

[42] S. Gao, Z. Huang, H. Xie, and Y. Zhang, "*All the 4G Modules Could Be Hacked*," Black Hat USA 2019, Las Vegas, NV, USA, Aug. 2019. [Online]. Available: https://i.blackhat.com/USA-19/Wednesday/us-19-Shupeng-All-The-4G-Modules-Could-Be-Hacked.pdf

[43] J. Pruitt, "*ubi_reader: Python Scripts for Working with UBI and UBIFS Images*," GitHub repository. [Online]. Available: https://github.com/jrspruitt/ubi_reader

[44] Nuand, "*BladeRF Software-Defined Radio*," open-source SDR platform. [Online]. Available: https://www.nuand.com/

[45] YateBTS, "*YateBTS - GSM/GPRS Base Station Software*." [Online]. Available: https://yatebts.com/

[46] hashcat, "*hashcat - Advanced Password Recovery*." [Online]. Available: https://hashcat.net/hashcat/

[47] R.-P. Weinmann and B. Schmotzle, "*TBONE - A Zero-Click Exploit for Tesla MCUs*," Kunnamon and Comsecuris, Oct. 16, 2020. [Online]. Available: https://kunnamon.io/tbone/

[48] R.-P. Weinmann and B. Schmotzle, "*TBONE: Attacking Tesla Infotainment via Wi-Fi*," CanSecWest 2021. [Online]. Available: https://www.youtube.com/watch?v=krSj81thN0w

[49] QEMU Team, "*QEMU - A Generic and Open Source Machine Emulator and Virtualizer*." [Online]. Available: https://www.qemu.org/

[50] M. Zalewski, American Fuzzy Lop, GitHub repository. [Online]. Available: https://github.com/google/AFL

[51] NIST, "*CVE-2021-26676 Detail*," National Vulnerability Database. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-26676

[52] NIST, "*CVE-2021-26675 Detail*," National Vulnerability Database. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-26675

[53] Scapy, "*Scapy - Packet Manipulation Program*." [Online]. Available: `https://scapy.net/`. [Accessed: May 2026].

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

6

Attacking the In-Vehicle Communications

There used to be a time when people imagined the in-vehicle network to be virtually unreachable by attackers. But as connectivity interfaces proliferated, the opportunities for breaching the in-vehicle network became abundant. In the most common scenario, attackers only need to establish a foothold in a remotely accessible **Electronic Control Unit** (**ECU**), before they are positioned right at the edge of the in-vehicle network where they can start interfering with the vehicle operation. Direct physical access to the **On-Board Diagnostics** (**OBD**)-II port is another attack vector, where an attacker could gain direct access to the in-vehicle network. The foundational security risks of the in-vehicle CAN bus were first demonstrated systematically in 2010, where an attacker with physical access to the OBD-II port could inject arbitrary CAN messages to control engine, brake, and other vehicle functions across a range of ECU types [1].

The goal of this chapter is to study the common in-vehicle communication protocols such as **Controller Area Network** (**CAN**), **Local Interconnect Network** (**LIN**), FlexRay, and Automotive Ethernet, to show how they can be attacked. To do so, we will examine the various weaknesses of each communication protocol before diving into penetration testing techniques that aim at exploiting such weaknesses. After that, we will explore the attack methods for sniffing, spoofing, and intercepting network communication while highlighting the various tools that can be used to mount such attacks.

To demonstrate the reality of the risks against such networks, we will present several case studies from real-life attacks that have compromised in-vehicle networking protocols. Finally, we draw lessons on security best practices to harden the networking protocols and increase the difficulty of penetration.

In the rest of this chapter, we will cover the following main topics:

- Overview of in-vehicle communications
- Understanding in-vehicle network protocols
- Common weaknesses of in-vehicle communications
- Penetration testing techniques for in-vehicle networks
- Case studies of real-life attacks on in-vehicle communications
- Best practices for securing in-vehicle communications

Overview of in-vehicle communications

As the vehicle **electrical/electronic** (**E/E**) architecture shifts from domain-based network segmentation to a zonal/central compute based, there is a steady shift from legacy networking protocols such as CAN to newer protocols such as **Automotive Ethernet**. But even as the number of CAN-based ECUs goes down, there seems to be no foreseeable architecture where CAN networks will be eliminated. One main reason behind this is the low cost of CAN networks and high reliability. Moreover, their real-time characteristics make them an ideal choice for resource-constrained ECUs that need to communicate reliably, efficiently, and with low latency over the network bus.

What makes **CAN** so important is that it continues to handle critical control and status messages for the engine, transmission, brakes, etc. LIN continues to be used due to it being a low-cost networking option that is suitable for resource-constrained sensors and smart actuators. FlexRay was introduced to meet the deterministic and fault-tolerant communication requirements of safety-critical systems such as active suspension and steer-by-wire, where CAN's event-driven behavior is insufficient. **Automotive Ethernet** in recent years has become the networking technology of choice to handle the large bandwidth needs for software defined vehicles.

Whether it is transferring large software update packages to multiple ECUs, retrieving large diagnostic data from target ECUs or transmitting rich sensor data, Ethernet has established itself as a strong in-vehicle networking choice. Moreover, the maturity of higher layer protocols and infrastructure tools makes it easier to adopt. Protocols such as **Audio Video Bridging** (**AVB**) and **Time-Sensitive Networking** (**TSN**) make it ideal for time-critical communication and bandwidth allocation.

Besides acting as a central backbone to interconnect high-speed domains (such as an infotainment unit, telematics unit, and central gateway), Ethernet is also used for interfacing high bandwidth sensors like **Light Detection and Ranging** (**LiDAR**) and **Cameras**.

Table 6.1 compares key in-vehicle network protocols and highlights their roles:

Protocol	Topology (Speed)	Typical role in E/E architecture
CAN (Classic | FD | XL)	Bus (Up to 1 Mbps Classic; Up to 8 Mbps CAN-FD; Up to 20 Mbps CAN-XL)	Powertrain, Chassis, Body control (real-time control networks)
LIN	Single-wire bus (19.2 kbps typical)	Body electronics, comfort features (low-speed, low-cost networking)
FlexRay	Dual-channel bus or star (10 Mbps)	Time-critical and safety systems (e.g., drive-by-wire in some designs)
Automotive Ethernet	Switched star/tree (100 Mbps, 1 Gbps, 5/10 Gbps)	Vehicle backbone and vehicle computer to high throughput sensors

Table 6.1: Comparison of common in-vehicle network protocols

Understanding in-vehicle network protocols

As with any network, in-vehicle networks consist of several protocols and layers that span from physical wires up to the application protocols. Understanding each in-vehicle network protocol layer is a prerequisite to identifying security weaknesses that will be exploited during the penetration phase. In the following sections, we present a brief overview of each protocol across the various networking layers, also illustrated in *Figure 6.1*. You are encouraged to study this further for the networking protocols of interest.

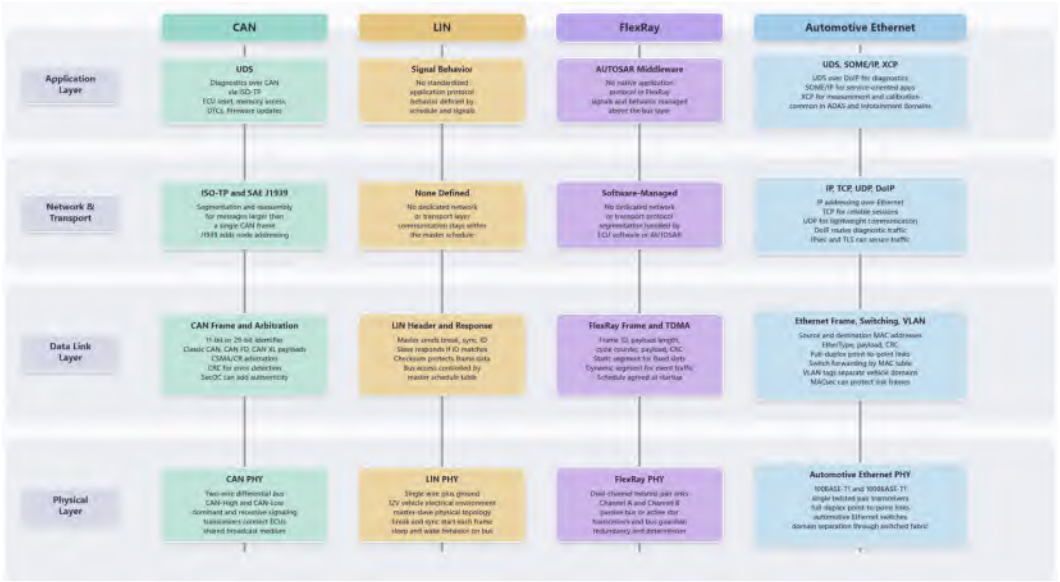


Figure 6.1: Overview of the in-vehicle network layers and technologies

Let's look at these layers in detail:

Physical layer

At the base of all networking layers is the **physical layer**. Each networking protocol must define how electrical signals are represented on the physical medium, and which components are needed to enable reliable communication. Let's see how each in-vehicle protocol defines their physical layer.

Controller area network physical layer (CAN PHY)

CAN uses a two-wire differential bus (CAN-High and CAN-Low) with dominant and recessive signaling. Each ECU connects to the shared medium via a CAN transceiver, which functions as the line driver and receiver for the physical layer, typically incorporating electrical robustness and in some implementations support for wake-on-bus signals.

Local interconnect network physical layer (LIN PHY)

LIN implements a single-wire plus ground topology with a master/slave architecture, where one designated master node controls all communication. Each node connects to the bus through a LIN transceiver, which handles the voltage conversion between the microcontroller and the 12V vehicle electrical system. The master initiates every frame by transmitting a break signal followed by a sync byte, which slave nodes use to synchronize their timing before responding. Power management is also handled at the physical layer; when the bus is idle, all

nodes enter a low-power sleep state and wake automatically when the master resumes activity.

FlexRay PHY

FlexRay uses a dual-channel bus topology, with Channel A and Channel B each carrying signals over a dedicated twisted pair, providing both redundancy and higher bandwidth compared to single-channel protocols. Each node connects to the bus through a FlexRay transceiver, which handles signal conversion and may include a bus guardian function. When implemented, the bus guardian helps ensure that a node transmits only during its pre-assigned time slot, preventing any single node from disrupting communication for others. FlexRay supports both passive bus and active star topologies.

Automotive Ethernet PHY

Automotive Ethernet (based on, e.g., IEEE 100BASE-T1 and 1000BASE-T1) uses specialized single-pair transceivers designed to operate reliably in the electromagnetic environment of a vehicle, supporting cable lengths of several meters between nodes. Unlike traditional multi-pair Ethernet used in enterprise networks, automotive Ethernet transceivers carry full-duplex communication over a single twisted pair. ECUs connect to the network in a point-to-point fashion through automotive-grade Ethernet switches, which direct traffic between specific ports rather than broadcasting it across a shared medium. When properly configured, these switches can support domain separation between different vehicle networks, for example, keeping infotainment and powertrain traffic isolated from each other.

Data link layer

Next, above the physical layer is the **data link layer**. It defines the frame format and media access rules for each protocol. Typically, this layer is implemented as dedicated hardware **intellectual property** (**IP**) inside the **microcontroller unit** (**MCU**) or **System on Chip** (**SoC**) to meet real-time processing demands of in-vehicle communication.

CAN data link

Because all nodes connected to the same CAN bus share the same physical medium, any ECU can transmit frames onto the bus, and all other nodes on that segment can observe them. CAN [2] frames consist of an identifier (either 11-bit in the standard format or 29-bit in the extended format), control bits, a data payload of up to 8 bytes in classic CAN, up to 64 bytes in CAN FD, and up to 2048 bytes in CAN XL, along with a **Cyclic Redundancy Check** (**CRC**) field for error detection. CAN uses **carrier-sense multiple access with collision resolution** (**CSMA/CR**) where multiple nodes may begin transmitting simultaneously and the node with the lowest identifier value wins arbitration while other nodes stop transmitting and retry when the bus becomes idle.

LIN data link

LIN [4] frames consist of a header transmitted by the master node, comprising a break field, a sync byte, and an identifier field, followed by a response containing data bytes and a checksum. Only the master sends headers while LIN slaves respond if their identifier matches. Bus access is entirely orchestrated by the master schedule table.

FlexRay data link

FlexRay [5] frames consist of a header containing a frame identifier, payload length, and cycle counter, followed by a data payload of up to 254 bytes and a CRC for error detection. Bus access is controlled by a **Time Division Multiple Access** (**TDMA**) scheme divided into a static segment, where each node is pre-assigned fixed communication slots, and a dynamic segment that allows event-driven transmission on a first-come, first-served basis. This schedule is configured consistently across participating nodes and remains fixed during operation, ensuring predictable and deterministic communication latency.

Ethernet data link

Automotive Ethernet [6] uses standard Ethernet framing, with each frame containing source and destination **media access control** (**MAC**) addresses, an EtherType field identifying the higher-layer protocol, a data payload, and a CRC for error detection. Because each link operates in full-duplex point-to-point mode between an ECU and a switch port, there is no shared medium and therefore no risk of collisions between nodes. Ethernet supports multiple higher-layer protocols concurrently, including IP, **Address Resolution Protocol** (**ARP**), and **Virtual Local Area Network** (**VLAN**)-tagged traffic, providing a flexible foundation for diverse in-vehicle services. Traffic forwarding decisions are made by the switch based on MAC address tables, and VLAN tagging allows traffic from different vehicle domains to be logically separated when sharing the same physical Ethernet network. In terms of security features, **MACsec** [7] can be applied to provide frame-level encryption and message authentication between directly connected Ethernet nodes.

Network and transport layers

Above the data link layer is the **network and transport layers**, which handles addressing, routing where applicable, and segmentation and reassembly of larger messages. CAN relies on transport overlays (ISO-TP, SAE J1939), while LIN and FlexRay do not define dedicated protocols at these layers, and Ethernet uses standard IP-based networking.

CAN network and transport

ISO-TP (ISO 15765-2 [8]) is a transport layer protocol over CAN that supports the transmission of messages that exceed the maximum CAN frame payload by segmenting large payloads into multiple CAN frames. Many diagnostic protocols are built on top of ISO-TP.

SAE J1939 [9] is a protocol suite built on CAN and widely used in heavy-duty vehicles, defining its own network, transport, and application layer functionality. It introduces a node addressing scheme in which each node is assigned a unique source address used to identify it across the bus, along with transport layer mechanisms for both multi-packet broadcast transfers and peer-to-peer connections.

LIN network and transport

LIN does not define dedicated network or transport layer protocols. Instead, all communication is contained within the master/slave schedule. LIN is typically used for simple, low-speed applications that do not require addressing or message segmentation.

FlexRay network and transport

Similar to LIN, FlexRay also does not define dedicated network or transport layer protocols. Node identification is based on slot assignments in the static communication schedule rather than network addresses. For applications requiring segmentation of payloads larger than a single FlexRay frame, this is handled by the ECU software stack, typically through AUTOSAR communication middleware, which manages segmentation and reassembly above the data link layer.

Ethernet network and transport

ECUs communicating over Ethernet use standard IP-based networking, with IP addresses assigned either statically or dynamically. At the transport layer, both **Transmission Control Protocol** (**TCP**) and **User Datagram Protocol** (**UDP**) are used depending on the requirements of the application: TCP for connection-oriented reliable delivery and UDP for lightweight connectionless communication. In addition, **Diagnostics over Internet Protocol** (**DoIP**) (ISO 13400 [10]) defines how diagnostic messages are routed over TCP and UDP to reach target ECUs. Furthermore, regarding security, at the network layer, **Internet Protocol Security** (**IPsec**) can be used to protect IP-layer communication between endpoints, while at the transport layer, **Transport Layer Security** (**TLS**) can be used to secure application sessions.

Application layer

Finally, the **application layer** sits at the top of the in-vehicle network stack and governs the content and behavior of communications between ECUs. Unlike the lower layers, application layer protocols in automotive networks are not always tied to a single bus technology; that is, several protocols can operate over multiple transport mechanisms.

It is worth noting that LIN and FlexRay do not define standardized application layer protocols; application behavior in these networks is managed through signal and frame definitions specified in configuration files and AUTOSAR middleware, respectively. Therefore, the protocols we focus on in this section are associated with CAN and Ethernet.

Unified diagnostic services (UDS)

UDS (ISO 14229 [11]) is the standard diagnostic request/response protocol used across the automotive industry, operating over CAN via ISO-TP and over Ethernet via DoIP. It provides a wide range of services, including reading diagnostic trouble codes, ECU reset, memory read and write operations, routine control, and firmware update workflows. It is intended for use by authorized diagnostic clients in workshops and production environments.

Scalable service-oriented middleware over IP (SOME/IP)

SOME/IP is a service-oriented communication protocol used in AUTOSAR Adaptive environments over Ethernet, providing service discovery and message serialization between ECUs. It is commonly used for communication between application services in domains such as infotainment and **Advanced Driver Assistance Systems** (**ADAS**).

Universal measurement and calibration protocol (XCP)

XCP [12] is used over both CAN and Ethernet, primarily during vehicle development for live measurement and calibration. XCP offers many useful features such as reading and writing arbitrary memory addresses and supports flash programming, enabling engineers to fine-tune control parameters such as fuel injection timing during runtime.

Secure onboard communication (SecOC)

SecOC [3] is an **AUTomotive Open System ARchitecture** (**AUTOSAR**)-specified security mechanism that provides message authenticity and replay protection for critical **Protocol Data Units** (**PDUs**), independent of the underlying bus protocol. It operates by appending a freshness value and a **Message Authentication Code** (**MAC**) to outgoing PDUs and verifying them on receipt. SecOC can be applied over CAN, CAN-FD, and Ethernet, making it a protocol-agnostic mechanism within the AUTOSAR communication stack.

With the in-vehicle network protocol stack reviewed across the physical, data link, network, transport, and application layers, the next step is to examine the common weaknesses present in these communications and how an attacker can take advantage of them.

Common weaknesses of in-vehicle communications

To perform an effective penetration testing campaign, it is essential to survey the various weaknesses of the target networks and communication protocols. Even today, there are plenty of automotive engineers who still believe the in-vehicle network is a closed, trusted environment. Unfortunately, this creates an exaggerated sense of how difficult it must be for attackers to reach the in-vehicle network.

Legacy protocols like CAN and LIN were designed before connectivity took off, so they are inherently insecure. FlexRay is more modern than CAN and LIN but followed the same pattern:

deterministic and fault-tolerant by design, yet with no native security built in. With the advent of Automotive Ethernet, native security support such as authentication, segmentation, and encryption improved the security posture, but weaknesses can still exist due to configuration and implementation issues. As vehicles evolved from mechanically isolated systems into networked, software-defined platforms with external connectivity, these weaknesses have become exploitable attack surfaces that attackers can target [13, 14].

This section examines those weaknesses systematically, focusing on the two most prevalent and high-risk protocols in detail, i.e., CAN and Automotive Ethernet, followed by LIN and FlexRay. For each protocol, the analysis addresses inherent design limitations, upper-layer communication software weaknesses, and relevant topology considerations that affect the attack surface. An overview of the common in-vehicle network weaknesses is depicted in *Figure 6.2*.

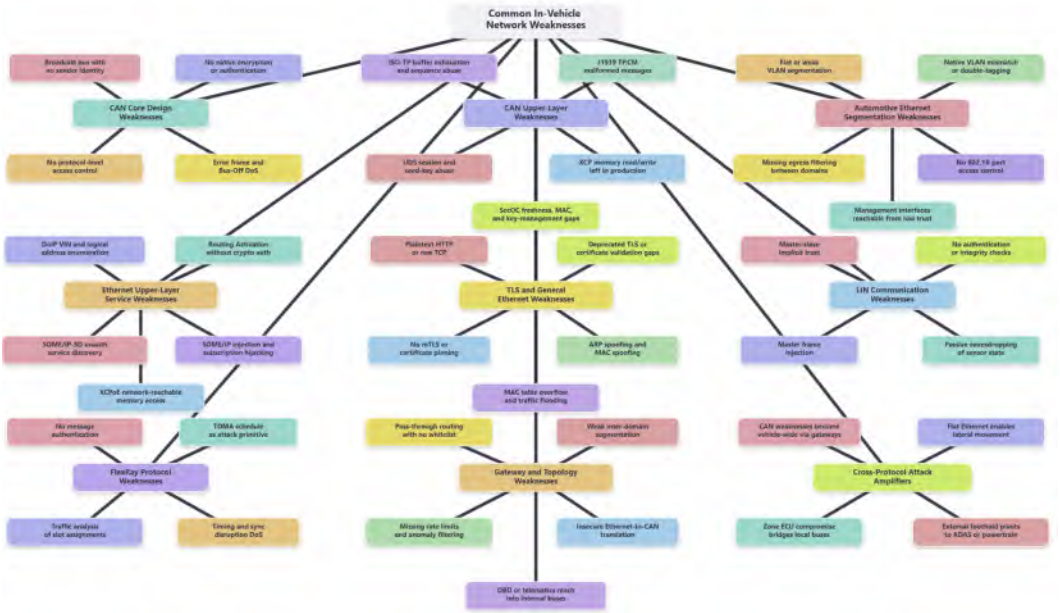


Figure 6.2: Common in-vehicle network weaknesses

CAN

CAN was designed for reliability and determinism in electrically noisy environments and not for security. Design decisions that made CAN robust for its original purpose simultaneously expose weaknesses for attackers who have access to the bus.

Broadcast architecture and lack of source authentication

CAN operates on a shared bus topology where every message transmitted by any node is received by every other node on the same bus segment. There is no concept of source addressing in the CAN data link layer. The CAN frame identifier specifies the message type and not the sender identity. Thus, any connected node can transmit any message identifier, and the receiving nodes have no protocol-level mechanism to verify the legitimacy of the source.

Lack of encryption and authentication support

CAN provides no native encryption and authentication support. All data fields are transmitted in plaintext across the bus. An attacker connected to the physical bus or having access to a compromised node can perform passive monitoring to gain a complete view of all in-vehicle communication on that bus. Moreover, due to the lack of native message authentication support, a compromised node can send spoofed, tampered or replayed messages without the ability of the receiver to discern that the message is not legitimate. In safety-critical domains such as powertrain and chassis, this translates to operational risks for ECUs consuming network data such as engine speed, throttle position, brake pressure, steering angle, and braking commands.

No access control

There is no built-in access control mechanism on CAN. That is, there is no whitelist of permitted senders per message identifier, no role-based filtering, and no mechanism to prevent a node from transmitting message identifiers outside its intended functional scope. This means that a compromised ECU or a malicious node that an attacker has clipped onto the bus can transmit arbitrary messages without any protocol-level defense to stop it.

Error frame and bus-off mechanism

CAN's error handling mechanism, while designed to ensure reliability, introduces an availability weakness. Each CAN node maintains a **Transmit Error Counter** (**TEC**) and **Receive Error Counter** (**REC**). When these counters exceed defined thresholds, the node transitions through **Warning** and **Error Passive** states. Further, only when TEC exceeds 255 does the node transition to the **Bus-Off** state, at which point the node is disconnected from the bus and ceases all communication. An attacker can deliberately trigger this state by causing the target node to repeatedly perceive its own transmissions as faulty and consequently increment its TEC error counter, eventually leading to Bus-Off [15]. Consequently, a Bus-Off attack against a safety-critical ECU such as an **Anti-lock Braking System** (**ABS**) controller or an airbag control module could lead to a denial-of-service attack with safety implications.

ISO-TP upper-layer weaknesses

ISO-TP is the transport protocol layer used on CAN to support messages exceeding a single CAN frame payload limit (8 bytes on CAN 2.0, and up to 64 bytes on CAN FD). ISO-TP introduces its own set of weaknesses at the communication transport layer.

ISO-TP defines four frame types:

- **Single frame**: payload up to 7 bytes when transmitted in a classic CAN frame; payload up to 62 bytes when transmitted in a CAN FD frame
- **First frame**: initiates a multi-frame transmission, including total message length
- **Consecutive frame**: carries subsequent segments with a sequence number
- **Flow control**: issued by the receiver to regulate the sender's transmission rate and pacing

The following weaknesses are relevant from a security perspective:

- **Buffer exhaustion via first frame manipulation**: An attacker can transmit a First Frame advertising a large message length (up to 4095 bytes in the classical 12-bit First Frame length format, or larger in later ISO 15765-2 versions) without following up with Consecutive Frames. The receiving ECU allocates a reassembly buffer and holds it open waiting for further segments. An attacker can send multiple such incomplete transmissions to resource-constrained ECUs to exhaust available memory.
- **Sequence number manipulation**: Consecutive Frames carry a 4-bit sequence number (0x0 to 0xF). An attacker can inject a Consecutive Frame with an incorrect or unexpected sequence number during an ongoing multi-frame transmission, which may cause the receiver to abort the session, corrupt the reassembly buffer, or in poorly implemented stacks, trigger undefined behavior.
- **Flow control frame injection**: An attacker can inject a spoofed Flow Control frame to disrupt an ongoing diagnostic session. For example, an attacker can spoof a Flow Control frame with a **Separation Time Minimum** (**STmin**) value of 0x7F (127 ms), causing the legitimate sender to insert a 127 ms delay between every Consecutive Frame, reducing throughput to near-zero and effectively stalling session.

SAE J1939 upper-layer weaknesses

SAE J1939 provides a standardized protocol for commercial vehicle communication, including a transport protocol layer designed to handle multi-packet messages exceeding eight bytes. However, several key weaknesses exist at the upper protocol layers [16].

Some examples include the following:

- **Buffer exhaustion via malformed TP.CM_BAM messages**: This broadcast announce message (**TP.CM_BAM**) instructs receiving ECUs to pre-allocate a buffer for an incoming multi-packet transmission. An attacker injecting a crafted TP.CM_BAM frame declaring an excessively large message size can force the target ECU to over-allocate memory, leading to exhaustion and potentially a crash, without requiring any prior authentication.
- **Out-of-range packet count manipulation via TP.CM_RTS**: An ECU receiving a request-to-send (**TP.CM_RTS**) message may process the declared packet count without adequate bounds checking. Thus, an attacker sending a crafted TP.CM_RTS frame with an out-of-range total packet count can cause the ECU to allocate or reference memory beyond intended boundaries, resulting in memory corruption or a crash of the target ECU.
- **Lack of sender authentication**: J1939 inherits the broadcast nature of CAN and provides no mechanism to authenticate the source address of a frame. Thus, an attacker with bus access can transmit control frames to any target ECU while impersonating a legitimate network node.

UDS application-layer weaknesses

While UDS provides various diagnostic services useful for analysis and diagnosing issues on ECUs, there are several key weaknesses. Some examples include the following:

- **Security access brute-forcing**: **Security access (Service 0x27)** uses a seed-key challenge-response mechanism. If the implementation uses static, weak, or deterministic key derivation algorithms, an attacker can brute-force or predict the correct key response. Moreover, many production ECUs implement insufficient attempt-limiting, allowing attackers to continuously repeat the seed-key challenge-response with no lockout.
- **Authentication bypass and certificate validation failures**: **Authentication (Service 0x29)**, where implemented, uses **Public Key Infrastructure** (**PKI**)-based certificate exchange to identify the client and grant access rights. If certificate signature and chain validation, expiry or revocation checking, proof-of-ownership verification, or binding of authenticated roles to permitted services is weak, an attacker may authenticate with an unauthorized certificate or retain broader access than intended.
- **Unauthenticated or under-authenticated session escalation**: **DiagnosticSessionControl (Service 0x10)** allows transition to Extended Diagnostic Session or Programming Session. On many ECUs, these sessions can be entered

without sufficiently strong prior authentication, providing access to sensitive services such as data identifier read/write (**Services 0x22/0x2E**), memory read/write by address (**Services 0x23/0x3D**), routine control (**Service 0x31**) to trigger predefined routines that may affect safety, and ECU reset (**Service 0x11**). With access to these services, an attacker can read and overwrite sensitive data, invoke manufacturer-defined routines including flash erase, boot mode transitions, or actuator tests, or repeatedly reboot an ECU to create a denial-of-service condition, depending on the specific services left unprotected.

- **Unsafe download and programming path**: **RequestDownload, TransferData, and RequestTransferExit (Services 0x34, 0x36, and 0x37)** provide a standardized mechanism to write data into an ECU. If programming session entry, authentication, address-range validation, integrity checks, or signature verification are weak, an attacker may upload malicious firmware, calibration data, or configuration data. In implementations that couple the download sequence with manufacturer-defined activation routines, this can result in persistent code execution.
- **Communication suppression and fault hiding**: **CommunicationControl (Service 0x28)** can disable ECU communications, **ControlDTCSetting (Service 0x85)** can suppress diagnostic trouble code updates, and **TesterPresent (Service 0x3E)** can keep a privileged session alive indefinitely. If these services are insufficiently protected, an attacker can reduce network visibility, inhibit fault recording, and extend the time window for further malicious actions.

XCP communication weaknesses

XCP was designed as an engineering tool protocol and should be disabled in production vehicles. If XCP is left behind inadvertently, without proper authentication, it can be used to perform invasive operations like modifying memory contents on the fly, which can result in arbitrary code execution. It operates over multiple transport layers including **XCP on CAN** (**XCPoC**) and **XCP on Ethernet** (**XCPoE**), making it relevant across both networks.

Key weaknesses in XCP include the following:

- **Unauthenticated connection and memory access**: XCP's **CONNECT** command initiates a session with no authentication requirement in the base specification. Once connected, a master can issue **SHORT_UPLOAD** to read arbitrary ECU memory regions and **SHORT_DOWNLOAD** or **DOWNLOAD** to write to them. Thus, an attacker is allowed unrestricted memory read/write capability over the CAN bus if XCP is not disabled in production ECUs.
- **Seed-key protection missing or weakly implemented**: XCP defines an optional seed-key protection mechanism for privileged resources. However, it is frequently disabled

during development and testing and therefore inadvertently left as disabled in production, and even if it is enabled, in many implementations it uses weak or publicly known key algorithms. An attacker who is able to obtain the seed-key algorithm, which often can be extracted from the ECU firmware, can unlock all protected resources.

- **Measurement data exfiltration**: XCP's **Data Acquisition Mode** (**DAQ**) allows continuous streaming of ECU internal variables. If supported, this provides an attacker with a persistent, high-frequency read channel into ECU internal state.

SecOC implementation weaknesses

SecOC is the primary mechanism for adding message authentication to CAN communication, by appending a MAC and a freshness value to each secured PDU. For CAN 2.0 frames and even CAN FD frames, both MAC and freshness value are usually truncated to make room for the payload data. A set of pre-shared keys among the sender and receiver ECUs is used to generate and verify the MACs. While SecOC is a mitigation for unauthenticated communication rather than an inherent protocol weakness, it introduces a new attack surface where SecOC implementation mistakes can lead to weaknesses.

Examples of such implementation weaknesses include these:

- **Truncated MAC with insufficient bit length**: SecOC allows MAC truncation to as few as 24 bits to accommodate CAN's payload constraints. The AUTOSAR specification acknowledges that MAC truncation reduces the security level and recommends a minimum of 64 bits for sufficient protection against guessing attacks. A 24-bit MAC provides only 16 million possible values, meaning the security of each authenticated message depends heavily on the correct implementation of the freshness value to prevent repeated guessing attempts. In deployments where freshness value management is weak or missing, the reduced MAC space becomes a practical attack surface.
- **Freshness value rollover and counter exhaustion**: SecOC freshness values prevent replay attacks by ensuring each authenticated message carries a unique counter value for the lifetime of the MAC key. Because SecOC relies on long-term pre-shared MAC keys, the freshness counter must never roll over as long as those keys remain in use. If a protocol weakness or implementation flaw allows an attacker to fast-forward the counter, i.e., to force rapid counter advancement without legitimate message traffic, the counter can be exhausted prematurely. Once rollover occurs, an attacker can replay previously captured authenticated frames sharing the same counter value and key, effectively defeating the anti-replay protection that SecOC is designed to provide.
- **Verification bypass in receiver implementations**: In some production implementations, SecOC verification failures result in the PDU still being passed to the

application layer but with a verification failure flag, rather than being discarded. If the application does not check this failure flag and handles it appropriately, which was a common oversight in initial SecOC integrations, an attacker can inject unauthenticated messages that are still processed by the target application as if verification succeeded.

- **Key management and symmetric key exposure**: SecOC relies on symmetric keys shared between communicating ECUs. If, e.g., the same keys are reused across multiple ECUs, or vehicle variants or model years, the compromise of a single ECU's key material, for instance, through firmware extraction or side-channel analysis affects all ECUs sharing that key. This is a key management weakness rather than a SecOC weakness per se, but it is imperative that organizations apply proper measures for key management.

Automotive Ethernet

Automotive Ethernet has become the backbone network for bandwidth-intensive domains in modern vehicles. Unlike CAN, Ethernet provides sufficient bandwidth to support standard IT security protocols such as TLS. However, the transition from CAN-centric architectures to Ethernet-based zone or domain architectures has introduced a new and broader attack surface, particularly where security has not been considered from the start [17].

Misconfiguration or lack of network segmentation

Automotive Ethernet networks are logically organized using VLANs (IEEE 802.1Q) to separate traffic between functional domains, for example, isolating ADAS sensor traffic from infotainment or telematics. If VLAN segmentation is misconfigured or lacking, the network could effectively operate as a flat broadcast domain similar to CAN. Thus, an attacker compromising any node on the network could gain visibility into traffic from all functional domains and the ability to inject packets targeting any connected ECU.

Examples of misconfigurations that can lead to weaknesses include the following:

- **Native VLAN mismatches at trunk ports enabling VLAN hopping**: In 802.1Q trunking, the native VLAN carries untagged traffic. If two connected trunk ports are configured with different native VLANs, an untagged frame sent from one side is assigned to the wrong VLAN on the receiving side, causing it to cross a VLAN boundary without passing through a router or gateway and therefore effectively bypassing any inter-VLAN access control policy entirely.
- **Double-tagging enabling VLAN hopping**: In 802.1Q trunking, an attacker can craft frames with two VLAN tags, where the outer tag matches the native VLAN and the inner tag targets a different VLAN. The first switch strips the outer tag and forward the frame as native traffic, while the second switch processes the inner tag and delivers it

to the target VLAN, bypassing inter-VLAN access controls. This could allow a compromised node in a lower-trust domain such as infotainment to inject traffic into a higher-trust domain such as ADAS or powertrain.

- **Management interfaces accessible from guest or untrusted VLANs**: Switch and gateway management interfaces, typically used for configuration, firmware updates, and monitoring, should only be reachable from a dedicated, restricted management VLAN. However, if these interfaces are accessible from lower-trust VLANs such as the infotainment or telematics domain, a compromised node on those domains could directly interact with the management interfaces.
- **Missing or permissive egress filtering between domain segments**: Without explicit egress filtering rules at domain boundaries, traffic originating from a lower-trust segment can reach higher-trust segments unchecked. For example, a compromised infotainment ECU could send packets directly to ADAS or powertrain domain if the gateway applies no outbound filtering on the infotainment segment's traffic.
- **Missing port-based access control (IEEE 802.1X) on Ethernet switch ports**: 802.1X requires a device to authenticate before being granted network access at the port level. If it is not enabled, any device physically connected to an Ethernet port is granted immediate network access with no identity verification. An attacker with physical access to the vehicle internals, e.g., via the OBD-II connector or directly tapping an Ethernet connection in the wiring harness, can connect an attacker-controlled device and gain full network access, similar to a CAN bus clipping attack.

Note

Terms:

- 802.1Q trunking allows a single Ethernet link to carry traffic from multiple VLANs simultaneously by adding a 4-byte tag to each frame identifying which VLAN it belongs to, allowing network switches to correctly separate and forward traffic to the appropriate VLAN.
- The native VLAN is the VLAN assigned to any untagged frames arriving on a trunk port. When a frame with no 802.1Q tag is received, the switch automatically associates it with the native VLAN.
- A trunk port is a switch port configured to carry tagged traffic from multiple VLANs simultaneously and are typically used to interconnect switches or to connect a switch to a router or gateway.

- Untagged traffic are Ethernet frames that carry no VLAN tag. On a trunk port, untagged frames are automatically assigned to the native VLAN.
- Double-tagging is a technique where an Ethernet frame is encapsulated with two 802.1Q VLAN tags, an outer tag and an inner tag, where the outer tag is stripped by the first switch and the inner tag is then processed by the second switch, allowing the frame to be delivered to a different VLAN than the one the sender is authorized to access.
- VLAN hopping is an attack technique where an attacker sends specifically crafted frames to cross from one VLAN into another without passing through a router or gateway, bypassing any inter-VLAN access control that would otherwise be enforced at that boundary.
- IEEE 802.1X is a port-based network access control standard that requires a device to authenticate before being granted access to a network. Until authentication succeeds, the port is blocked and no network traffic is permitted.
- The OBD-II connector is a standardized port that provides physical access to the vehicle's diagnostic interface.

DoIP upper-layer weaknesses

DoIP allows UDS diagnostic sessions over Ethernet, replacing the CAN-based ISO-TP for Ethernet-connected ECUs. DoIP is typically accessible on a dedicated diagnostic VLAN and reachable via the OBD-II connector or remotely via the telematics unit.

Relevant weaknesses include the following:

- **Unauthenticated vehicle identification response**: The UDP **Vehicle Identification Response** discloses the **Vehicle Identification Number (VIN)**, **Entity Identifier (EID)**, and logical address of reachable ECUs to any host on the network without requiring any prior authentication. An attacker can use this to perform the reconnaissance step to gain a complete map of accessible target vehicle systems.
- **Logical address enumeration**: The 16-bit logical address space (0x0000–0xFFFF) used to address individual ECUs or diagnostic functions is small enough that it can be enumerated exhaustively. That is, an attacker can perform a bruteforce attack by

iterating through the address space sending a **TesterPresent** request (**Service 0x3E**) to each address and observe responses to map the complete set of reachable ECUs.

- **Routing Activation without cryptographic authentication**: The **Routing Activation Request** supports an **Original Equipment Manufacturer** (**OEM**)-specific authentication byte field, but its implementation is optional and often missing in production systems. As a result, if it is not configured, any host that can reach TCP port 13400 can activate a routing connection and initiate UDS sessions.

Note

Terms:

- Entity Identifier (EID) is a unique identifier assigned to a DoIP entity, typically a network node such as a gateway or ECU, which is used to distinguish it from other entities on the vehicle network. It is commonly derived from the MAC address of the Ethernet interface.
- Logical address is a 16-bit identifier assigned to each ECU in a DoIP network, used to route diagnostic messages to the correct target. Unlike an IP address which identifies a network interface, a logical address identifies a specific ECU or diagnostic function within the vehicle's diagnostic architecture.
- A Routing Activation Request is a DoIP message sent by a diagnostic client to a DoIP gateway to establish a diagnostic routing connection over an already established TCP session. Once the gateway confirms the routing activation with a positive response, the client is permitted to send diagnostic messages targeting specific ECUs behind the gateway.

SOME/IP middleware weaknesses

SOME/IP supports service-oriented communication in Ethernet-connected vehicle architectures. Key weaknesses include the following:

- **SOME/IP-SD service discovery without access control**: **SOME/IP Service Discovery** (**SOME/IP-SD**) uses multicast UDP to announce and find services. As such, any node on the network can send a **FindService** message and receive **OfferService** responses from all visible services. There is no authentication or authorization mechanism specified in the SOME/IP-SD specification. As a result, an attacker with network access can enumerate all available services, their interfaces, and their port bindings without any credentials.

- **Lack of message authentication**: SOME/IP messages have no native message authentication protection. Consequently, an attacker can inject spoofed **Request** messages or **Notification** messages targeting any discovered service. Since the receiving service cannot verify the origin of a message, it will process the injected message as if it came from a legitimate sender, potentially triggering unintended behavior in the receiving application.
- **Subscription hijacking**: The publish-subscribe mechanism does not authenticate subscribers. Therefore, an attacker can register as a subscriber to high-frequency sensor event streams such as camera feeds or LiDAR feeds, gaining access to a continuous data feed.
- **Malformed payload handling**: SOME/IP serialization is a frequent source of implementation vulnerabilities. For example, oversized length fields, type confusion in variant fields, and malformed **Type-Length-Value** (**TLV**) encodings have been shown to trigger buffer overflows and denial-of-service conditions in SOME/IP stacks.

Note

Terms:

- A Request message (Message Type 0x00) is sent by a SOME/IP client to call a method on a remote service, expecting a response. It is the primary mechanism for remote procedure calls in SOME/IP service-oriented communication.
- A Notification message (Message Type 0x02) is sent by a SOME/IP service to all registered subscribers when an event occurs or a value changes. It is used in the publish-subscribe communication pattern and requires no prior request from the receiver.
- Type confusion is a vulnerability that occurs when a program processes data as a different type than what was originally intended. An attacker can declare one data type in the type identifier and provide data structured as a different type, which would cause the receiving implementation to misinterpret the data, potentially leading to memory corruption or unexpected program behavior.
- Type-Length-Value (TLV) is a data serialization format where each field in a message is described by three components: a Type identifier indicating the data type, a Length value specifying how many bytes of data are included, and the Value which contains the actual data. An attacker can exploit TLV parsing by declaring a Length value that does not match the actual bytes provided, for

example, declaring 65,536 bytes while only sending 8 bytes, which can cause unintended behavior.

Lack of or misconfiguration of TLS

Unlike CAN, automotive Ethernet has the bandwidth to support TLS. However, TLS is either not used, configured with deprecated cipher suites, or implemented with improper certificate validation logic. Common misconfigurations include the following:

- **Services accessible over plaintext HTTP or raw TCP where HTTPS/TLS is feasible**: Running diagnostic, configuration, or **Over-the-Air (OTA)** update services over unencrypted HTTP or raw TCP exposes all communication to any node on the same vehicle network segment.
- **TLS 1.0 or 1.1 accepted due to legacy client compatibility requirements**: TLS 1.0 and 1.1 are deprecated versions with known cryptographic weaknesses. If these versions are supported due to legacy reasons, an attacker may force a downgrade from TLS 1.2 or 1.3, and attack the weaker protocol version.
- **Self-signed certificates with no root of trust or no revocation checking**: In a vehicle PKI, ECU certificates should be issued by a trusted Certificate Authority, which allows establishing a verifiable chain of trust. On the other hand, **self-signed certificates** provide no such security assurance, and without revocation checking, a certificate belonging to a compromised ECU would remain valid indefinitely.
- **Lack of certificate pinning, enabling man-in-the-middle attacks from compromised gateway nodes**: Certificate pinning ties an ECU to the specific certificate it expects from a given communication partner, rejecting any other certificate even if it is otherwise valid. Without certificate pinning, a compromised gateway can establish separate TLS sessions with two communicating ECUs by presenting its own valid certificate to each side, and then decrypting, reading, and potentially modifying all traffic between them, with neither ECU being able to detect the attack.
- **Lack of mutual TLS, allowing any client to connect to sensitive service endpoints**: Standard TLS only authenticates the server to the client, not the other way around. In an Automotive Ethernet context where ECU-to-ECU communication should be restricted to authorized nodes, the lack of **mutual TLS (mTLS)** means that an attacker who gains access to the vehicle network can connect to sensitive service endpoints without authentication.

Note

Terms:

- A self-signed certificate is a digital certificate signed by the same entity that issued it, rather than by a trusted Certificate Authority. Since a trusted Certificate Authority has not verified the certificate, the receiving party has no way to confirm that it is communicating with the intended entity.
- Revocation checking is the process of verifying that a certificate has not been revoked before trusting it. For example, a certificate may be revoked if the corresponding private key has been compromised or the issuing Certificate Authority determines it should no longer be trusted.

XCP over Ethernet weaknesses

The common XCP weaknesses described in the CAN section also apply to XCPoE. It is worth noting that Ethernet's network reach further expands the attack surface, and as such, additional Ethernet-specific weaknesses for XCPoE include the following:

- **Network-reachable unauthenticated XCP slave**: An XCPoE slave accessible on the vehicle's Ethernet backbone, or reachable through a misconfigured gateway from the telematics or OBD interface, exposes full ECU memory read/write capability to any host that can reach the UDP or TCP port. In such configurations, the physical bus access prerequisite that limits exploitation of XCPoC no longer applies, making remote exploitation more practical.
- **No transport-layer authentication in base specification**: The XCPoE specification does not mandate TLS or any transport-layer security. In practice, XCPoE connections often run over plaintext UDP or TCP with no encryption or session authentication, making them vulnerable to both passive eavesdropping of measurement data and active injection of commands from an attacker connected on the same network segment.
- **Broadcast-based slave discovery**: XCPoE supports slave discovery via UDP broadcast, allowing an attacker to enumerate all XCP-capable ECUs on the network without prior knowledge of their IP addresses. The information gathered provides the attacker with a clear picture of available targets before proceeding with connection and memory access attempts.

Note

Terms:

In XCP, communication follows a master-slave model. The XCP slave is the ECU exposing its internal memory and calibration data through the XCP interface. The XCP master is the client, typically a calibration tool or potentially an attacker-controlled node, that initiates the session and issues commands to read from or write to the slave's memory.

Other common Ethernet weaknesses

Beyond the automotive-specific weaknesses described above, several general Ethernet protocol weaknesses are relevant to in-vehicle Ethernet networks and are worth noting:

- **ARP spoofing**: ARP, which maps IP addresses to MAC addresses within a local network, performs no authentication of the replies it receives. Any device on the network can send a fake ARP reply associating its own MAC address with the IP address of another device, causing other nodes to update their ARP cache with incorrect mappings. In an automotive Ethernet network, this allows an attacker with network access to position themselves between two communicating ECUs or between an ECU and a diagnostic system, enabling interception or modification of packets in transit without the communicating parties being aware.
- **MAC spoofing**: Ethernet networks identify devices by their MAC address, and access control mechanisms such as MAC filtering rely on this identifier. Because MAC addresses are software-configurable on most network interfaces, any device can present an arbitrary MAC address. In an automotive Ethernet network, this allows an unauthorized device to impersonate a legitimate ECU or diagnostic tool, potentially bypassing access controls and gaining access to vehicle subsystems or safety-critical ECUs that would otherwise be restricted.
- **MAC table overflow**: Ethernet switches maintain a Content Addressable Memory (CAM) table that stores mappings between MAC addresses to physical switch ports to ensure traffic is forwarded only to its intended destination. When the table is full, most switches enter a fail-open mode in which all incoming frames are broadcast to every port. In an automotive Ethernet network, this loss of traffic isolation means that communications between safety-critical ECUs, normally confined to specific network segments, become visible to all devices connected to the switch, exposing control messages and diagnostic traffic to unauthorized observers.

LIN

LIN is primarily used for body domain functions such as seat positioning, mirror adjustment, and window control. While its limited speed and narrow physical attack surface make it less prominent than CAN or Ethernet from a security perspective, it becomes relevant once an attacker has gained access to the vehicle network and can pivot to target LIN-connected devices.

Implicit master trust and lack of authentication

LIN's master-slave model assumes that the master is trusted by definition. That is, slave nodes respond to frame headers issued by the master without any cryptographic verification of the master's identity. As such, an attacker who can gain access to the LIN bus, either by compromising the master ECU, which is typically connected to the CAN network, or by physically tapping the single-wire bus, can impersonate the master and send arbitrary frame headers to which slave nodes would respond. As a result, an attacker could read relevant sensors and actuator state data from connected LIN devices and peripherals.

FlexRay

While FlexRay is a deterministic, fault-tolerant in-vehicle network protocol designed for safety-critical applications requiring guaranteed timing, it is not that widely adopted and therefore has a limited attack surface. Despite being designed for safety-critical applications, FlexRay has similar security limitations as CAN.

Lack of message authentication

FlexRay frames contain no authentication and no source identity fields. As such, the protocol provides no assurance that the content of a message originates from the expected sender or has not been tampered with. Therefore, an attacker who has gained access to the FlexRay bus can spoof frames with arbitrary payloads that the receiving ECU would process as is.

TDMA schedule as an attack primitive

FlexRay's TDMA schedule is statically pre-defined and can be obtained either from network description files such as **Field Bus Exchange** (**FIBEX**) or **AUTOSAR XML** (**ARXML**) extracted from development environments or reconstructed through passive traffic analysis by observing slot assignments across sufficient captured communication cycles. Armed with the knowledge of this schedule, an attacker can time message injections precisely to target specific communication slots with the intention to override legitimate values with attacker-chosen values, which could influence vehicle behavior. Furthermore, disrupting the FlexRay clock synchronization mechanism, for example, by transmitting frames with incorrect cycle counter values, can cause cluster nodes to declare synchronization errors and transition to a safe state,

resulting in a denial of service that disrupts safety-critical systems such as steer-by-wire or brake-by-wire.

Network topology considerations

The protocol weaknesses described in previous sections do not exist in isolation. Their practical exploitability is shaped by the network topology where these protocols are deployed and the trust boundaries enforced or not enforced between them.

Weak network segmentation and lack of enforcement boundaries

In older, domain-based architectures with minimal segmentation, multiple CAN buses and connected ECUs may share a gateway with limited filtering capabilities. If the gateway does not enforce proper whitelist-based message routing, i.e., permitting only expected message IDs to cross domain boundaries, a compromised node in one domain can send messages that propagate through the gateway to target ECUs in other domains.

Moreover, flat or poorly segmented Ethernet backbones compound this problem significantly. Without VLAN enforcement and inter-VLAN routing policies at the central gateway or switch, a compromised infotainment system could, for example, directly send messages to an ADAS ECU or a domain controller managing powertrain CAN traffic.

Gateway and domain controller trust boundaries

Gateways and domain controllers are the primary trust boundary enforcement points in vehicle network architectures. They play a major role in whether protocol-level weaknesses in individual bus segments can be chained into attacks that can pivot and target other systems beyond their initial entry point.

Relevant concerns include the following:

- **Message ID whitelisting vs. pass-through routing**: A gateway that passes all CAN message IDs between network segments without filtering provides no isolation between domains. To address this concern, **whitelisting** message IDs, i.e., only allowing specified message IDs to be routed to the expected destination domain is a fundamental control that limits cross-domain message injections.
- **Rate limiting and anomaly-based filtering**: Even with message ID whitelisting enabled, an attacker can, for example, perform a denial-of-service attack by replaying high-volume, high-rate traffic of valid message IDs. Providing **rate limiting** capabilities at the gateway, combined with, for instance, timing anomaly detection features, allows for addressing this concern.
- **Ethernet-to-CAN translation security**: In zone architectures, a **zone ECU** translates communication between an Ethernet backbone and local CAN or LIN segments. The

translation function is a critical security boundary: a compromised zone ECU can inject traffic into connected CAN or LIN segments to manipulate the intra-zone communication without the **central gateway** being aware. Therefore, zone ECUs should require the same security posture as central gateways.

Note

Terms:

- A zone ECU is vehicle network node that aggregates communication for all ECUs within a defined physical region of the vehicle, such as front, rear, left, or right zones, regardless of their functional domain. Zone ECUs act as local gateways between their attached CAN or LIN segments and the central Ethernet backbone network.
- A central gateway is a central vehicle network node that routes and filters traffic between distinct network segments or domains, such as powertrain, chassis, ADAS, and infotainment. It enforces trust boundaries across the vehicle architecture, controls which messages may traverse segment boundaries, and serves as the primary point through which external interfaces, such as OBD-II and telematics communication, interact with internal vehicle networks.

How topology amplifies protocol weaknesses

It is important to consider that the relationship between **topology** and **protocol weaknesses** is multiplicative rather than additive. For example, a CAN bus with no message authentication has a limited risk if it is physically isolated. However, the same bus becomes a vehicle-wide attack surface if it is reachable from an external-facing interface through a gateway with insufficient filtering capabilities. Similarly, a flat Ethernet backbone transforms individual ECU vulnerabilities into lateral movement opportunities that an attacker can exploit to move from, e.g., the infotainment domain to ADAS or powertrain domain to cause potential safety damage.

The weaknesses identified across in-vehicle network protocols, spanning unauthenticated access, missing message authentication, and exposure through misconfigured gateways, provide the foundation for understanding how a penetration tester can approach these systems in practice. The following section presents the techniques a security tester can use to test for these weaknesses in CAN, Ethernet, LIN, and FlexRay during an assessment of an in-vehicle network.

Penetration testing techniques for in-vehicle networks

Penetration testing of in-vehicle networks shares many of the basic concepts of pentesting a typical IT network. The differences arise due to legacy protocols which may not be familiar to a typical IT pentester. To ensure our penetration testing is effective, we must follow a structured approach that is tailored for in-vehicle networks.

This section organizes testing techniques per protocol, reflecting the same structure used in the previous section, so that each technique can be understood in the context of the weakness it pertains to.

CAN

CAN's broadcast architecture, lack of authentication, and lack of access control make it one of the most accessible in-vehicle networks from a penetration testing perspective. Once access to the bus is established, a wide range of techniques can be applied. Common tools for penetration on CAN include **python-can** [18], **cantools** [19], **SavvyCAN** [20], **candump** [21], **caringcaribou** [22], **BUSMASTER** [23], **Scapy** [24], and **Kayak** [25].

Passive bus monitoring and traffic analysis

Passive monitoring is the foundational first step in any CAN penetration test. Since all CAN traffic is broadcast to all nodes on the bus segment, a device connected to the bus, via an OBD-II interface, a directly tapped physical connection, or a compromised ECU, receives a complete copy of all traffic on the bus. The primary objectives of passive monitoring are bus enumeration and payload reverse engineering. Bus enumeration involves connecting to the CAN bus and logging all observed message IDs, transmission frequencies, and **Data Length Codes** (**DLC**), then comparing to network description files such as **Database CAN** (**DBC**) and **ARXML** to identify undocumented or unexpected messages. For message IDs without available DBC definitions, payload reverse engineering can be performed by applying various physical controls such as steering, throttle, brakes, etc. while monitoring the bus for payload changes.

Message injection and spoofing

Due to CAN's lack of source authentication, any node on the bus can transmit any message ID with any payload, and receiving ECUs have no mechanism to distinguish a legitimate frame from an injected or spoofed one. Targeted signal injection involves identifying a target signal through passive monitoring or CAN DBC analysis, such as a vehicle speed signal or door unlock command, then constructing and transmitting a frame with the corresponding message ID and desired payload value. For sustained state manipulation, an attacker can inject the target

message at a rate higher than the legitimate sender's transmission frequency, causing the receiving ECU to continuously process the attacker's payload instead of the legitimate one.

Community-maintained databases such as **opendbc** [26] provide pre-decoded DBC files for a wide range of production vehicle models, significantly lowering the effort required to map captured CAN traffic to meaningful signals. It is worth noting that AI-assisted signal correlation, applied across multiple messages captured from a live bus, can further accelerate the reverse engineering of signals not covered by existing databases.

Replay attacks

Replay attacks exploit the lack of freshness protection on CAN messages. A pentester can capture a legitimate frame sequence associated with a specific vehicle function, such as a remote keyless entry unlock sequence or a diagnostic session initiation, and retransmit it later to trigger the same function. Moreover, partial replay of a multi-message sequence can leave the target ECU in an undefined intermediate state, as the state machine advances without completing the expected transition, leading to unexpected behavior or potential denial of service. This technique is particularly effective against UDS diagnostic session transitions and body control functions that rely on multi-step activation sequences.

CAN bus fuzzing

Fuzzing targets the input validation and error handling behavior of ECU CAN stacks by systematically transmitting malformed payloads to identify frames that trigger unexpected behavior, such as crashes, memory corruption, or unintended state transitions. There are different approaches to fuzzing. For example, **random fuzzing** transmits frames with randomized payload bytes and can identify ECUs that process frames without adequate range validation. It is an easy technique but often has low efficiency.

On the other hand, **mutation-based fuzzing** starts from a known valid payload and applies systematic bit-flips or byte-substitution mutations, making it significantly more efficient, as such fuzzed frames generally pass initial validation checks and reach further into the ECU's processing logic, where it can find defects.

Furthermore, **generation-based fuzzing** constructs frames according to a model of the target protocol or message format, allowing the fuzzer to generate inputs that are syntactically valid but semantically malformed, maximizing the depth of code coverage and the likelihood of triggering unexpected behavior due to implementation-level defects in the CAN stack or application logic. More details about CAN fuzzing can be found in [27,28].

Bus-off attack

The objective of **bus-off attack** testing is to determine whether a target ECU implements adequate error recovery and plausibility logic to remain resilient against denial-of-service

attacks that target the bus-off mechanism. The bus-off attack exploits CAN's error handling mechanism to force a target ECU into the bus-off state, effectively removing it from the network and causing a denial of service. The attack is executed by transmitting a dominant bit during the acknowledgment slot of a frame being sent by the target ECU, causing it to interpret the interference as a transmission fault and increment its error counter, TEC. Repeating this at sufficient frequency drives the TEC above 255, transitioning the target node to Bus-Off. Successful Bus-Off attacks against safety-critical nodes such as ABS and airbag controllers could have direct safety implications and should thus be considered as part of the penetration testing scope.

ISO-TP testing

ISO-TP testing targets the transport layer segmentation and reassembly logic. The objective of this testing is to uncover implementation errors and insufficient resilience to malformed or unexpected inputs. For example, a buffer exhaustion test can be performed by sending a First Frame advertising the maximum supported message length without following up with consecutive frames. A pentester can repeat this activity with varying source message IDs and monitor the target ECU for degraded response times or failure to handle subsequent legitimate requests.

Moreover, a pentester can perform a consecutive frame sequence injection, which involves inserting a frame with an out-of-sequence counter value during an ongoing multi-frame transmission between two legitimate nodes, and then observing whether the target ECU aborts the session, raises an error, or processes the injected message as part of the reassembly, which would corrupt the entire message.

SAE J1939 transport protocol testing

SAE J1939 testing targets the segmentation and reassembly logic of the J1939 transport layer. A buffer exhaustion test can be performed by transmitting a crafted TP.CM_BAM frame declaring an excessively large message size without following up with the advertised data packets, then monitoring the target ECU for memory exhaustion symptoms such as degraded response times or loss of bus participation. For connection-mode sessions, a pentester can transmit a crafted TP.CM_RTS frame with an out-of-range packet count to a target ECU and observe whether the ECU performs bounds checking or proceeds to allocate memory beyond intended boundaries. Additionally, since J1939 provides no source address authentication, a pentester can transmit spoofed frames using the source address of a legitimate node and verify whether the target ECU accepts and processes the frames.

UDS diagnostic exploitation

The objective for the pentester is to escalate diagnostic session privileges, access protected services, and exercise write or control capabilities that should be restricted to authorized diagnostic tools.

A pentester can perform the following types of testing:

- **Service enumeration**: Transmit requests for each of the 256 possible **UDS service IDs** (0x00–0xFF) in the different diagnostic session and record responses. A positive response (0x40 + service ID) confirms service availability; a negative response with **Negative Response Code** (**NRC**) **0x11 (serviceNotSupported)** confirms the service is not implemented; **NRC 0x7F (requestCorrectlyReceivedResponsePending)** indicates the service exists but requires further session state. Based on the responses, it is possible to build a service availability matrix for all accessible session types.
- **Session escalation without authentication**: Attempt **DiagnosticSessionControl (Service 0x10)** transitions to **Extended Diagnostic Session (sub-function byte 0x03)** and **Programming Session (sub-function byte 0x02)** without prior **Security Access**. Document which session types are accessible without authentication and enumerate the additional services available in each escalated session.
- **Attacking Security Access (Service 0x27)**: Request multiple seeds from the target ECU using **SecurityAccess RequestSeed (Service 0x27, sub-function byte 0x01)** and analyze the collected seed values for weak seed generation characteristics. If the seed is fixed or exhibits low entropy, a previously captured valid key response can be replayed directly to pass the challenge-response. If the seed generation seems sound, the key generation algorithm can be targeted instead. For example, a pentester may be able to determine the relationship between the seed and key by analyzing collected valid seed-key pairs, allowing the correct key to be computed for any received seed. Where the key generation algorithm cannot be determined by observation only, a pentester could attempt to extract it from the ECU firmware or from a diagnostics tool and reverse-engineer it.
- **Data identifier enumeration (Services 0x22/0x2E)**: Enumerate all 65,536 possible **Data Identifiers** (**DIDs**) using **ReadDataByIdentifier (Service 0x22)**. Document which DIDs return data, their data length, and their content. Then attempt **WriteDataByIdentifier (Service 0x2E)** on writable DIDs in privileged sessions to assess write access controls and identify which configuration parameters that can be modified.
- **Memory read/write by address (Services 0x23/0x3D)**: Attempt **ReadMemoryByAddress (Service 0x23)** with varying base addresses and memory

lengths to identify accessible memory regions, including **Random Access Memory (RAM)**, flash, and mapped peripheral registers. Where supported, attempt **WriteMemoryByAddress (0x3D)** in privileged sessions to assess whether arbitrary memory writes are possible. Successful exploitation of these services can enable runtime code modification, configuration tampering, or extraction of sensitive data such as cryptographic keys and configuration parameters.

- **RoutineControl enumeration (Service 0x31)**: Enumerate routine identifiers using **RoutineControl StartRoutine (Service 0x31 sub-function byte 0x01)**. Identify routines that trigger observable vehicle behavior, allow clear fault codes can modify ECU configurations, or can affect safety functions.
- **ECU reset (Service 0x11)**: Attempt ECU reset using soft reset (**sub-function byte 0x01**), hard reset (**sub-function byte 0x03**), and keyOffOnReset (**sub-function byte 0x02**), across all accessible session types. Observe the results and identify which reset types are executable without Security Access. Where possible, this test should be performed under conditions that mimic a vehicle in operation, such as active software execution or simulated driving state, since a reset that succeeds under these conditions should be treated as a valid safety-relevant finding. It is important to note that an attacker may exploit unauthenticated reset capability as part of a denial-of-service attack or to clear diagnostic session state during an attack sequence.

XCP testing on CAN

XCP testing on CAN targets calibration and measurement interfaces on ECUs that have not disabled XCP. A pentester can begin by identifying the XCP slave's CAN ID by scanning for positive responses to a **CONNECT** command. Once connected, **SHORT_UPLOAD** can be used to read arbitrary memory regions of interest, including calibration parameters, stack regions, and memory-mapped peripheral registers. Moreover, **SHORT_DOWNLOAD** for single-frame targeted writes or **DOWNLOAD** for larger block writes can be used to write modified values to calibration parameters.

Where seed-key protection is present but suspected to be weakly implemented, multiple **GET_SEED** requests can be sent, and the collected seed values analyzed for weak patterns. Furthermore, to assess DAQ exposure, a tester can issue a **GET_DAQ_LIST_INFO** command to enumerate configured DAQ lists and their associated measurement variables, then activate a DAQ session using **START_STOP_DAQ_LIST** to confirm whether continuous streaming of ECU internal variables is accessible without authentication.

SecOC bypass testing

SecOC bypass testing assesses whether the SecOC implementation in the target system provides effective protection against message forgery and replay. First, using passive

monitoring, a pentester can identify SecOC-protected PDUs by observing the fixed-length authentication data appended to the payload and determine the MAC bit length in use. For short MACs, particularly 24-bit truncations, a pentester can assess brute-force forgery feasibility by estimating how many injection attempts are required to statistically achieve a valid MAC guess. With a 24-bit MAC space with approximately 16 million possible values, the observed message transmission rate determines how quickly this threshold can be reached. Furthermore, freshness value behavior can be assessed by monitoring counter increments across consecutive frames and ignition cycles. For example, if the counter resets on power cycle, the previously captured authenticated frames may be replayable in a subsequent cycle before the counter advances past the captured value.

In addition, a pentester can also examine how the target ECU handles loss of synchronization of freshness value. For example, an ECU may be configured to transmit only a truncated portion of the freshness counter, such as the 8 least significant bits of a 64-bit value, leaving the receiver to reconstruct the upper bits from its full internal counter. By forcing an ECU off the network and back on, a pentester can observe the resynchronization process and determine whether the implementation enforces strict bounds on acceptable counter ranges or whether the exchange can be manipulated to enable replays or premature counter exhaustion. Moreover, to test for verification bypass, a pentester can inject a SecOC-protected PDU with a valid payload but incorrect MAC and observe whether the target application continues to process the message despite the verification failure. Finally, a pentester can analyze ECU firmware and potentially extract symmetric keys used for SecOC.

Automotive Ethernet

Automotive Ethernet penetration testing covers the network layer, the transport and diagnostic protocols (DoIP, SOME/IP), and the security layer (TLS). Unlike CAN, Ethernet-based testing benefits from more mature IT security tools that can be adapted for the automotive context, though the proprietary nature of some automotive Ethernet protocols (particularly SOME/IP) requires specialized tooling. Common tools include **Wireshark** [29], **Nmap** [30], **tcpdump** [31], **Scapy**, **sslyze** [32], **mitmproxy** [33], and **Ettercap** [34].

Network discovery and traffic capture

The first step in automotive Ethernet testing is establishing a complete picture of the network topology, i.e., which ECUs are reachable, on which VLANs, at which IP addresses, and with which services exposed. A pentester connects to the automotive Ethernet network via a monitor port on a switch or directly via the OBD-II Ethernet interface and captures all traffic using Wireshark with automotive protocol dissectors for DoIP, SOME/IP, and XCP to identify all active IP addresses, VLAN tags, and protocol types present on the network. Moreover, VLAN enumeration involves analyzing VLAN tags observed in captured traffic to identify the VLANs in use across functional domains.

To test for native VLAN mismatches, a pentester can transmit untagged frames from a connected port and observe whether the frames are received on an unintended VLAN on the other side of the trunk port. If so, this confirms a misconfiguration that allows VLAN hopping where traffic crosses a VLAN boundary without routing through the gateway. Additionally, a pentester can assess whether management interfaces are reachable from lower-trust VLANs by attempting to connect to switch or gateway management services from the infotainment or telematics domain. Where physical access to internal Ethernet ports is in scope, connecting an unauthorized device to an unprotected switch port can confirm whether IEEE 802.1X port-based access control is missing, which would then grant the pentester immediate unauthenticated network access.

DoIP enumeration and exploitation

A pentester can begin by sending a DoIP Vehicle Identification Request to the broadcast address on UDP port 13400 and collect Vehicle Identification Response messages to obtain the VIN, EID, and logical addresses of reachable ECUs. This requires no authentication and provides a complete map of DoIP-accessible targets. If the Vehicle Identification Response is missing or returns incomplete address information, logical address enumeration can be performed by brute-forcing through the 16-bit address space and sending a UDS TesterPresent request to each address, identifying live ECUs by their positive acknowledgment responses.

To test for unauthenticated routing activation, a pentester establishes a connection to TCP port 13400 on the DoIP gateway, transmits a Routing Activation Request with no OEM-specific authentication data, and verifies whether the gateway returns a successful activation response. Once routing is activated, the UDS diagnostic exploitation techniques described in the earlier section for CAN can also be applied over DoIP.

SOME/IP testing

A pentester can send a service discovery **FindService** message to the SOME/IP-SD multicast address on UDP port 30490 and capture all **OfferService** responses to build a complete inventory of available services. This activity requires no authentication. To test message injection, a pentester can send spoofed **Request** messages to a discovered service, or spoofed **Notification** messages impersonating a service provider to its subscribers and observe whether the receiving ECU processes the injected messages as legitimate.

Additionally, to test whether subscription hijacking is possible, a pentester can send a **SubscribeEventgroup** message targeting high-value event groups such as camera feeds or LiDAR data streams with the tester's IP address as the subscriber endpoint and confirm whether the target service begins delivering event notifications without any authentication. Finally, SOME/IP service fuzzing can be performed by sending **Request** messages to each discovered service with malformed header fields, such as oversized length values, invalid

message types, or malformed TLV encodings, and then monitoring the target ECU for crashes, error responses, or unexpected behavior.

TLS configuration analysis

TLS testing on Automotive Ethernet assesses whether services that should be protected by TLS are actually enforcing it, and whether the TLS configuration meets current security standards. From the network discovery capture, a pentester can identify TCP services without TLS carrying sensitive information such as diagnostic, configuration, or OTA update data in plaintext. Even if TLS is enabled, the supported protocol versions and cipher suites can be enumerated by probing with ClientHello messages to identify deprecated versions such as TLS 1.0 or 1.1, which may be exploitable through a downgrade attack.

Further, to assess certificate validation, a pentester can provide a self-signed or expired certificate to the target ECU and observe whether the target ECU correctly rejects the connection or incorrectly completes the handshake, confirming whether the target enforces certificate validation. Moreover, lack of certificate pinning can be tested from a compromised gateway node already in the traffic path, which can establish separate TLS sessions with each communicating ECU by presenting its own certificate to each side.

If the handshake is completed successfully without either ECU rejecting the substitute certificate, this confirms that certificate pinning is not enforced, allowing the pentester full plaintext access to the intercepted communication. Finally, lack of mTLS can be confirmed by connecting to an ECU without presenting a client certificate and observing whether the connection is accepted.

XCP over Ethernet testing

A pentester can send an **XCP GET_SLAVE_ID** broadcast on UDP port 17525 to enumerate all XCP-capable ECUs on the network and their transport endpoint details, requiring no prior knowledge of target IP addresses. For ECUs that do not respond to broadcast discovery, a targeted port scan across UDP and TCP port 17525 on the discovered IP address range can reveal additional slaves. Once a slave is identified, a TCP or UDP session can be established using the **CONNECT** command to enumerate the slave's capabilities. If transport-layer security is not enforced, the communication is over plaintext, allowing the pentester to perform passive eavesdropping to capture measurement data.

The pentester can also inject commands to read and write memory using **SHORT_UPLOAD**, **SHORT_DOWNLOAD**, and **DOWNLOAD** described in the CAN-based XCP testing section. Note that over Ethernet, the 8-byte CAN payload constraint is no longer a concern, and therefore larger data transfers per transaction are possible, making bulk memory reads and writes significantly faster and more practical than over CAN.

LIN

LIN penetration testing is constrained by the physical access requirement and the protocol's narrow attack surface relative to CAN and Ethernet. The techniques below assume either direct physical access to the single-wire LIN bus via a LIN-to-USB or LIN-to-CAN adapter, or that the LIN master ECU has been compromised through lateral movement from CAN or Ethernet, from which the security of the LIN-connected slave nodes can then be assessed.

Passive monitoring and frame analysis

A pentester can capture all LIN frames to analyze the frame schedule, the sequence of **Protected Identifiers** (**PIDs**) issued by the master and the corresponding slave responses. This allows us to build a map of active PIDs, their response data length, and transmission timing, and correlate observed PIDs with vehicle behavior to identify body function control frames.

Master impersonation and frame injection

Since LIN slaves respond to any valid frame header without authenticating the master's identity, a pentester can impersonate the master ECU and receive responses from any slave by transmitting frame headers for target PIDs identified during passive monitoring. Additionally, for PIDs where the slave normally provides the response to the master, a pentester can inject a spoofed response before the legitimate slave can respond. Since LIN uses a single-wire bus with no collision detection, the first response on the wire wins, allowing the pentester to suppress the legitimate slave response by providing an attacker-controlled payload.

FlexRay

FlexRay penetration testing is the most specialized of the four protocols covered in this chapter. The TDMA-based schedule, the dual-channel configuration, and the requirement for cycle-synchronized participation make active FlexRay testing significantly more complex than CAN, Ethernet or LIN.

Passive traffic analysis and schedule reconstruction

FlexRay testing begins with passive traffic capture by connecting a FlexRay-capable tool to one or both channels of the FlexRay cluster, either via a direct tap on the bus or through a gateway that provides FlexRay monitoring access. Capturing traffic across sufficient communication cycles allows the pentester to reconstruct the static TDMA slot assignments, i.e., which node transmits in which slot, and map the payload structure of each slot. If a FIBEX or ARXML network description file is available, it can be loaded into the analysis tool to decode the signal values directly. However, if the network description file is not available, the pentester can correlate slot payloads with observable vehicle behavior using the same signal reverse engineering approach applied to CAN.

Frame injection and timing manipulation

Active FlexRay frame injection requires participation in the cluster as a synchronized node, either by integrating a FlexRay-capable hardware interface that can join the cluster's communication schedule or by compromising an existing cluster node and operating through its scheduling slot. Once synchronized, a pentester can transmit frames in static slots identified from the FIBEX or ARXML schedule, targeting safety-critical signal slots such as chassis control, steer-by-wire, or brake-by-wire commands, with attacker-controlled payloads that the receiving node will process as legitimate due to the lack of message authentication. Additionally, since FlexRay's deterministic timing is also a safety property, a pentester can assess timing robustness by transmitting frames with incorrect cycle counter values or disrupting the clock synchronization mechanism, which can cause cluster nodes to declare synchronization errors and enter a safe state, effectively resulting in a denial of service against the safety-critical functions the cluster supports.

With the common penetration testing techniques for in-vehicle networks surveyed, the following case studies illustrate how some of these techniques have been applied in real-life attacks against in-vehicle communications.

Case studies of real-life attacks on in-vehicle communications

In this section, we examine three real-world automotive attacks, analyzing how attackers combined various techniques against in-vehicle networks. The case studies illustrate how weaknesses in the in-vehicle network communications can be discovered and exploited by an attacker. While the primary focus is on the in-vehicle network aspects of each attack, relevant context is included where it is necessary to understand the full scope and impact of the attack. Together, the three case studies illustrate the breadth of the threat landscape facing in-vehicle communications, from privacy violations to vehicle theft and remote control of safety-critical functions.

Case study 1: Reconstructing vehicle routes by eavesdropping on CAN bus traffic

This case study focuses on a privacy attack that exploits the lack of confidentiality protections on the CAN bus. It demonstrates how an attacker can reconstruct the precise route driven by a vehicle by passively reading CAN bus traffic, without injecting messages, without compromising an ECU, and without triggering any observable anomaly in vehicle behavior [35].

The research demonstrates that although the CAN bus carries no GPS coordinates, it contains sufficient sensor data to reconstruct a driver's travel history with high accuracy. Using a physical vehicle model and a map-matching algorithm, the researchers achieved an average path reconstruction accuracy of 95% across four real vehicles and 41 test tracks in varied road and traffic conditions. The work highlights that in-vehicle network access represents not only a safety and security risk but also a serious and underappreciated threat to driver privacy.

Note

Source: `https://doi.org/10.1109/EUROSP63326.2025.00016 [35]`

The sequence of attack steps for this case study is provided in *Figure 6.3*.

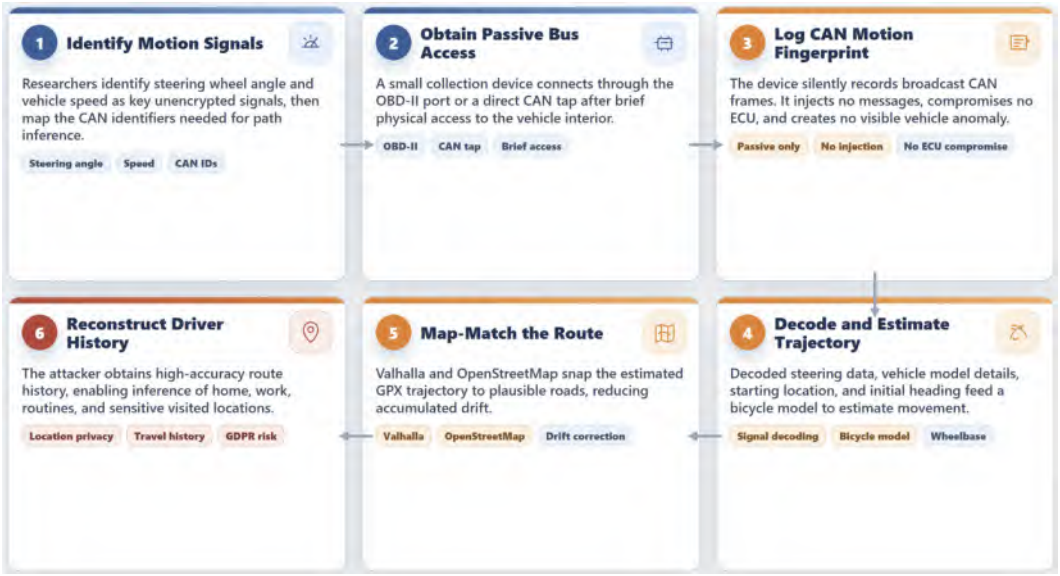


Figure 6.3: Reconstructing vehicle routes by eavesdropping on CAN bus traffic exploitation flow

Discovery and reconnaissance

The first phase involves identifying which CAN and OBD-II signals are necessary and sufficient to reconstruct a vehicle's driven path. The researchers began by surveying the signals available on the CAN bus across multiple vehicle platforms. Although the CAN bus does not transmit GPS coordinates, it carries a rich set of sensor data that can serve as a proxy for location inference. The key signals identified are the steering wheel angle and vehicle speed, both of which are broadcast periodically on the CAN bus without encryption or access control. It is important to note that these signals individually do not reveal any specific location information. However, taken together over time, they constitute a motion fingerprint of the

vehicle that can be correlated against a digital road map to reconstruct the sequence of streets and turns driven.

To enumerate the relevant CAN identifiers and signal mappings for each vehicle under test, the researchers used standard OBD-II request and response messages, which are supported across all compliant vehicles and require no special authorization to access. To identify the proprietary CAN identifier for the steering wheel angle sensor, the researchers moved the steering wheel left and right with the vehicle in a steady state while logging CAN bus packets, observing that minimal changes in wheel angle produce minimal and consistent changes in the corresponding data field. The CAN identifier for speed was obtained through standard OBD-II request and response messages, which are standardized across compliant vehicles. This allowed them to correlate raw CAN frame payloads with physical vehicle behavior and extract the signal values needed for path inference.

Initial access

The attack requires direct access to the CAN bus, which the researchers achieved by connecting a data collection device to the vehicle's OBD-II diagnostic port or directly tapping onto the CAN bus. Both access methods require brief physical access to the interior of the target vehicle, which the researchers note is realistic in several practical scenarios. For example, a mechanic or service technician performing routine work on the vehicle could attach a small collection device in a concealed location near the OBD-II port in a matter of minutes. The researchers also mention that there are commercial car theft kits, available through online marketplaces, which provide tools allowing an attacker to access a vehicle's interior quickly to attach a collection device without triggering alarms.

Once a collection device has access to the CAN bus, it operates silently in a passive, eavesdropping mode. Because the CAN bus is a broadcast medium with no node authentication, the device receives all frames transmitted by every ECU on the network without injecting any traffic and without being detectable by other ECUs.

Exploitation

The core of the attack is a deterministic path inference algorithm that translates raw CAN sensor data into a reconstructed GPS track. Once the steering wheel angle and vehicle speed are collected, the researchers apply the bicycle physical model, a well-established kinematic model used in vehicle dynamics research, to compute the vehicle's position incrementally over time. Given the vehicle's known starting location and initial heading direction, the model calculates the distance and direction traveled during each time window, producing a sequence of estimated geographic coordinates in **GPS Exchange Format (GPX)**. The external inputs required are the vehicle's starting location, its initial heading, and knowledge of the vehicle model. The vehicle model determines the correct steering wheel angle identifier, the decoding formula for the sensor, and the wheelbase value that the bicycle model uses to compute

turning radius. In practice, these are readily accessible to an attacker. The owner's home address is often inferable from public records, social media, or prior observation, and the vehicle's initial heading can be estimated from the orientation of the driveway or parking space. The vehicle model is observable directly from the exterior of the car, or retrievable from publicly available registration records.

Because kinematic models accumulate positional error over time, the researchers incorporated a map-matching step using the open-source **Valhalla** routing engine together with **OpenStreetMap** geographic data. The map-matching algorithm corrects the estimated position at each time window by snapping the inferred trajectory to the nearest plausible road on the map. This correction step significantly reduces cumulative drift and is the key to achieving the high accuracy reported by the researchers.

The technical exploitation consists of four core phases: CAN data capture using a passive collection device; signal extraction and decoding using OBD-II requests and the reverse-engineered steering wheel angle CAN identifier; trajectory estimation using the bicycle model with the vehicle wheelbase and initial bearing; and map-matched path correction using Valhalla and OpenStreetMap. No machine learning training data, no vehicle-specific road databases, and no access to the driver's smartphone or any other personal device are required. The only preconditions are the ability to collect CAN bus data by attaching a data collection device to the vehicle or through a compromised ECU, as well as knowledge of the vehicle's starting location and initial heading, both of which are readily accessible to an attacker through prior observation or publicly available information.

Persistence

The attack, called **On Path Diagnostic - Intrusion and Inference** (**OPD-II**), does not rely on persistent access in the traditional sense, as it is a data collection attack rather than an ongoing compromise. If the collection device remains physically attached to the vehicle, it will continuously log CAN bus traffic across every subsequent trip, enabling the attacker to build a detailed and growing record of the driver's movements over time. It is worth noting that a real-world attacker could extend this approach by using a cellular-enabled collection device to exfiltrate the CAN bus data remotely in near-real time.

Because the device operates silently and injects no traffic onto the CAN bus, it generates no observable anomalies in vehicle behavior and cannot be detected by other ECUs on the network. What could potentially reveal the attack is the physical presence of a small OBD-II dongle or a device concealed behind the dashboard. However, the likelihood of detection is minimal unless the driver actively inspects the vehicle.

Impact

The primary impact of the OPD-II attack is a severe violation of driver location **privacy**. By reconstructing the routes driven by a vehicle with 95% average coordinate accuracy, an attacker gains a detailed record of the driver's movements over time. This information can be used to infer the driver's home address, workplace, frequently visited locations, daily routines, and social associations, all of which represent sensitive personal data under modern privacy regulations including the **General Data Protection Regulation** (**GDPR**).

The attack is also significant from a systemic perspective. Because CAN bus and OBD-II signals are standardized across vehicle platforms, the methodology is not limited to a specific make or model. The researchers validated the attack across four different vehicles, demonstrating its broad applicability. The use of open-source tools, including Valhalla and OpenStreetMap, means that the attack requires no proprietary infrastructure and could be replicated at low cost.

From an automotive cybersecurity perspective, this case study demonstrates that the lack of confidentiality on the CAN bus is not merely a theoretical concern. It is a tangible and exploitable privacy risk affecting every vehicle that relies on unencrypted in-vehicle network communications. The driver has no indication that this data collection is occurring, and the vehicle operation is entirely unaffected, making it one of the more difficult attack categories to detect.

Case study 2: CAN injection attack to bypass the smart key system for vehicle theft

This case study describes a real-world vehicle theft technique that exploits the lack of message authentication on the CAN bus to bypass a vehicle's smart key immobilizer system. Unlike traditional key fob relay attacks that intercept and retransmit wireless key fob signals, the CAN injection attack bypasses the smart key system entirely by injecting forged CAN messages directly onto the in-vehicle network.

The attack was investigated and documented by security researchers after one of them had their Toyota RAV4 stolen [36, 37]. The research revealed that this was not an isolated incident but an attack technique that is widespread and used for organized vehicle theft targeting numerous vehicle models from multiple manufacturers. The attack itself requires no cryptographic knowledge and no interaction with the smart key. Instead, it exploits one of the most fundamental weaknesses of the CAN bus: any node connected to the bus can transmit any message, and all other nodes will accept it as legitimate. The research demonstrated that a purpose-built theft device costing approximately ten US dollars in components can unlock a vehicle and disable its immobilizer. CCTV footage of a real-world theft confirms the full process takes approximately 90 seconds.

Note

Sources:

- `https://www.youtube.com/watch?v=XZK5TvnJjXg` [36]
- `https://kentindell.github.io/2023/04/03/can-injection/` [37]

The attack flow steps are illustrated in *Figure 6.4*.

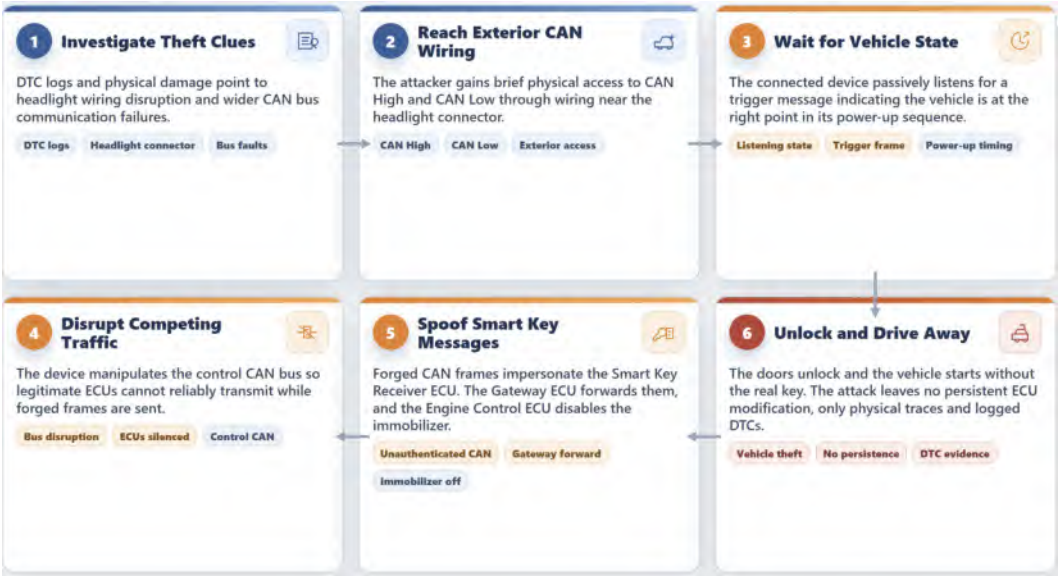


Figure 6.4: CAN injection attack to bypass the smart key system for vehicle theft exploitation flow

Discovery and reconnaissance

The investigation began after one of the security researchers' vehicles was tampered with on two separate occasions, with the front bumper pulled away and the headlight wiring connector unplugged. Initially it seemed like vandalism, but two days later the vehicle was stolen. Using the vehicle's built-in telematics system, the researchers retrieved a log of **Diagnostic Trouble Codes** (**DTCs**) recorded around the time of the theft. The DTCs indicated that communication with the lighting control ECU had been lost, which is consistent with the headlight connector being physically disconnected. However, more significantly, the logs showed that multiple other systems such as the front cameras control and hybrid engine control system had also reported communication failures at the same time. Thus, this indicated that it was not a single ECU failure but rather an issue with the CAN bus itself.

The researchers then conducted reconnaissance, searching forums, dark web marketplaces, and publicly accessible websites selling car security bypass products. This revealed a commercial ecosystem of **theft devices** targeting a wide range of vehicle models across multiple manufacturers. The theft device in this example looked like a small portable JBL Bluetooth speaker, providing plausible deniability if the thieves were stopped by law enforcement. This JBL **CAN injection theft device** was sold as an "emergency start device."

The researchers purchased the JBL CAN theft device and performed a thorough analysis. Hardware teardown of the device revealed a PIC18F microcontroller containing CAN hardware with pre-programmed firmware, a CAN transceiver to convert digital signals into the analog voltages used on the CAN bus wiring, and an additional circuit connected to the CAN transceiver that is used to drive the connected CAN bus into a recessive state, effectively preventing other ECUs from transmitting. The total component cost was estimated at approximately ten US dollars.

Initial access

This attack requires physical access to the vehicle's CAN bus wiring, but an attacker can gain access to the CAN bus without breaking into the vehicle itself. The vehicle in this case study has multiple CAN buses serving different functional domains, including a **control CAN bus** connecting ECUs for lighting, door control, and telematics, a **powertrain CAN bus** connecting ECUs for engine and hybrid motor control, and an **autonomy CAN** bus for radar and camera systems. The attackers targeted the control CAN bus, which connects the **Smart Key Receiver ECU**.

In this case, the entry point to the control CAN bus was through the front headlight connector, which the attackers reached by pulling away the front bumper. An attacker only needs a few minutes and basic tools to do this. Once the headlight connector is accessed, the CAN injection theft device is connected directly to the CAN High and CAN Low lines of the control CAN bus, drawing power from the device's internal battery. Access to the headlight connector is the only physical interaction required with the target vehicle as all subsequent attack steps occur over the CAN bus.

Exploitation

Once connected and first powered on, the CAN injection theft device enters a listening state, monitoring the bus for a specific CAN message that indicates the vehicle is in a ready state. This design prevents the device from triggering prematurely and ensures the attack sequence begins at the correct moment in the vehicle's power-up cycle.

It is worth noting why this attack is effective despite the smart key system using a cryptographic challenge-response protocol to authenticate the key fob. This cryptographic protection only covers wireless communication between the physical key fob and the Smart

Key Receiver ECU. The subsequent CAN messages sent by the Smart Key Receiver ECU to the **Engine Control ECU** to deactivate the immobilizer are not authenticated. This means that any device connected to the CAN bus can impersonate the Smart Key Receiver ECU by injecting the same messages without any cryptographic knowledge.

Once the CAN injection theft device receives the trigger message that the vehicle is in a ready state, the device executes two actions simultaneously. First, the device activates its modified CAN transceiver circuit, which actively forces the bus into a recessive state that other nodes cannot override. This prevents all other nodes on the control CAN bus from transmitting and suppresses the CAN error detection mechanism. Second, the device begins transmitting CAN frames at approximately 20 frames per second, containing spoofed messages essentially saying "Key valid, unlock immobilizer," impersonating the Smart Key Receiver ECU. In a vehicle with no message authentication on the CAN bus, these frames are indistinguishable from legitimate messages transmitted by the genuine Smart Key Receiver ECU. In addition, since the real Smart Key Receiver ECU is prevented from sending messages, the injected forged messages can be received by the **Gateway ECU** without interference.

The Gateway ECU, which connects the control CAN bus to the powertrain CAN bus, forwards the spoofed injected CAN messages to the powertrain bus, where the **Engine Control ECU** receives them and deactivates the immobilizer. The burst repetition at 20 frames per second compensates for the instability introduced by the bus manipulation to silence the other ECUs, ensuring the Gateway ECU reliably receives and forwards the forged messages.

Furthermore, to unlock the vehicle doors, the thief presses the Play button on the JBL speaker, which is connected to the PIC18F microcontroller. This changes the injected CAN message payload to instruct the door control ECU to unlock the doors. The CAN injection theft device is then disconnected, and the thief enters and starts the car without a key.

Persistence

Unlike many cyberattacks that aim to establish long-term access to a compromised system, the CAN injection attack has no persistence objective. The attack does not modify any ECU software or firmware, and no persistent changes are made to the vehicle's configuration or behavior. The sole objective is vehicle theft, which is achieved in a single operation. Once the immobilizer is deactivated and the doors are unlocked, the vehicle is driven away, and the theft device is disconnected. No further interaction with the in-vehicle network is required. The only evidence left behind is the physical disturbance to the front bumper and headlight connector, as well as the DTCs logged by the vehicle's ECUs due to the CAN bus disruption during the attack.

Impact

The immediate impact is the theft of the target vehicle without any key fob and without generating any alert visible to the owner in real time. The scale of the impact extends well beyond a single vehicle. The researchers' reconnaissance revealed a commercially organized marketplace of CAN injection theft devices targeting dozens of vehicle models across multiple manufacturers. The devices are manufactured at low cost, sold at prices ranging up to five thousand euros, and represent a structured criminal business with product catalogs and vehicle model compatibility lists. Because the underlying vulnerability is architectural, rooted in the CAN bus design principle that all nodes trust all messages, any ECU on the network that acts on unauthenticated CAN messages is potentially exploitable through the same attack technique.

From an automotive cybersecurity perspective, this case study illustrates how a protocol-level weakness, i.e., the lack of message authentication on the CAN bus, translates directly into a scalable, low-cost, real-world attack with significant financial consequences for vehicle owners and manufacturers. It also demonstrates that the physical routing of CAN bus wiring, such as through headlight connectors near the vehicle exterior, should be considered as part of the attack surface during threat analysis.

Case study 3: Remote exploitation of a passenger vehicle - from cellular network to CAN bus

This case study describes one of the most significant and widely cited automotive cybersecurity research demonstrations, in which security researchers achieved full remote access to an unaltered production vehicle over the cellular network and subsequently injected CAN messages to control safety-critical physical functions [38, 39]. The research was conducted on a 2014 Jeep Cherokee equipped with the Uconnect infotainment and telematics system and was publicly demonstrated in 2015.

Unlike the two previous case studies, which required brief physical access to the target vehicle, this attack was conducted entirely remotely, without any prior physical interaction with the vehicle. The attack chain spans multiple layers, beginning with the cellular network and progressing through the vehicle's infotainment system, an intermediate chip responsible for CAN communication, and ultimately reaching safety-critical ECUs on the CAN buses in the vehicle.

As the focus of this chapter is on in-vehicle network communications, we first provide a brief overview of how the researchers gained remote access to the vehicle, before describing the subsequent exploitation of the in-vehicle network.

The researchers identified that the Uconnect system exposed a Wi-Fi access point with a predictable password derivable from the timestamp of Wi-Fi hotspot activation. Connecting to this Wi-Fi network and scanning for open ports, they identified an unauthenticated **Desktop Bus** (**D-Bus**) service running on the head unit, which they exploited to gain code execution on the Uconnect system running the QNX operating system.

Note

Terms:

D-Bus (Desktop Bus) is an inter-process communication mechanism commonly used in Linux and QNX systems to allow processes to communicate with each other and expose callable services.

To go beyond Wi-Fi range, the researchers used a femtocell, a small cellular base station, to connect their equipment to the same cellular network IP range used by vulnerable vehicles. This allowed them to scan the cellular operator's IP address range, identify vehicles with open ports, and reach the unauthenticated D-Bus service directly over the cellular network. Once a target was identified, the same exploit was delivered remotely, granting code execution on the head unit without any interaction with the vehicle or its driver. From this point of access, the focus of this case study begins. The next steps involve gaining access to the in-vehicle CAN buses and injecting messages to affect physical vehicle functions.

Note

Sources:

- `https://www.youtube.com/watch?v=OobLb1McxnI` [38]
- `https://ioactive.com/wp-content/uploads/pdfs/IOActive_Remote_Car_Hacking.pdf` [39]

The sequence of attack steps is shown in *Figure 6.5*.

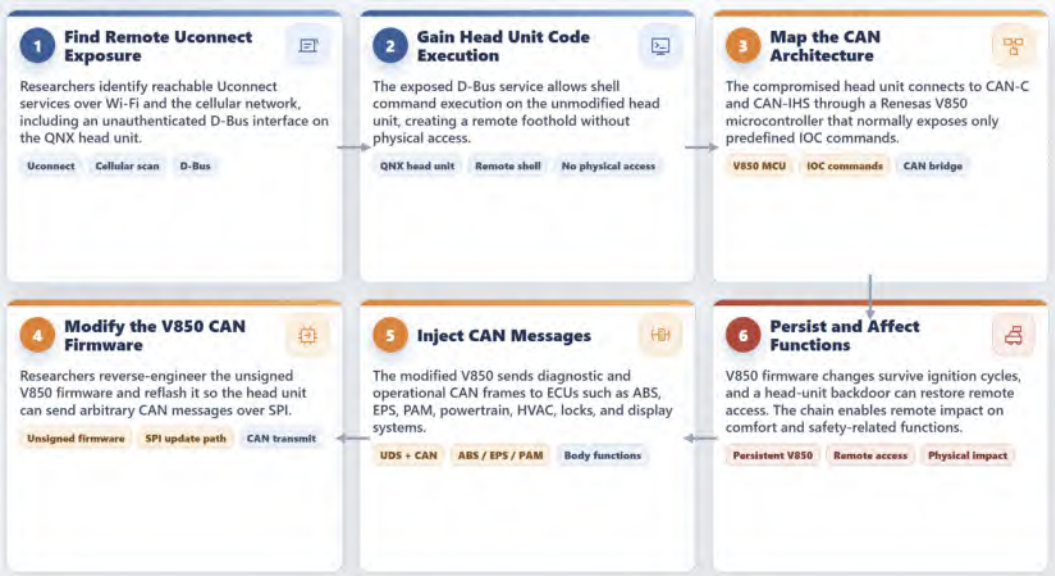


Figure 6.5: Remote exploitation of a passenger vehicle - From cellular network to CAN bus exploitation flow

Discovery and reconnaissance

With code execution established on the Uconnect head unit, the researchers began exploring the vehicle's internal architecture. The 2014 Jeep Cherokee uses two CAN buses: the **CAN-C bus**, which connects safety-critical ECUs including the **Anti-lock Brake System** (**ABS**), **Electric Power Steering** (**EPS**), powertrain control, transmission control, and the **Park Assist Module** (**PAM**), and the **CAN-IHS bus**, which connects comfort and body ECUs including **Heating, Ventilation, and Air Conditioning** (**HVAC**), door modules, and heated seats. A key finding regarding the architecture was that the Uconnect head unit, running QNX on a Texas Instruments OMAP-DM3730 processor, which the researchers had already compromised, is connected to both CAN buses simultaneously, making it a potential pivot point for reaching all ECUs in the vehicle.

However, the head unit running QNX did not have the direct ability to send arbitrary CAN messages. The researchers discovered that CAN communication from the head unit was handled by a separate chip, the Renesas V850ES/FJ3 microcontroller, which sits between the head unit and the CAN buses and is responsible for executing specific CAN-related functions. This communication with the V850 was based on predefined **Input Output Controller** (**IOC**) commands, where each command triggered a specific hardcoded CAN action in the V850 firmware. The researchers discovered that to send arbitrary CAN messages, they needed to understand and modify the V850 firmware itself.

Initial access

As described in the overview, remote code execution on the Uconnect head unit was achieved by exploiting the unauthenticated D-Bus service exposed on port 6667 of the QNX operating system. The **NavTrailService** D-Bus service, which was designed for inter-process communication, exposed an **execute** method that allowed arbitrary shell commands to be run directly. Using four lines of **Python** via the D-Bus library, the researchers opened a remote root shell on the unmodified head unit without requiring any jailbreak or prior system modification.

As mentioned, the head unit could not send arbitrary CAN messages due to the predefined IOC command structure of the V850 microcontroller. To be able to send arbitrary CAN messages, the researchers needed to modify and reflash the V850 firmware.

To achieve this, the researchers reverse engineered the V850 firmware using **IDA Pro**, which already included a processor module for the V850 architecture. **Python** scripts were used within IDA Pro to normalize the disassembly, map cross references, and identify the functions responsible for CAN message transmission. **IAR Workbench**, a development environment commonly used in automotive embedded development, was also consulted as it included example CAN code for the same processor family, which helped the researchers identify and confirm the relevant CAN transmission functions in the firmware. Through this analysis, the researchers identified a function, which accepted a CAN ID, data length, and message payload, and mapped the **Serial Peripheral Interface** (**SPI**) communication channel used by the OMAP chip to send commands to the V850. This modified V850 firmware was then ready to be flashed.

Moreover, the firmware image was not cryptographically signed, meaning no digital signature verification was performed during the update process. The **iocupdate** binary, the tool used to flash new firmware onto the V850 over SPI, was already present on the head unit filesystem, which meant that the researchers did not need to load any additional tools onto the target system.

However, the standard firmware update process required the V850 to be in bootloader mode, which could only be triggered by a full system reset, temporarily taking the OMAP processor offline and causing the researchers to lose remote access. To work around this, they made the **MultiMediaCard** (**MMC**) partition writable using standard mount commands and replaced the **hd** hex dump utility, an executable called early in the update boot sequence, on the MMC filesystem with a custom shell script that called **iocupdate** with the modified firmware image and then restarted the system back into application mode. They then triggered the system to reboot into bootloader mode using **Lua** scripts that invoked the **onoff** system module. The entire reflashing process, including the reboot cycle, took approximately 25 seconds, after which the system returned to application mode with the modified V850 firmware running.

Note

Terms:

Lua is a lightweight scripting language commonly embedded in software systems for configuration and automation purposes.

Exploitation

With a modified V850 firmware in place, the researchers had the ability to send arbitrary CAN messages onto both the CAN-C and CAN-IHS buses from a remote position over the cellular network. The exploitation phase then consisted of identifying and crafting the specific CAN messages needed to trigger physical actions on the vehicle.

The researchers used standard UDS diagnostic CAN messages as well as normal operational CAN messages to interact with target ECUs. For each target function, the relevant CAN IDs and message payloads were identified through a combination of reverse engineering the V850 firmware and monitoring live CAN traffic.

The safety-critical actions achieved by injecting CAN messages from the head unit via the modified V850 firmware included disabling the engine while the vehicle was in motion, disabling the brakes at low speed by sending diagnostic messages to the ABS module, and controlling the electric power steering by first placing the PAM into a diagnostic session to silence its operational messages and then sending commands to the EPS ECU.

Non-safety functions including the HVAC system, radio volume, display output, and door locks were controlled through Lua scripts that invoked D-Bus services on the head unit directly, without requiring CAN message injection.

Together, these techniques demonstrated that a remote attacker with no prior physical access to the vehicle could affect both comfort functions and safety-critical vehicle behavior.

Persistence

Persistence in this attack can be considered at two levels. At the in-vehicle network level, flashing the modified V850 firmware onto the microcontroller ensured that the capability to send arbitrary CAN messages persisted across power cycles, as the V850 firmware is retained after the vehicle is switched off and restarted. This meant that the attacker's capability to send arbitrary CAN messages was retained across ignition cycles independently of the head unit's software state.

At the head unit level, the remote root shell obtained via the D-Bus service did not persist across reboots, meaning that after a power cycle the attacker would need to re-exploit the unauthenticated D-Bus service to regain remote shell access. However, the researchers

demonstrated that a more persistent backdoor could be established by uploading an SSH public key and modifying the boot.sh script on the writable MMC partition to start the SSH daemon on every boot. Since changes to the MMC partition persist across reboots, this would give the attacker a permanent SSH backdoor that survives power cycles without requiring repeated D-Bus exploitation.

The combination of a persistent SSH backdoor on the head unit and a persistent V850 firmware modification would therefore give an attacker reliable and repeatable remote access to the head unit and also the capability to send arbitrary CAN messages on the in-vehicle CAN buses across ignition cycles, for as long as the connected vehicle remained reachable through the cellular network.

Impact

The impact of this attack was significant both technically and from an industry perspective. At the individual vehicle level, the researchers demonstrated that a remote attacker with no previous physical access to the vehicle could disable safety-critical functions including the engine and brakes, cause the steering wheel to jerk, and monitor or manipulate the driver's environment, all while the vehicle was in motion and the driver had no indication that the vehicle had been compromised.

The scale of the potential impact extended well beyond a single vehicle. The researchers demonstrated that the cellular network attack could be automated to scan for and exploit all vulnerable vehicles connected to the cellular operator's network. Because the Uconnect vulnerability was present across an entire product generation, this meant that a large number of vehicles on the road at the time were potentially reachable from anywhere with a cellular connection.

From an in-vehicle network perspective, this case study illustrates several critical weaknesses. The head unit's direct connection to both CAN buses without adequate isolation meant that a single point of compromise in the infotainment system provided access to safety-critical ECUs. The V850 firmware update mechanism, accessible from the head unit over SPI, lacked sufficient authentication controls and digital signature verification of the V850 firmware itself, allowing the researchers to replace it with a modified version capable of sending arbitrary CAN messages.

Moreover, the lack of message authentication on the CAN bus and adequate network segmentation meant that once the V850 was modified, there was no mechanism to prevent or detect the injected messages on the CAN bus. Together, these weaknesses formed an unbroken attack chain from a remote cellular connection to direct control of safety-critical vehicle functions, demonstrating the severe consequences of insufficient isolation and inadequate security controls between the telematics system and the in-vehicle network.

The case studies presented in this chapter demonstrate how weaknesses in in-vehicle communication can be identified and exploited under real-world conditions. The following section outlines best practices that automotive organizations can adopt to address these weaknesses and reduce the overall attack surface of in-vehicle networks.

Best practices for securing in-vehicle communications

Securing in-vehicle network communication requires a multi-layered approach that addresses weaknesses at the protocol, architecture, and software levels simultaneously. The weaknesses described in this chapter highlight a set of underlying security gaps, such as lack of message authentication, unencrypted communication, insufficient network segmentation, and unprotected diagnostic interfaces.

Addressing these gaps starts with adopting a zero-trust mindset, i.e., assuming that any ECU on the network can be compromised and that attackers may already have a foothold within the vehicle. In practice, this translates to enforcing mutual authentication between communicating ECUs, applying message authentication and encryption where applicable, and segmenting the network to contain the impact of a compromised node. The following sections cover the key best practices for addressing these gaps.

Implementing message authentication

Legacy in-vehicle network protocols such as CAN, LIN, and FlexRay were designed without message authentication, meaning any node on the bus can transmit any message without any verification. Addressing this security concern requires authentication to be added above the inherent protocol layer, and considering bandwidth and performance requirements, message authentication should be applied only to the messages where the security risk is highest.

For CAN and LIN, **SecOC** provides the standard mechanism for appending a MAC to secured PDUs. A correctly deployed SecOC implementation must use a sufficiently long MAC to resist guessing attacks, implement freshness values to prevent replay, and discard any PDU that fails MAC verification, i.e., failed PDUs must never be passed to the application. Moreover, symmetric keys used for MAC calculations must be provisioned and stored securely, ideally in a **hardware security module**, and must not be shared across vehicles or ECU variants in a way that makes a single key compromise a fleet-wide risk.

For FlexRay, the same authentication principles apply. Protection must be implemented above the protocol layer since FlexRay provides no native mechanism for authentication. Priority should be given to slots carrying safety-critical signals such as chassis control commands and steer-by-wire signals, where message forgery could lead to direct safety implications.

For automotive Ethernet, transport-layer authentication is addressed through TLS, preferably with mutual authentication enforced between communicating ECUs, ensuring that both

parties verify each other's identity before exchanging data. Further details on TLS and confidentiality controls are discussed in the following subsection.

Applying encryption and transport layer security

All in-vehicle network protocols transmit data in plaintext by default. In contrast to CAN, LIN, and FlexRay, which have limited support for encryption, on Automotive Ethernet, this is a configuration mistake rather than a protocol limitation. That is, the bandwidth to support **TLS** is available, and there is no justification for running diagnostic, OTA update, or safety-relevant services over unencrypted connections.

Thus, TLS should be applied to Ethernet services handling sensitive data, with **mTLS** enforced for ECU-to-ECU communication to ensure both parties are mutually authenticated. ECU certificates should be issued by a trusted Certificate Authority, revocation must be enforced, and certificate pinning should be considered for communication relationships that are defined at design time. **MACsec** can complement TLS by securing individual Ethernet links at the data link layer, though it should be understood as a hop-by-hop control rather than an end-to-end solution. It is important to know that an intermediate node such as a gateway that terminates a MACsec session has full plaintext visibility of the traffic passing through it.

On classical CAN, encryption is constrained by the 8-byte payload limit and real-time requirements. Therefore, the recommended approach is to use SecOC for authentication on CAN, and use Ethernet where confidentiality is required.

Enforcing network segmentation and access control

A flat, poorly segmented vehicle network amplifies protocol-level weaknesses. For example, a compromised infotainment system in unsegmented architecture could allow a remote attacker to send messages directly to a safety-critical ECU. Therefore, **network segmentation** is imperative to limit the impact of a compromised node by enforcing boundaries between domains of different trust levels and functional criticality [40].

Vehicle architectures should organize ECUs into security zones or functional domains, with communication between zones routed exclusively through a central gateway or zone controller that enforces explicit **access control policies**. On CAN, this means gateways must maintain a **whitelist** of permitted message IDs per domain boundary and drop messages that are not on the list. On Ethernet, inter-VLAN routing policies must restrict traffic to specific services and ports required for legitimate vehicle operation. For example, **access to switch and gateway management interfaces** should be restricted to be reachable only from a dedicated management VLAN to prevent a compromised node on a lower-trust VLAN from directly interacting with the management interface.

Moreover, connected trunk ports must be configured with the same native VLAN to **prevent VLAN hopping**, and **port-based access control** (IEEE 802.1X) should be enforced to prevent

unauthorized devices from gaining network access by physically connecting to an exposed Ethernet port. Moreover, port security should be configured on Ethernet switch ports to restrict the number of MAC addresses permitted per port, preventing CAM table overflow attacks.

Securing diagnostic interfaces

Diagnostic interfaces such as UDS over CAN, DoIP over Ethernet, and XCP over both CAN and Ethernet provide access to ECU memory, configuration, and control functions. If not properly secured, these interfaces represent one of the highest-risk attack surfaces in the vehicle network.

Several security controls should be applied to address these risks. **UDS Security Access** must be enforced before granting access to privileged sessions, and the seed-key algorithm must be sufficiently complex to resist brute-force attacks. As such, the use of standardized cryptographic algorithms with appropriate key lengths is recommended. Moreover, **DoIP routing activation authentication** must be implemented to prevent unauthorized hosts that can reach the gateway from establishing diagnostic sessions targeting ECUs behind the gateway. In line with the network segmentation principle described in the previous subsection, **access to diagnostic interfaces** must be restricted by network segment, with DoIP accessible only from a dedicated diagnostics VLAN and not reachable from other domains such as the infotainment domain.

Finally, XCP must be disabled in all production builds. Leaving XCP active in its default unauthenticated state provides any network-connected node with direct, unrestricted memory read and write access to the ECU. This security risk is further amplified by XCPoE's network reachability on Automotive Ethernet. Thus, the transition from development to production build must include an explicit verification step confirming XCP is properly disabled.

Establishing secure development and configuration practices

Several of the weaknesses identified in this chapter arise not from protocol design limitations but from configuration and development mistakes. For example, security features may be disabled during development and not re-enabled for production builds, or there may be implementation errors in protocol software components. It is imperative that security is built into the development process following **security-by-design** principles rather than applied as an afterthought or considered during a post-development review. ISO/SAE 21434:2021 [41] provides the framework for establishing such a cybersecurity engineering process across the vehicle development lifecycle, from concept phase through decommissioning.

Production builds must be explicitly verified against a **security configuration checklist** confirming that XCP is disabled, SecOC is enabled on all required message IDs, Security Access is enforced on privileged UDS sessions, DoIP routing activation authentication is configured,

and all development-only interfaces are disabled. Beyond configuration, all protocol software implementations must apply strict **input validation** on received payloads, including length field validation, type checking, and boundary checks, since implementation errors that can lead to buffer overflows in protocol stacks are a common source of exploitable vulnerabilities. Moreover, **security testing** of in-vehicle network communication, including security functional testing, fuzz testing, and penetration testing, should be integrated into the development lifecycle rather than applied only at the end so that findings can be addressed as early as possible.

Enabling continuous monitoring and anomaly detection

Preventive security controls reduce the attack surface but cannot provide complete protection. An attacker who has gained access to the in-vehicle network through a compromised ECU or an unknown vulnerability may operate undetected without monitoring in place. Therefore, an in-vehicle **Intrusion Detection System** (**IDS**) can detect attack activity and anomalies over CAN and Ethernet by monitoring for unexpected message IDs, transmission frequency anomalies, malformed protocol frames, unauthorized diagnostic session attempts, and SOME/IP service discovery from unexpected sources.

Security-relevant events such as SecOC verification failures, UDS Security Access failures, DoIP routing activation rejections, and IDS alerts should be collected, centralized, and forwarded to a backend **Vehicle Security Operations Center** (**VSOC**) where such events can be correlated across the fleet. It is important to note that **continuous monitoring** should be seen as a complementary measure that detects what preventive controls miss, and not as a substitute for applying appropriate security controls for authentication, encryption, and network segmentation as described in the previous subsections.

Summary

In-vehicle network protocols form the communication backbone of modern vehicles, enabling real-time coordination between ECUs across safety-critical and non-safety-critical domains. As this chapter has shown, many of these protocols were designed for reliability and determinism rather than security. The weaknesses that result, such as lack of authentication, plaintext communication, and insufficient access control, represent real attack vectors that attackers can and have exploited in the past.

This chapter examined the security weaknesses present in in-vehicle network protocols and mapped them to concrete penetration testing techniques that security engineers can apply in a test lab or vehicle environment. To better understand how these techniques can be applied in practice, the chapter also reviewed several real-world case studies in which in-vehicle network communications were compromised. These case studies demonstrate how protocol

weaknesses and implementation mistakes can be exploited to gain unauthorized access and interfere with vehicle functionality.

It is worth noting that many of the weaknesses discussed in this chapter are not inherent protocol limitations but rather deployment and configuration mistakes that security engineers and architects have the means to address. The best practices presented provide a principle-based framework for securing in-vehicle network communications against attacks described in this chapter.

With in-vehicle network communications security covered, the next chapter goes deeper into in-vehicle systems.

References

[1] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "*Experimental Security Analysis of a Modern Automobile*," in Proc. IEEE Symposium on Security and Privacy (S&P), Oakland, CA, USA, May 2010, pp. 447-462. https://doi.org/10.1109/SP.2010.34

[2] ISO, "*ISO 11898-1:2024 Road Vehicles -- Controller Area Network (CAN) -- Part 1: Data Link Layer and Physical Coding Sublayer*," International Organization for Standardization, Geneva, Switzerland, 2024. [Online]. Available: https://www.iso.org/standard/86384.html

[3] AUTOSAR, "*Specification of Secure Onboard Communication (SecOC)*," AUTOSAR Classic Platform, Document ID 654, Release 4.4.0, 2019. [Online]. Available: https://www.autosar.org/fileadmin/standards/R18-10_R4.4.0_R1.5.0/CP/AUTOSAR_SWS_SecureOnboardCommunication.pdf

[4] LIN Consortium, "*LIN Specification Package, Revision 2.2A*," Dec. 2010. [Online]. Available: https://www.lin-cia.org/fileadmin/microsites/lin-cia.org/resources/documents/LIN_2.2A.pdf

[5] FlexRay Consortium, "*FlexRay Communications System -- Protocol Specification, Version 3.0.1*," Oct. 2010. [Online]. Available: https://docslib.org/doc/6271729/flexray-communications-system-protocol-specification-v3-0-1

[6] IEEE, "*IEEE Std 802.3-2022: IEEE Standard for Ethernet*," IEEE, New York, USA, 2022. [Online]. Available: https://standards.ieee.org/ieee/802.3/10422/

[7] IEEE, "*IEEE Std 802.1AE-2018: IEEE Standard for Local and Metropolitan Area Networks -- Media Access Control (MAC) Security*," IEEE, New York, USA, 2018. [Online]. Available: https://standards.ieee.org/standard/802_1AE-2018.html

[8] ISO, "*ISO 15765-2:2016 Road Vehicles -- Diagnostic Communication over Controller Area Network (DoCAN) -- Part 2: Transport Protocol and Network Layer Services*," International Organization for Standardization, Geneva, Switzerland, 2016.

[9] SAE International, "*SAE J1939-21: Data Link Layer*," SAE International, Warrendale, PA, USA, 2006 (reaffirmed 2019).

[10] ISO, "*ISO 13400-2:2022 Road Vehicles -- Diagnostic Communication over Internet Protocol (DoIP) -- Part 2: Transport Protocol and Network Layer Services*," International Organization for Standardization, Geneva, Switzerland, 2022.

[11] ISO, "*ISO 14229-1:2020 Road Vehicles -- Unified Diagnostic Services (UDS) -- Part 1: Application Layer*," International Organization for Standardization, Geneva, Switzerland, 2020.

[12] ASAM e.V., "*ASAM MCD-1 XCP: Universal Measurement and Calibration Protocol (XCP), Part 1: Overview*," ASAM, Gaimersheim, Germany, 2022. [Online]. Available: `https://www.asam.net/standards/detail/mcd-1-xcp/`

[13] M. Bozdal, M. Samie, S. Aslam, and I. Jennions, "*Evaluation of CAN Bus Security Challenges*," Sensors, vol. 20, no. 8, p. 2364, 2020. `https://doi.org/10.3390/s20082364`

[14] E. Aliwa, O. Rana, C. Perera, and P. Burnap, "*Cyberattacks and Countermeasures for In-Vehicle Networks*," ACM Computing Surveys, vol. 54, no. 1, pp. 1-40, 2021. `https://doi.org/10.1145/3431233`

[15] K. Iehira, H. Inoue, and K. Ishida, "*Spoofing Attack Using Bus-Off Attacks Against a Specific ECU of the CAN Bus*," in Proc. IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, NV, USA, Jan. 2018. `https://doi.org/10.1109/CCNC.2018.8319180`

[16] H. Lee, H. Lee, S. Jun, and H. K. Kim, "*Expanding the Attack Scenarios of SAE J1939: A Comprehensive Analysis of Established and Novel Vulnerabilities in Transport Protocol*," ArXiv, preprint arXiv:2406.00810, Jun. 2024. `https://doi.org/10.48550/arXiv.2406.00810`

[17] M. De Vincenzi, G. Bartoli, I. Matteucci, and C. Bernardeschi, "*A Systematic Review on Security Attacks and Countermeasures in Automotive Ethernet*," ACM Computing Surveys, vol. 55, no. 14s, pp. 1-39, Jul. 2023. `https://doi.org/10.1145/3588027`

[18] python-can Developers, "*python-can: CAN Bus Interface for Python*," [Online]. Available: `https://python-can.readthedocs.io`

[19] cantools Developers, "*cantools -- CAN Bus Database Library for Python*," [Online]. Available: `https://github.com/cantools/cantools`

[20] T. Ely, "*SavvyCAN -- CAN Bus Traffic Analyzer*," [Online]. Available: https://www.savvycan.com/

[21] linux-can Project, "*can-utils: Linux CAN Utilities (candump, cansend, cangen)*," [Online]. Available: https://github.com/linux-can/can-utils

[22] CaringCaribou Contributors, "*caringcaribou -- A Friendly Car Security Exploration Tool*," [Online]. Available: https://github.com/CaringCaribou/caringcaribou

[23] BUSMASTER Contributors, "*BUSMASTER -- Open Source CAN Bus Tool*," [Online]. Available: https://github.com/rbei-etas/busmaster

[24] Scapy Development Team, "*Scapy -- Packet Manipulation Library*," [Online]. Available: https://scapy.net/

[25] Kayak Contributors, "*Kayak -- CAN Bus Analysis Tool*," [Online]. Available: https://dschanoeh.github.io/Kayak/

[26] comma.ai, "*opendbc -- Open Source CAN DBC Database for Production Vehicles*," [Online]. Available: https://github.com/commaai/opendbc

[27] H. Kim, Y. Jeong, W. Choi, D. H. Lee, and H. J. Jo, "*Efficient ECU Analysis Technology Through Structure-Aware CAN Fuzzing*," IEEE Access, vol. 10, pp. 23259-23271, 2022. https://doi.org/10.1109/ACCESS.2022.3151358

[28] D. K. Oka, "*Building Secure Cars: Assuring the Automotive Software Development Lifecycle*," Wiley, 2021. ISBN: 978-1-119-71074-5. DOI: 10.1002/9781119710783.

[29] Wireshark Foundation, "*Wireshark -- Network Protocol Analyzer*," [Online]. Available: https://www.wireshark.org/.

[30] Nmap.org, "*Nmap Network Scanner*," [Online]. Available: https://nmap.org/

[31] tcpdump/libpcap Developers, "*tcpdump -- Packet Capture and Analysis Tool*," [Online]. Available: https://www.tcpdump.org/.

[32] sslyze, "*Fast and powerful SSL/TLS scanning library*," [Online]. Available: https://github.com/nabla-c0d3/sslyze

[33] mitmproxy contributors, "*mitmproxy -- Interactive HTTPS Proxy*," [Online]. Available: https://mitmproxy.org/

[34] Ettercap Project, "*Ettercap -- A Comprehensive Suite for Man-in-the-Middle Attacks*," [Online]. Available: https://www.ettercap-project.org/

[35] T. Bianchi, A. Brighente, M. Conti, and A. Valori, "*Your Car Tells Me Where You Drove: A Novel Path Inference Attack via CAN Bus and OBD-II Data*," in Proc. IEEE 10th European

Symposium on Security and Privacy (EuroS&P), 2025, pp. 113-132. https://doi.org/10.1109/EUROSP63326.2025.00016

[36] I. Tabor and K. Tindell, "*CAN injection: Keyless car theft*," presented at DEF CON 31, Las Vegas, NV, USA, Aug. 2023. [Online]. Available: https://www.youtube.com/watch?v=XZK5TvnJjXg

[37] K. Tindell, "*CAN Injection: Keyless Car Theft*," Apr. 3, 2023. [Online]. Available: https://kentindell.github.io/2023/04/03/can-injection/

[38] C. Miller and C. Valasek, "*Remote exploitation of an unaltered passenger vehicle*," presented at DEF CON 23, Las Vegas, NV, USA, Aug. 2015. [Online]. Available: https://www.youtube.com/watch?v=OobLb1McxnI

[39] C. Valasek and C. Miller, "*Remote exploitation of an unaltered passenger vehicle*," IOActive Technical White Paper, Aug. 2015. [Online]. Available: https://ioactive.com/wp-content/uploads/pdfs/IOActive_Remote_Car_Hacking.pdf

[40] National Highway Traffic Safety Administration, "*Cybersecurity Best Practices for the Safety of Modern Vehicles*," Updated 2022. [Online]. Available: https://www.nhtsa.gov/sites/nhtsa.gov/files/2022-09/cybersecurity-best-practices-safety-modern-vehicles-2022-tag.pdf

[41] ISO/SAE 21434:2021, "*Road Vehicles -- Cybersecurity Engineering*," International Organization for Standardization, Geneva, Switzerland, 2021.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

Part 3

Busting the Myth of the Deeper Architecture Layers Security

This part addresses a persistent assumption in the industry that the deeper layers of vehicle architecture are inherently secure by virtue of their physical inaccessibility. You will discover that microcontroller-based ECUs, high-performance computing platforms, vehicle sensors, and AI/ML systems all carry significant attack surfaces that a skilled adversary can exploit. The part progresses from hardware-level attacks such as fault injection and debug interface exploitation through sophisticated software exploits targeting hypervisors and machine learning inference pipelines. The final chapter then pivots to a forward-looking defensive strategy, outlining how organizations can apply a Cybersecurity Management System (CSMS) and secure-by-design principles within an AI-enabled secure engineering lifecycle to achieve lasting resilience across the full vehicle lifecycle.

This part has the following chapters:

- *Chapter 7, Attacking MCU-Based Embedded ECUs*
- *Chapter 8, Attacking the High-Performance ECUs*
- *Chapter 9, Attacking Vehicle Sensors and AI/ML*
- *Chapter 10, A Path Forward*

7

Attacking MCU-Based Embedded ECUs

Even with the shift toward centralized computing systems, **electronic control units (ECUs)** built on **microcontrollers (MCUs)** continue to be workhorses for automotive sensing and actuation. Unlike the high-performance system-on-chips (SoCs) found in vehicle computers and advanced driver-assistance systems, traditional MCU-based ECUs leverage constrained resources to perform deterministic real-time control. These MCUs typically integrate a CPU core (or a few cores) with limited on-chip memory and peripherals, optimized to interface directly with a mixture of vehicle sensors and actuators. They run lightweight **real-time operating systems** (**RTOS**) or even bare metal, in contrast to the rich operating systems on automotive SoCs. You will find microcontroller-driven ECUs throughout a modern vehicle, where they manage specific functions such as engine control, anti-lock braking, airbag deployment, power steering, and body control. Many of these functions are safety-critical, so the control units must react to events within milliseconds while operating under strict memory, computing, reliability, and safety constraints.

Historically, automotive MCUs were not designed with hostile adversaries in mind. As vehicles became connected and attackers turned their attention to internal vehicle networks, even these constrained ECUs became the target of attacks. Depending on the vehicle's E/E architecture, the embedded ECUs may be directly accessible through a network or physical interface or may lie at the deeper layers of the vehicle's architecture. In either case, an attacker who is aiming to influence the vehicle's behavior is interested in eventually compromising an embedded ECU either directly or indirectly.

Moreover, studying MCU-based ECU architecture is not only useful for penetrating deeply embedded ECUs, but also important when studying HPC security (as we will see in the next chapter). It is quite common for such systems to integrate an embedded MCU, for example to

handle real-time safety tasks such as an embedded safety domain, or as an SoC companion that mainly acts as a watchdog and a fail-over for the entire system. These MCUs will run the same software stack as a standalone embedded ECU, making it possible to reuse the same attack strategies against the HPC. In addition to the obvious vehicle control effects, attacking such ECUs can result in recovering security assets that are useful in more advanced exploits, such as recovering long-term cryptographic keys for in-vehicle networks or credentials used to connect to the vehicle charging infrastructure.

In this chapter, we will dive into the offensive security domain of MCU-based ECUs. We begin with an overview of their hardware architecture and common software stacks. We then examine their common weaknesses both in hardware and software. Next, we outline methodologies for building attack chains against these constrained devices, from initial recon and firmware extraction up to exploitation and persistence. Several case studies are then shared to illustrate real-world compromises of MCU-based ECUs and their impact. Finally, we discuss security best practices and defenses to thwart the most common attacks while emphasizing the need for a layered approach to security. By the end of this chapter, you will gain insights into the techniques by which MCU-based ECUs are compromised and how to make those attacks less effective.

In the rest of this chapter, we will cover the following main topics:

- Overview of MCU-based ECUs
- Understanding the hardware and software anatomy
- Common hardware and software weaknesses in MCU-based ECUs
- Penetration testing techniques and attack-chain construction
- Case studies of MCU-based ECU compromise
- Security best practices for resilient embedded ECUs

Overview of MCU-based ECUs

Before we can dive into the attack techniques against MCU-based ECUs, we must familiarize ourselves with the architecture of those systems and the unique aspects that both make them a target and contribute to their susceptibility to attacks.

Modern automotive MCUs are predominantly 32-bit microcontrollers engineered for real-time, safety-critical execution. Examples of MCU architectures include Infineon TriCore (found in the AURIX family), and Renesas RH850 which support a mixture of proprietary CPU cores and ones that are commonly available like the ARM Cortex-R family [4] and RISC-V. These devices frequently adopt Harvard architectures with tightly coupled memories to eliminate unpredictable access latencies and ensure consistent execution timing.

Safety-critical MCUs incorporate safety mechanisms such as lockstep execution, watchdog timers, error-correcting code (ECC) protection for memory, and built-in self-test routines. The objective of these designs is to ensure that faults such as random hardware failures or transient bit flips are detected promptly and that the system can transition into a defined safe state. Memory protection hardware is also commonly integrated to isolate software components of differing safety criticality. Typically, a distributed Memory Protection Unit (MPU) provides multiple protection contexts that restrict which software components may access specific memory regions. This mechanism helps prevent errant code from overwriting safety-critical data or altering control logic.

It's common to find MCUs with multiple cores such as the AURIX TC3xx device which integrates six CPU cores operating at up to 300 MHz, delivering more than 4000 DMIPS of throughput for real-time workloads [1]. This level of performance is needed for the tightly scoped control functions these ECUs are designed to perform, while ensuring that hard timing deadlines for ignition events, braking loops, or torque control are consistently met.

Figure 7.1 shows a typical MCU-based ECU with the various sensors, actuators, and network interfaces it supports.

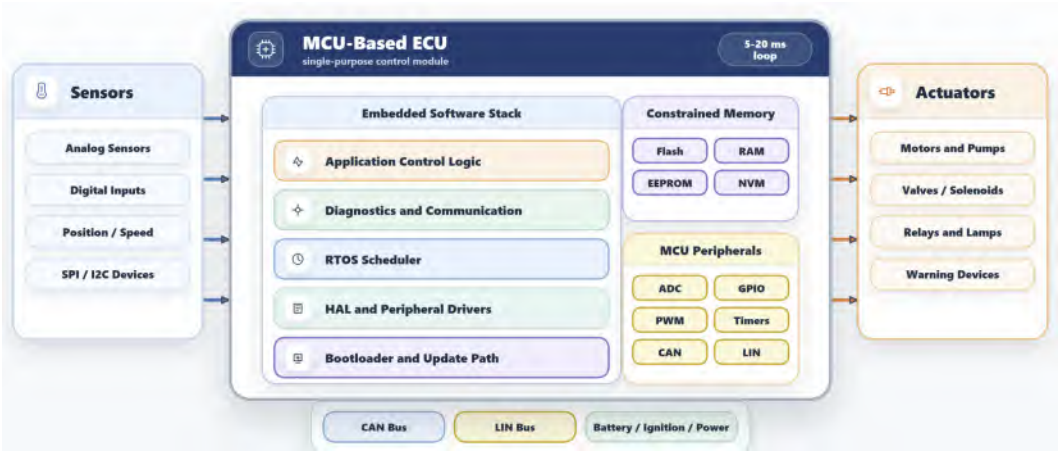


Figure 7.1: Simplified view of an MCU-based ECU hardware and software components

Traditional MCU-based ECUs, as shown in the figure, share several defining characteristics. They typically operate with constrained memory resources, often limited to a few megabytes of flash and RAM. They are designed for deterministic execution, with control-loop timing on the order of 5 to 20 milliseconds. They also provide direct electrical interfaces to sensors and actuators, including analog and digital inputs, Pulse Width Modulation (PWM) outputs, and native support for in-vehicle networks such as CAN and LIN. These interfaces enable the ECU to function as a standalone embedded system responsible for a narrowly defined task such as

engine spark timing, air-fuel mixture control, braking pressure modulation, battery charge management, steering assistance, airbag deployment logic, and climate regulation. Because these systems directly influence physical vehicle behavior, a successful compromise can have immediate and tangible operational and safety consequences.

Given their focused role, they typically do not require multi-gigahertz clock speeds or gigabytes of memory. Instead, they prioritize timing consistency, fault tolerance, and functional safety. This design philosophy stands in contrast to high-performance automotive SoCs, such as those used for infotainment or centralized vehicle computers, as we will see in the next chapter. While those platforms offer significantly higher aggregate compute capacity, they generally do not provide the same millisecond-level response determinism that an MCU designed for real-time control can provide. Modern automotive MCUs frequently include dedicated coprocessors, such as hardware security modules, to handle cryptographic operations. This reflects the growing recognition that cybersecurity requirements must be addressed alongside traditional functional safety concerns.

From the software side, MCUs typically execute specialized real-time operating systems. A long-standing industry standard is OSEK/VDX and its successor, AUTOSAR OS [2], which provides a statically configured RTOS with fixed-priority scheduling and interrupt handling designed for predictability. AUTOSAR OS is deployed as part of the AUTOSAR Classic Platform, which standardizes the software architecture for many production ECUs. This standardization benefits engineers by improving interoperability and reuse, but as we will see later in the chapter, it also provides attackers with publicly available specifications that document supported services, interfaces, and diagnostic behaviors.

Alternatively, ECU suppliers may choose a bare-metal solution that relies on a lightweight RTOS such as FreeRTOS or its safe sibling called SafeRTOS. Regardless of implementation, these operating systems provide only essential services such as task scheduling, timing counters, and memory protection. They intentionally avoid the complexity and feature set of general-purpose operating systems like Linux. As a result, application software is commonly written in C and constrained by MISRA C guidelines [3]. The resulting software stack is narrow, purpose-built, and fundamentally different from the multi-application environments found on automotive SoCs.

Historically, the lack of direct user interfaces and external connectivity led many engineers to assume that such ECUs were effectively isolated from remote threats. The remainder of this chapter will challenge that assumption by examining how attackers can successfully target these seemingly self-contained systems. To do that, we must first take a deeper look at both the hardware and software anatomy of these systems.

Understanding the hardware and software anatomy

Carrying out a successful exploit in an automotive microcontroller requires understanding its low-level anatomy both at the hardware and software levels. In this section, we dissect the typical hardware components of an MCU-based ECU (the attack surface from a hardware perspective) and the structure of the software and firmware running on it (the attack surface from a software perspective). This knowledge will act as the foundation upon which specific exploits are built.

> **Note**
>
> Throughout this chapter, the term **software** is used to refer to the applications running within the context of the RTOS. The term **firmware** is used to refer to software components that are not intended for frequent updates, such as the flash bootloader and the HPSE firmware.

Hardware components of an automotive MCU

A traditional automotive ECU consists of a printed circuit board, as shown in *Figure 7.2*, with at least one microcontroller alongside supporting electronic components all enclosed within a hardened case. When the ECU housing is opened, a relatively consistent set of hardware components is exposed.



Figure 7.2: Typical PCB View of an MCU-based ECU [28]

We will survey the most common components found in an MCU-based ECU in the following sections.

Microcontroller

The microcontroller is the primary integrated circuit (IC) on the board and serves as the execution engine for all real-time software running on the ECU. Automotive MCUs commonly include one or more CPU cores that support multiple execution modes, such as privileged and user modes, along with architectural protections like MPU. While the underlying core architecture influences exploit development details, such as whether the target uses a TriCore, ARM, or another CPU instruction set, from an attacker's perspective the MCU core is fundamentally the component responsible for executing injected or manipulated code. Properties such as endianness, word size, and the presence of specialized instructions, including hardware-accelerated cryptographic primitives, become relevant when crafting shellcode or return-oriented programming (ROP) [4] payloads tailored to a specific MCU family.

Memory (Flash, RAM, EEPROM)

Automotive microcontrollers typically integrate on-chip flash memory used to store bootloader, application firmware, and calibration parameters, with capacities ranging from a few hundred kilobytes on low-end devices to several megabytes on higher-end MCUs. On-chip Static Random-Access Memory (SRAM) is used for handling runtime data and is generally more constrained, often ranging from a few hundred kilobytes to a few megabytes. In addition to internal memory, ECUs may include an external serial EEPROM or an external flash device to store special calibration data or vehicle personalization parameters. When enumerating attack surfaces of an MCU-based ECU, we usually pay special attention to external memory interfaces as they act as useful attack targets for extracting firmware and calibration data.

Communication interfaces and buses

Microcontrollers interface with vehicle networks through a variety of communication peripherals. The most common is the Controller Area Network (CAN), implemented via an on-chip CAN controller and an external transceiver. Many ECUs support multiple CAN channels, for example connecting simultaneously to a powertrain network and a body control network that includes the Onboard Diagnostic (OBD) interface. Additional interfaces often include LIN for low-speed sensor and actuator communication, FlexRay in some legacy high-end systems, and increasingly automotive Ethernet for communicating with domain controllers and central compute applications.

From an offensive standpoint, these interfaces define the attack surfaces that enable local and remote attacks. As we saw in *Chapter 6*, an ECU connected to the OBD-II port, for instance, may be reachable through injected CAN traffic using the Unified Diagnostic Service (UDS) protocol.

CAN and Ethernet, further expand the attack surface by providing security-sensitive capabilities like memory read/write access and firmware reprogramming. Beyond in-vehicle networks, some MCUs expose UART or USB interfaces when used in gateways and body control modules, which can serve as entry points if they are insufficiently protected.

Direct I/O and sensor interfaces

Beyond communication buses, the MCU interfaces directly with the physical environment through analog and digital I/O. This includes Analog to Digital Converters (ADCs) for reading sensor signals, digital inputs for switches or pulse-width modulated signals, and outputs used to control actuators such as servo motors. While these interfaces may appear peripheral to cybersecurity concerns, they become critically important once arbitrary code execution is achieved by the attacker on the MCU. Control over these I/O paths allows an attacker to induce physical effects, such as falsifying sensor values or abusing actuators.

Internal peripherals

MCUs integrate a wide range of on-chip peripherals, including high precision timers, watchdogs, communication controllers, ADCs, PWM units, and cryptographic accelerators. The overall security posture of the ECU depends heavily on how these peripherals are configured and controlled. A watchdog timer, for example, is intended to reset the system if software execution stalls, but an attacker may exploit this property to trigger unwanted resets or disable it to maintain persistence.

Many contemporary MCUs also incorporate a Hardware Protected Security Environment (HPSE), typically aligned with SAE J3101 [5], to isolate cryptographic keys and operations from the main application core. These modules can serve as strong defensive assets when properly integrated, but they may also become attractive targets for extracting cryptographic keys if they are poorly designed or hardened. Understanding which security features are present, how keys are provisioned, and whether protections rely on unique or default secrets is a critical reconnaissance step for any attacker.

Debug and programming ports

Interfaces such as Joint Test Action Group (JTAG) or Serial Wire Debug (SWD) are indispensable during development, as they allow engineers to halt execution, inspect memory, and program flash. MCUs offer on-chip debuggers to expedite development and troubleshoot hard-to-reproduce issues. Modern MCUs typically provide locking mechanisms for debug interfaces either through passwords or through cryptographic challenge-response mechanisms, which are usually enforced by the HPSE.

In production systems, these interfaces are intended to be disabled or restricted, but failures to properly disable such interfaces are not uncommon. An exposed or improperly locked debug port provides direct control over the microcontroller, enabling firmware extraction or

modification with minimal effort. Even when debug access is disabled, many MCUs support alternate programming or bootloader modes that listen for firmware update commands over CAN or serial interfaces.

The security of these mechanisms depends on the robustness of authentication and access control mechanisms. The use of weak or global credentials, as well as the support of update paths that lack cryptographic protection, can undermine otherwise strong designs. As we will see in the *Hardware weaknesses* section, abuse of debug and programming interfaces remains one of the most common prerequisites for constructing an ECU exploit.

This brief introduction to the ECU hardware attack surfaces should act as a motivation for offensive security experts to familiarize themselves with the hardware components of these systems including the physical package and PCB, internal and external memory, network interfaces, direct I/O paths, and debug or programming access points. Careful reconnaissance of each of these elements is a crucial step for carrying out a successful hardware-focused attack.

Now that we have established the key hardware building blocks, we will turn our attention to the software anatomy of our target systems.

Software architecture and execution model

As we saw earlier, the software running on automotive MCUs is shaped by hardware constraints and follows a unique layered architecture of its own. Understanding how this software is structured and how it executes is essential when attempting to identify exploitable weaknesses. In most MCU-based systems, we encounter a flash bootloader [6] and a software application stack that leverages an RTOS.

While many ECUs on the market today still rely on proprietary software architectures, these systems still share many of the basic components of AUTOSAR classic, namely the presence of an RTOS, a hardware abstraction layer, middleware for handling memory management and diagnostics, and an application interface layer to enable the sensing, actuation, and various control functions. Given the openness and maturity of the AUTOSAR specifications, we will use AUTOSAR as the reference software architecture model throughout this discussion, with the understanding that the same observations generally apply to non-AUTOSAR-based embedded systems. But before we dive into the application software architecture, let's look at the unsung hero of every ECU, the flash bootloader.

Bootloaders and boot process

MCU-based ECUs often execute code directly from internal flash rather than loading it into RAM. When power is applied or a reset occurs, execution begins in the primary bootloader which is typically stored in a protected region of flash. This protection may be enforced through a hardware flash locking mechanism or a software check that prohibits unauthorized update of the flash bootloader memory region. The initial code performs basic microcontroller initialization, such as configuring clocks, watchdog timers, and essential communication peripherals like CAN, before transferring control to the main application. The bootloader specifically manages peripherals that enable it to perform software updates, which is its primary job.

Bootloaders commonly expose Unified Diagnostic Services routines to support firmware updates, including services such as Request Download (0x34), Transfer Data (0x36), and Request Transfer Exit (0x37) [7]. As shown in *Figure 7.3*, the bootloader firmware typically contains hardware drivers for setup and initialization, communication drivers, a lightweight transport protocol and diagnostic software stack, a lightweight crypto stack, flash drivers, and the bootloader application logic that orchestrates the software updates.

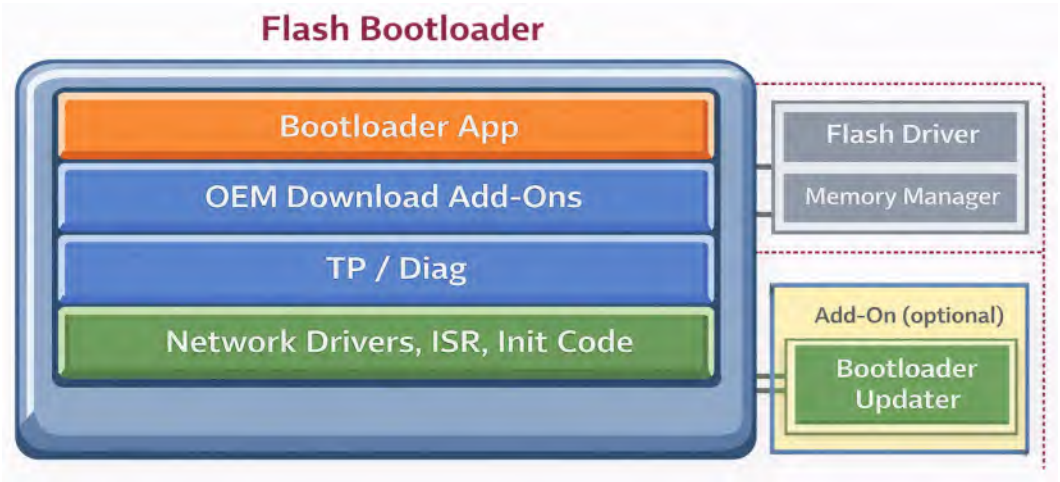


Figure 7.3: Sample flash bootloader architecture

In addition to managing software updates, many ECUs rely on the flash bootloader to handle secure boot for the software application. In that case, the flash bootloader verifies a digital signature or message authentication code over the application software and calibration data before transferring control to the application. In more recent designs, an embedded HPSE may participate in the boot sequence by verifying the integrity of its own firmware, the firmware of the primary bootloader, and that of the application software in a staged manner to meet the startup timing constraints. As we will see in later sections, weaknesses in bootloader design are a common way for attackers to install unauthorized code on the ECU.

HPSE firmware

Many modern MCUs offer an HPSE to enable security use cases such as secure boot, secure communication, cryptographic authentication, and secure software updates. At the heart of the HPSE is a dedicated security processor which has exclusive access to resources such as cryptographic accelerators, secure SRAM, and dedicated flash regions for the purpose of storing cryptographic keys. The HPSE runs firmware that is, by design, meant to be a trusted code base.

Typically, the HPSE firmware contains a simple scheduler or a lightweight RTOS. A hardware abstraction layer enables functions such as flash management, cryptographic engine drivers, and other peripheral drivers that are uniquely under the control of the HPSE. At the boundary of the HPSE lies a shared communication interface to allow the HPSE to exchange data securely with the application software via dedicated registers or designated shared memory buffers. This mailbox system allows the communicating partners to exchange commands or pointers to shared memory that contain the cryptographic job parameters.

An interrupt mechanism typically serves as a doorbell to notify each endpoint that new data has arrived requiring processing, or that a response is ready. When used in the context of an AUTOSAR-based system, the application processor runs a crypto driver [8] to communicate with the HPSE proprietary firmware. This architecture is illustrated in *Figure 7.4* which shows the separation between the host application processor and the HPSE domain.

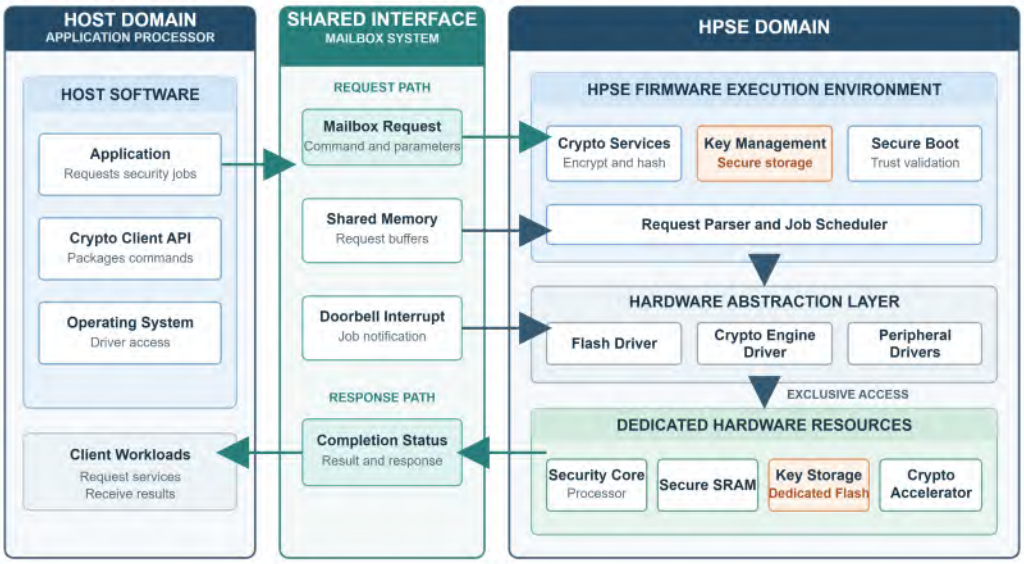


Figure 7.4: HPSE firmware interface to the application processor

Naturally, an attacker would be interested in the shared memory communication protocol between the application and the HPSE firmware to identify potential exploitable security weaknesses. Being able to abuse this protocol can lead to examining cryptographic job information, injecting code into the HPSE execution environment, or even exfiltrating cryptographic keys. Risks concentrate around malformed request parsing, including incorrect handling of job length fields, command identifiers, nested data structures, or pointer-like offsets into shared memory. Shared memory communication also introduces time-of-check versus time-of-use (TOCTOU) risks, where the host modifies buffers after validation but before use by the HPSE processor.

Given that the HPSE firmware likely uses a job scheduler, state machine abuse emerges as another attractive target, such as invoking privileged services in an unexpected lifecycle state or confusing session context across requests. Improper access control can further allow a compromised application processor context to invoke services beyond its intended authorization. Since the HPSE firmware is not standardized, the security of such firmware is highly dependent on the security maturity of the firmware supplier and whether they have designed it with the right trust model.

AUTOSAR basic components

To familiarize ourselves with the software application portion of the MCU-based ECU, we will consider three distinct areas of the AUTOSAR classic architecture as shown in *Figure 7.5*: the

RTOS, the communication services, and the AUTOSAR Runtime Environment (RTE). You are encouraged to study the AUTOSAR specifications for your area of interest as you attempt to enumerate exploitable software services.

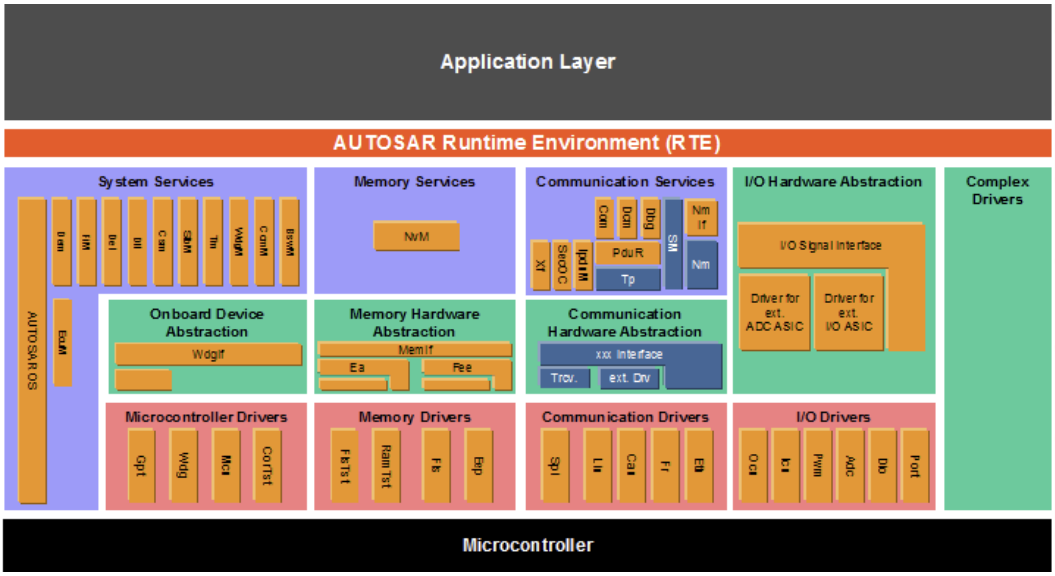


Figure 7.5: The software block diagram of AUTOSAR Classic 4.x (credit autosar.org)

Real-time operating system (RTOS) and task scheduling

Once application execution begins, it typically runs under the control of a real-time operating system such as AUTOSAR OS. This OS is an evolution of the OSEK/VDX standard, designed to provide a statically configured, deterministic execution environment for safety-critical automotive applications. Unlike general-purpose operating systems, the AUTOSAR OS makes most architectural decisions at build time rather than at runtime.

At its core, the AUTOSAR OS enforces separation through the concept of OS-Applications. These represent logical partitions of software with distinct privilege levels, memory regions, and access rights. AUTOSAR OS uses the term "trust" as a property that implies freedom from interference between mixed criticality applications based on a pure safety perspective. This can be confusing as security engineers may assume that a Trusted Application is immune to certain attacks when in fact it just means that this application has a set of isolated memory and hardware resources due to safety considerations. Attacks against AUTOSAR-based ECUs may, in theory, originate from a "Trusted Application" if that application is the one that is exposed to the ECU attack surface and contains exploitable security weaknesses.

Note

The motivation for understanding how the AUTOSAR OS enforces isolation is twofold: to help with discovering configuration weaknesses that result in overly permissive sharing of memory and resources, and to aid in the design of successful exploits that abuse such weaknesses.

Task scheduling and preemption

The AUTOSAR OS uses a fixed, priority-based scheduling model. Task priorities are assigned statically during system configuration and cannot be changed at runtime. This means execution order is predictable, which is beneficial for real-time guarantees but also highly informative to an attacker performing reconnaissance. Tasks are divided into two classes. Basic Tasks cannot block and move only between Suspended, Ready, and Running states. Extended Tasks can wait on OS events and therefore have a richer execution lifecycle.

Scheduling is fully pre-emptive which means that whenever a higher-priority task becomes Ready, it immediately displaces the currently running lower-priority task. In multi-core systems, each core runs its own independent scheduler with tasks and Interrupt Service Routines being statically bound to a specific core. From an attack perspective, this deterministic binding may aid in the design of software exploits that rely on reliable timing behavior.

Interrupt service handling

Interrupt service routines (ISRs) occupy a privileged position in the AUTOSAR OS execution model. All interrupts are treated as having higher priority than a task, regardless of task priority configuration. This alone makes interrupt handling code a high-value target. AUTOSAR OS defines two types of interrupts as illustrated in *Table 7.1*.

Category 1 ISR	**Category 2 ISR**
Bypasses the AUTOSAR OS entirely and executes directly when the interrupt is issued.	Is wrapped and managed by the AUTOSAR OS, so execution passes through OS-controlled interrupt handling.
Designed to minimize interrupt latency.	Designed to balance interrupt handling latency with OS control and protection.
Has higher priority than tasks, regardless of task priority configuration.	Has higher priority than tasks, regardless of task priority configuration.

Category 1 ISR	Category 2 ISR
Runs outside the scope of OS-enforced memory, timing, and service protection.	Can be subject to OS-enforced memory, timing, and service protection policies.

Table 7.1: Differences in the handling of Cat1 and Cat2 interrupt service routines

As a result of how AUTOSAR OS manages interrupts, any vulnerability in a Category 1 ISR effectively executes with minimal OS oversight, making it an attractive target.

In contrast, Category 2 interrupts are wrapped by the OS, which in turn applies service and timing protection policies. Furthermore, many MCUs offer a relocatable interrupt vector table to allow the flash bootloader and the application to manage separate interrupt service routines, each within their own execution domain. If an attacker can relocate the interrupt vector table offset, then interrupts can be redirected to attacker-controlled code. This makes the knowledge of vector table layout and memory placement, which can be derived from datasheets or firmware analysis, a critical step in exploiting interrupt service routines.

Privileged and user mode execution

The AUTOSAR OS safety model relies heavily on hardware-enforced processor modes. "Trusted" OS-Applications execute in privileged mode and have dedicated access to memory, peripherals, and the full OS API. Since MCUs do not have the rich separation of privilege levels as in an SoC, it is common for both the operating system and the "Trusted" OS application to run in the same trusted context as they share the same privilege level!

"Non-Trusted" OS-Applications are typically executed in user mode. Their access to memory regions and hardware control registers, such as MPU configuration or system timers, can be restricted usually for ensuring freedom from interference. This separation is the primary mechanism used to contain faults and limit the impact of misbehaving application code.

Memory protection and application isolation

Memory isolation in AUTOSAR OS implementations is typically enforced using a Memory Protection Unit. The MPU partitions the address space into regions with distinct access rights tied to the currently executing OS-Application. *Figure 7.6* shows an example partitioning between two tasks belonging to different OS applications. Both flash and RAM regions can be restricted based on which task has the execution context.

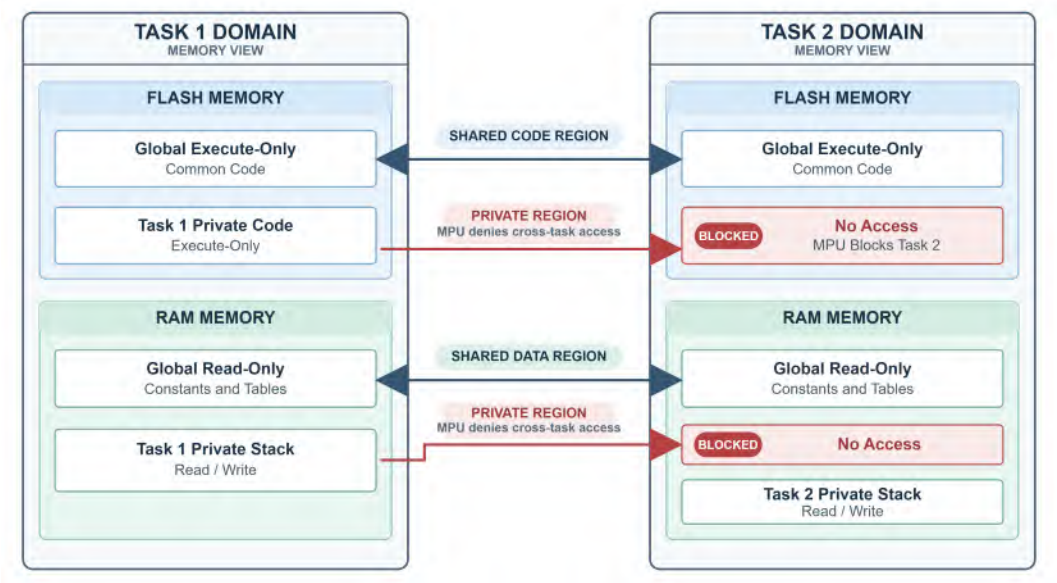


Figure 7.6: Memory map model of MCU-based system

The OS prevents one OS application from writing to data sections of another OS application. Stack regions are also isolated where hardware support exists. In systems without sufficient hardware protection, the OS may provide stack monitoring as a best-effort mechanism by checking for overflows during context switches rather than preventing them outright.

In addition to raw memory protection, AUTOSAR enforces ownership of OS objects such as Alarms, Events, and Schedule Tables. Each object belongs to a specific OS-Application where attempts to access objects without proper ownership result in service protection violations.

Temporal isolation and timing protection

Beyond spatial isolation, the AUTOSAR OS enforces temporal isolation to prevent fault propagation across tasks. This is implemented through timing protection mechanisms that define explicit task execution budgets.

Each task or Category 2 interrupt can be assigned an execution time budget. Additionally, lock budgets limit how long a task or interrupt may hold shared resources or keep interrupts disabled. Finally, time frames enforce minimum inter-arrival times to protect against interrupt storms or misbehaving triggers. This is especially useful if an externally controlled interface is permitted to trigger interrupts.

Note that violations of these constraints are handled through Protection Hook. Depending on how the software is configured, the OS may terminate the offending task, terminate the entire

OS-Application, or initiate a full system shutdown. From a security standpoint, this response mechanism represents both a containment strategy and a potential denial-of-service vector. An attacker who can deliberately trigger Protection Hook policies may be able to repeatedly reset or degrade system functionality.

Peripheral access and communication services

At the lowest level of the AUTOSAR Classic stack lies the **Microcontroller Abstraction Layer** (**MCAL**). MCAL drivers provide direct access to on-chip peripherals such as SPI interfaces, CAN controllers, LIN interfaces, Ethernet MACs, and UARTs. These drivers are tightly coupled to the specific microcontroller and expose low-level configuration parameters, including CAN acceptance filters, interrupt behavior, DMA usage, and buffer sizing.

From an attack perspective, MCAL forms the first software attack surface that is exposed to malformed or unexpected hardware inputs and network traffic. Weak or overly permissive filtering at this layer can allow messages to propagate upward that were never intended to reach higher software layers, effectively expanding the attack surface before any protocol-level or application-level validation occurs.

Above MCAL, the AUTOSAR communication stack introduces progressively higher levels of abstraction. Network-specific modules such as the CAN Interface, LIN Interface, and Ethernet Interface normalize access to different buses and decouple upper layers from hardware details. These modules feed into services that operate on PDUs, signals, or sessions rather than raw frames. While this abstraction simplifies application development and portability, it also shifts many security-relevant decisions into static configuration artifacts generated during integration. Once deployed, these configurations are typically assumed to be correct and remain unchanged for the lifetime of the ECU.

At the upper layers of the stack, directly beneath the application layer, AUTOSAR defines several communication services that are particularly relevant from an offensive security standpoint:

- **Communication (COM)**: The COM layer enables signal-based data exchange between software components. It defines which signals are transmitted or received, how they are packed into Protocol Data Units (PDUs), and when transmissions occur. AUTOSAR configuration parameters control CAN identifiers, transmission periodicity, signal update semantics, timeout monitoring, and group activation. Overly broad configurations can cause ECUs to process CAN messages from unintended senders, or transmit messages not intended for external consumers. For an attacker with bus access, predictable periodic transmissions combined with weak timeout handling can be leveraged for replay, spoofing, or task-level denial-of-service.

- **Network management (Nm)**: This layer governs how an ECU participates in network sleep and wake-up coordination. The Nm module determines when communication channels are active, how long they remain awake, and which messages signal network activity. Incorrect Nm configuration can allow attackers to keep ECUs awake indefinitely, drain vehicle batteries, or force repeated transitions between operational states. In tightly coupled networks, these effects can propagate beyond a single ECU, impacting multiple network domains.
- **Secure onboard communication (SecOC)**: For systems that implement message authentication, SecOC adds cryptographic protection to selected PDUs. Thsi protocol introduces freshness values, and authentication tags to counter replay attacks and message spoofing. While this significantly improves network resilience, the complexity of the implementation can introduce ways for overcoming these protections. Weak key management, possibility for rollback of freshness counters, or partial deployment where only a subset of messages is protected can result in inconsistent trust assumptions. From an offensive perspective, looking for weaknesses in SecOC simply shifts the goal post from circumventing an insecure CAN protocol to circumventing a poorly implemented SecOC software solution.
- **Diagnostic communication manager (Dcm)**: This protocolexposes diagnostic functionality to onboard and offboard diagnostic clients. Dcm acts as the relay between incoming diagnostic requests and internal application callbacks that implement UDS services. Each diagnostic service is statically mapped to handler functions and guarded, if at all, by session and security checks. Basically, a diagnostic service may be permitted in programming sessions but prohibited in the default session. Similarly, a diagnostic service may be allowed if a certain security level has been activated and denied otherwise. Misconfiguration in this area is a particular source of risk. Diagnostic routines that lack proper authentication, or improper preconditions for session control can allow attackers to invoke sensitive operations, including memory access, calibration updates, ECU resets, or entry into firmware programming mode under unsafe conditions. Because Dcm is explicitly designed to accept external input, it frequently represents one of the richest and most reliable attack surfaces in an AUTOSAR-based ECU.
- **Ethernet**: With the rise of Ethernet use, it is common to find MCU-based ECUs that support a full networking stack including the Ethernet drivers, ARP, ICMP, IPv4, IPv6, TCP, UDP, DHCP and TLS. The presence of these software components should alert an offensive security engineer to look for known vulnerabilities or configuration mistakes that can expand the attack surface against the target ECU.

Table 7.2 provides a list of the common AUTOSAR classic modules that support a networking capability making them a useful target for security analysis.

Module	**Description**
Com (COM)	Packs/unpacks application signals to/from I-PDUs and exposes a signal-oriented API to applications.
PduR (PDU Router)	Routes PDUs between upper modules (e.g., COM, DCM) and bus/bearer-specific modules, enabling gatewaying.
Dcm (Diagnostic Communication Manager)	Handles network-independent diagnostics (UDS), dispatching services and interfacing with transport/bearer layers.
ComM (Communication Manager)	Arbitrates ECU communication modes (full/no/silent) and coordinates communication availability for users.
BswM (BSW Mode Manager)	Rule-based coordinator that manages BSW and application modes and triggers configured actions.
Nm (Network Management)	Coordinates transitions between normal operation and bus-sleep across networks, independent of the bus.
UdpNm	Network management over UDP for IP/Ethernet clusters to coordinate communication and sleep states.
SoAd (Socket Adapter)	Bridges AUTOSAR PDU world to the socket-based TCP/IP stack, mapping I-PDUs to sockets.
DoIP (Diagnostics over IP)	Transports UDS diagnostics over IP (typically TCP) and integrates with SoAd/TcpIp.
TcpIp (TCP/IP stack)	AUTOSAR TCP/IP stack implementing IPv4/IPv6, UDP, TCP and related protocols (ARP, ICMP, DHCP, NDP, TLS).
EthSM (Ethernet State Manager)	Controls Ethernet controller/transceiver operating states and reports link state via EthIf.
EthIf (Ethernet Interface)	Hardware-independent interface between upper layers and Ethernet MAC/PHY drivers.

Module	Description
Eth (Ethernet driver)	MCAL driver providing uniform access to Ethernet MAC hardware for EthIf.
EthTrcv (Ethernet Transceiver Driver)	MCAL driver abstracting PHY transceivers and exposing a standard interface upstream to EthIf.

Table 7.2: List of AUTOSAR modules that must be analyzed when looking for network and state related weaknesses

Note

You are encouraged to dive deeper into the AUTOSAR specifications (`https://www.autosar.org/search?tx_solr%5Bfilter%5D%5B0%5D=category%3AR25-11`) to understand the architecture of each module and how it can be misconfigured to introduce a security weakness.

AUTOSAR runtime environment (RTE)

The AUTOSAR **Runtime Environment** (**RTE**) is the standardized middleware layer defined by AUTOSAR that decouples application software components from the underlying ECU hardware and Basic Software (BSW). Its primary purpose is to enable portability, reusability, and integration of application software across different ECUs and hardware platforms without requiring changes to application logic.

The AUTOSAR architectural model relies on the RTE to provide a uniform communication abstraction between Application Software Components (SW-Cs) and the AUTOSAR BSW. This abstraction is meant to enable OEMs and Tier-1 suppliers to independently develop software components while relying on the RTE to resolve communication bindings at system integration time.

Incidentally, the RTE is not dynamic middleware. Instead, it is statically generated from system descriptions (ARXML) during the build process to cover core architectural responsibilities such as the following:

- **Intra-ECU communication**: Function calls or memory-based data exchange between SW-Cs mapped to the same ECU
- **Inter-ECU communication abstraction**: Transparent access to signals and services that are ultimately transported via COM, PDU Router, and network stacks (CAN, LIN, FlexRay, Ethernet)

- **Runnable invocation**: Mapping of application runnables to OS tasks, events, and timing constraints
- **Mode and service access**: Controlled access to ECU modes, diagnostics, NVM services, and other BSW functionality

Note

ARXML (AUTOSAR XML) is a standardized, XML-based file format used in the AUTOSAR ecosystem to describe and exchange automotive software architecture, system topologies, and ECU configurations. It allows multiple OEMs and suppliers to collaborate by transferring design data and communication parameters across diverse development tools from different tool vendors.

From a functional safety perspective, the RTE directly supports ISO 26262 goals by enforcing strict interface contracts between software components. Communication between SW-Cs occurs only through statically defined ports and interfaces, which prevents unintended data access and reduces the risk of accidental coupling. When combined with OS-level memory protection and task isolation, the RTE contributes to freedom from interference by ensuring that components interact only through approved pathways. The elimination of dynamic binding and runtime memory allocation further enables deterministic execution, a prerequisite for building safe systems. In addition, the RTE provides consistent communication boundaries where safety mechanisms such as end-to-end protection can be applied in a uniform manner.

From a security perspective, the RTE prevents implicit communication channels from unintended communication partners. Software components do not exchange data unless an explicit sender–receiver or client–server relationship has been configured. This constraint reduces unwanted communication and simplifies reasoning about data flow. The RTE also avoids dynamic middleware features commonly found in richer execution environments, such as runtime service discovery or dynamic routing. By doing so, it reduces code complexity and narrows the attack surface associated with dynamic behavior.

Of course, one could argue that without proper memory isolation between Application SW-Cs and their respective data objects, an SW-C can access a data port belonging to another SW-C by discovering the memory address of that data object. In practice, the AUTOSAR RTE should be viewed as an enabler rather than an enforcer of security. It constrains how software components interact, but it does not protect those interactions. Nevertheless, understanding the inner workings of the RTE is important when reverse engineering a system and attempting to identify critical data objects and interfaces that are candidate attack targets.

Calibration and debugging code

Beyond the main control logic that governs vehicle behavior, many MCU-based ECUs contain auxiliary software intended for calibration, validation, and debugging during development and testing. While these capabilities are essential for engineering workflows, they frequently persist, partially or fully into production, where they can significantly expand the attack surface if not tightly controlled.

Calibration protocols and memory access

The Universal Measurement and Calibration Protocol (XCP) is standardized to provide calibration tools with direct read and write access to ECU memory over a vehicle communication bus. Unlike UDS, which is primarily intended for diagnostics and logging faults in the field, XCP is designed for development and production environments. It supports high-rate measurement, online calibration, and, in many implementations, flash programming of ECU software.

A common use case is the tuning of control algorithms such as braking, powertrain, or chassis control for a specific vehicle configuration. Calibration parameters including vehicle mass, wheelbase, tire characteristics, and controller gains are exposed through an A2L description file. Calibration engineers use XCP to modify these parameters in real time while the ECU is in operation mode. Once acceptable behavior is achieved, the finalized values are typically committed to non-volatile flash memory. Many XCP stacks also support ECU reprogramming, allowing firmware images to be downloaded over the same interface rather than through a dedicated bootloader protocol for convenience during development.

From an offensive security standpoint, the risk is straightforward. If calibration services remain enabled in production builds without strong authentication and access control, an attacker with bus access can potentially read sensitive memory regions or modify control variables at runtime. Even if write access is limited, read access alone can leak proprietary data that can be used to stage further attacks. Similarly, flash programming access can be abused to install malicious software.

Debug modes and hidden entry points

In addition to formal calibration protocols, some ECUs implement special debug or development modes that can alter normal execution behavior. These modes may relax access controls, enable additional command handlers, or expose internal state for diagnostic purposes. In some historical cases, ECUs have been shown to enter undocumented backdoor states when a specific hard-coded value in a diagnostic request was received, effectively unlocking privileged functionality [9].

Such mechanisms are rarely advertised and are often remnants of development workflows. For attackers, they represent high-value targets. By reverse-engineering firmware images, an analyst can identify suspicious message identifiers, dormant command tables, or conditional branches that appear to gate privileged behavior [10]. Undocumented command handlers can indicate the presence of hidden debug paths that were never fully removed.

AUTOSAR DLT

In AUTOSAR Classic systems, debugging and observability are often implemented using the Diagnostic Log and Trace (DLT) module [11]. DLT is designed to provide structured logging, tracing, and runtime diagnostics from ECUs to external tools over communication channels such as CAN or Ethernet.

While DLT is invaluable during development, it introduces a distinct attack surface when left enabled in production. DLT messages can expose internal software states, execution paths, timing information, and error conditions. Verbose log levels and unfiltered application contexts can inadvertently leak sensitive implementation details that significantly reduce the effort required for reverse engineering.

Now that we have a reasonable grasp on the hardware and software architecture of MCU-based ECUs, it's time to dig into the common weaknesses that make exploiting such systems possible.

Common hardware and software weaknesses in MCU-based ECUs

MCU-based ECUs exhibit a range of security weaknesses that broadly fall into two categories: hardware-level weaknesses related to physical and electrical exposure, and software-level weaknesses stemming from implementation flaws or improper design assumptions. In practice, successful attacks often combine both. A hardware technique may be used to extract firmware, which then enables detailed analysis and exploitation of software vulnerabilities. Given the nearly limitless ways systems can be designed, it is not feasible to enumerate every possible flaw that can be exploited. Instead, this section focuses on the most common security weaknesses that repeatedly appear in real-world attack chains against MCU-based ECUs.

Hardware weaknesses

As seen in *Figure 7.7*, hardware weaknesses range from insecure hardware debug interfaces to lack of encrypted memory. In this section, we will dive into each category of security weaknesses and learn how they make it possible for security exploitation.

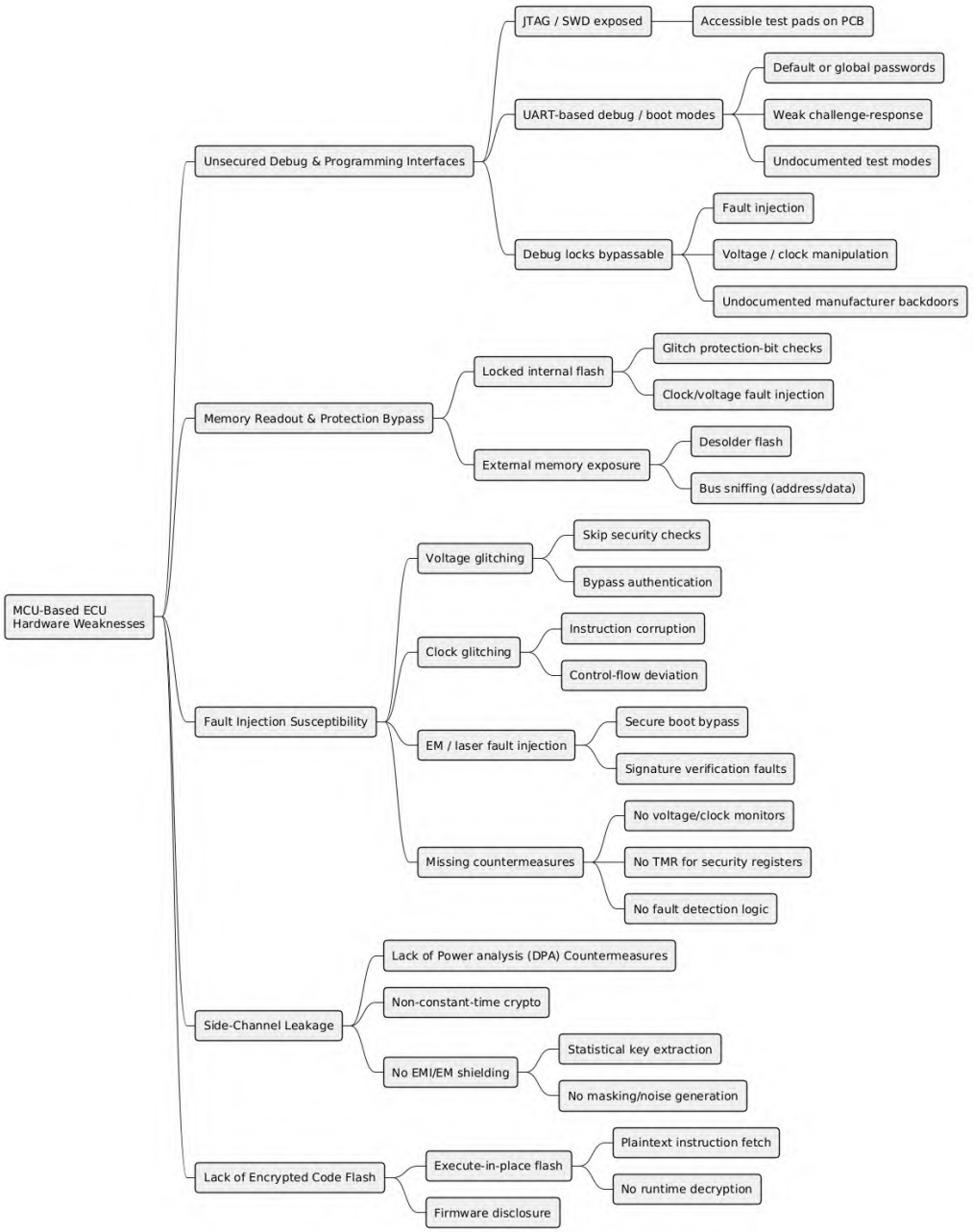


Figure 7.7: A snapshot of common hardware weaknesses

Unsecured debug/programming interfaces

One of the most reliable entry points in an attack chain is an unsecured hardware debug or programming interface. Interfaces such as JTAG and SWD are routinely included in ECU designs to support development, manufacturing, troubleshooting, or failure analysis. In production hardware, these interfaces may remain accessible through populated connectors, exposed test pads, unpopulated footprints, or traces leading to the relevant device pins. Even if test pads have been removed, an attacker with physical access may be able to trace the signals back to accessible vias, components, or package pins.

Once the relevant signals are identified, access does not necessarily require constructing a dedicated test fixture. A tester may connect directly by soldering wires to accessible points or, where repeated access is needed, use a probing fixture. Debug or programming access may also remain exposed through alternative paths, such as UART-based bootloader modes or other serial recovery interfaces. If these interfaces are not disabled, authenticated, or otherwise protected in production, they may allow firmware extraction, memory inspection, reprogramming, or bypass of software-enforced security controls.

With access to unprotected JTAG or SWD, an attacker can halt the CPU, read memory contents, and extract firmware for offline analysis. With the firmware in hand, the attacker can analyze it for vulnerabilities, create a malicious patch, and reflash it on the target for further discovery and exploitation. This often leads to the recovery of hard-coded secrets, which in turn proves useful for constructing more sophisticated attack chains. That is why from an offensive security perspective, debug interface assessment is a common first step during the discovery phase.

UART-based debug paths are similarly dangerous. These often rely on special command sequences to enter a programming or debug mode and may be protected by a password or challenge-response mechanism. In multiple documented cases [34], production ECUs have shipped with these interfaces without sufficient protection. Typical weaknesses include the usage of default or global passwords, relying on the obscurity of undocumented debug and test modes, or weaknesses in the cryptographic implementation of the challenge-response mechanism.

Even when hardware lock bits or debug disable flags are set, determined attackers have demonstrated successful bypasses through fault injection, voltage or clock manipulation, or discovery of undocumented manufacturer backdoors [29].

Memory readout and bypass of protection

When debug ports are properly locked, attackers often attempt to extract the firmware from the ECU's non-volatile memory. Many MCUs have "code protection" OTP bits that prevent directly reading the internal flash contents after the MCU has entered production. However,

hardware hackers have developed techniques to bypass these. One approach is voltage glitching or clock glitching at the moment the MCU checks the protection bit, causing a faulty read event to allow access (more on glitching later).

Another approach is to exploit manufacturer test modes. For example, some microcontrollers have a factory test pin or mode that, if activated (usually requiring decapsulating the chip and connecting to tiny pads), will dump the internal flash contents. Additionally, if the MCU relies on an external memory chip to store code or data, then that becomes an easy target by desoldering the chip or sniffing the address and data buses. Getting a hold of the ECU code and calibration data does not always require special reverse engineering skills. In fact, some OEMs offer software images through their dealership portals, with the only requirement being knowledge of the vehicle VIN number!

Susceptibility to fault injection (glitching)

Fault injection involves deliberately introducing faults in the MCU's operation to make it do something unintended. Common methods are voltage glitching (briefly dropping the supply voltage) or clock glitching (introducing a spurious clock pulse) at precise moments. Other methods include electromagnetic fault injection pulses (EMFI) or even laser shots to perturb the chip. The goal is often to skip an instruction or have the CPU execute incorrectly. For example, consider code that checks a password before allowing execution to continue:

```
"if(password_correct)
proceed;
else reset;"
```

A carefully timed glitch could flip the condition or skip the instruction, allowing an attacker to execute their desired code path without providing a correct password. Researchers have repeatedly demonstrated their ability to inject glitches during the secure boot process of a critical instruction, for example to corrupt the signature verification check and gain code execution privileges through their modified code [30]. As we saw in *Chapter 3*, modern tools for glitching have become quite accessible, making this no longer the realm of nation-state labs only. In fact, hobbyists and researchers alike have achieved successful MCU glitch attacks with open-source hardware like ChipWhisperer for side-channel analysis and fault injection, and ChipSHOUTER for electromagnetic fault injection [31].

The bottom line is that fault injection completely breaks assumptions of trust in the device's hardware behavior: with the right glitch, security steps can be bypassed. While fault injection is not a weakness by itself, lack of any defensive measures against fault injection is considered a weakness. For example, lack of support for clock and voltage monitoring circuits that are tuned for detecting malicious glitches is a weakness.

For security-critical registers exposed to fault-injection threats, the absence of hardware redundancy such as Triple Modular Redundancy (TMR) may be considered a weakness, depending on the threat model and other implemented countermeasures. TMR can mask certain single-point transient faults by majority voting which can be useful for glitch detection. Note that TMR does not protect against all glitch or fault-injection scenarios, especially faults affecting multiple replicas or the voter. The bottom line is that MCU suppliers who are aware of the effectiveness of glitch attacks and are not offering any hardware protection are effectively producing inherently weak microcontrollers.

Side-channel leakage (Timing/Power/EM analysis)

Side-channel attacks exploit physical leakage from a microcontroller while it performs computations to infer confidential information. The most relevant targets are cryptographic secrets used by ECUs for message authentication, secure boot, and diagnostic access. The impact of successful side-channel key extraction can be especially severe when an ECU relies on long-term cryptographic keys that are rarely rotated or are shared across a vehicle model or class of vehicles. In such cases, recovering a single key may enable repeated attacks against the same ECU or attacks against other ECUs that rely on the same key material. This makes the abuse of side-channel leakage especially attractive to attackers who want to recover such secrets as part of a fleet-wide attack chain.

Without side channel mitigations in hardware such as Electromagnetic (EM) leakage and Electromagnetic Interference (EMI) shielding, attackers observing EM traces across repeated cryptographic operations and applying statistical analysis, can recover cryptographic material. Similarly, differential power analysis (DPA) can be applied by measuring changes in power consumption while the cryptographic operation is underway. Timing side channels pose a similar risk when authentication or encryption routines are not implemented in constant time, allowing execution time differences to reveal information about the used keys.

These weaknesses are not theoretical. Real-world attacks against immobilizer ECUs have demonstrated that cryptographic keys used to authenticate ignition keys can be extracted through power analysis, effectively defeating anti-theft systems [32]. In fact, any cryptographic operation, including message authentication code (MAC) generation, can leak information if the underlying hardware and firmware do not actively mitigate side-channel exposure. Many MCUs lack built-in support for masking, noise generation, or constant-time execution within their cryptographic accelerators. Those crypto engines are often optimized for throughput and determinism rather than leakage resistance.

Timing side-channel analysis is more difficult when cryptographic operations are performed entirely inside the HPSE, because the hardware separation limits what an attacker can observe from the general-purpose CPU. However, even though HPSEs are now widely available in automotive MCUs, some ECU suppliers still perform certain cryptographic operations in

software on the main CPU cores, often for performance or integration reasons. Those software-based implementations may remain exposed to timing side-channel attacks.

An attacker with prolonged physical access to an ECU can further increase success rates by modifying firmware to insert markers or delays around cryptographic operations, making it easier to align measurements or target electromagnetic fault injection and side channel analysis. In this context, the absence of hardware-level side-channel defenses turns physical possession into a powerful advantage. Similar to fault injection, MCU suppliers may omit side-channel analysis mitigations to avoid expensive intellectual property licensing fees, performance penalties, and silicon area overhead.

Lack of encrypted memory support

As mentioned earlier, a defining characteristic of most automotive MCU architectures is that application code is executed directly from embedded flash memory. Instructions are fetched in place rather than being loaded into RAM at boot time. While this execution model simplifies design and supports deterministic startup behavior, it has a significant security implication: encrypted code storage is effectively unsupported without introducing external memory components.

Because the CPU must fetch and decode instructions directly from flash, the contents of flash must be available in plaintext at execution time. Unlike higher-end compute platforms that can decrypt code pages on the fly as they are loaded into memory, MCUs lack hardware mechanisms for transparent instruction decryption. Once an attacker gains the ability to read flash content, whether through a debug interface or physical extraction, the entire firmware image becomes immediately intelligible without any need to recover additional cryptographic keys.

This architectural limitation significantly lowers the barrier for reverse engineering as a raw memory dump is sufficient to expose application logic, diagnostic handlers, calibration tables, security algorithms, and protocol state machines. This visibility accelerates vulnerability discovery and enables precise targeting of attack surfaces such as weak authentication routines or backdoor functionality that would otherwise be obscured.

Some MCU-based ECUs support external flash devices, which in theory could be encrypted. However, this approach introduces additional cost, board complexity, and performance constraints, and is therefore uncommon in cost-sensitive automotive ECUs. From an offensive security perspective, this makes MCU-based ECUs particularly vulnerable to attacks that rely on firmware understanding, including protocol abuse, downgrade attacks, and the discovery of latent debug or manufacturing functionality.

As we have seen so far, while hardware weaknesses require physical proximity or contact, they are extremely powerful in recovering secrets and confidential data, which is a steppingstone

for more sophisticated software-based attacks. In the next section, we will survey some of the common software weaknesses that are abused by attackers aiming to penetrate the MCU-based ECUs.

Software weaknesses

As seen in *Figure 7.8*, software weaknesses range from insecure bootloaders to flaws in key management and cryptographic implementations. In this section, we will dive into each category of security weaknesses and learn how they enable security exploitation.

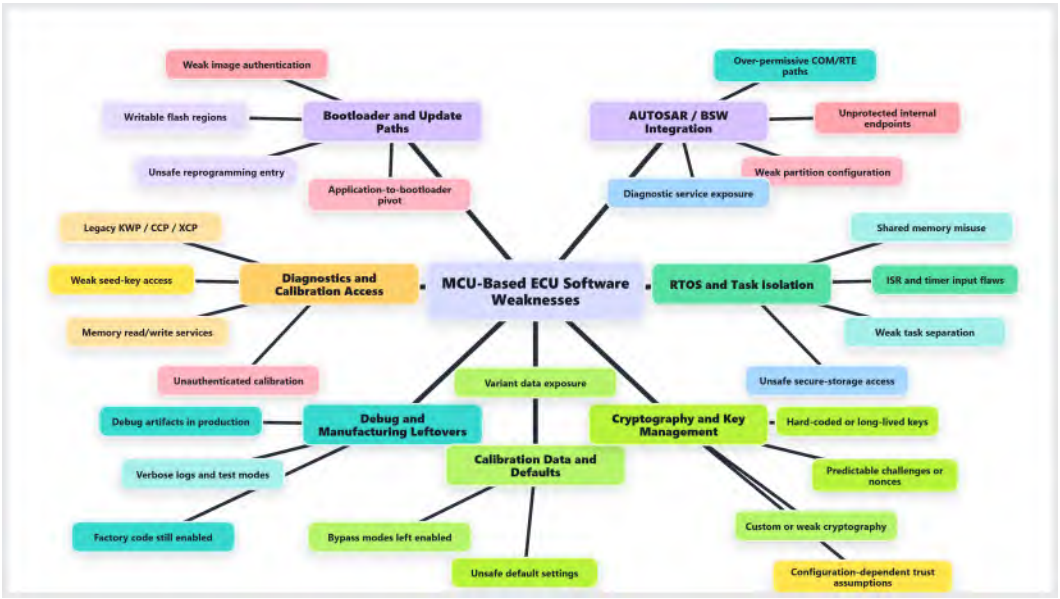


Figure 7.8: Snapshot of common software weaknesses in MCU based ECUs

Insecure bootloaders and firmware update mechanisms

As we saw in the *Bootloaders and boot process* section, flash bootloaders have two critical responsibilities: managing the boot of application software and managing software updates. This makes bootloaders one of the most security-critical components in an MCU-based ECU. They execute at the highest privilege level, run early in the startup sequence, and directly control what application code is allowed to persist on the device. Weaknesses in this layer routinely translate into full and persistent compromise.

When tasked with secure boot, strict startup time requirements and limited memory resources often require optimizations at the expense of cybersecurity. For example, to meet power-on timing deadlines, some implementations verify only the memory regions considered essential for startup and control flow. These design tradeoffs can create security gaps that attackers may try to exploit as we see below.

Common flash bootloader weaknesses include the following:

- **Incomplete integrity coverage**: Calibration data or other non-code regions may be excluded from secure boot checks under the assumption that they do not affect control flow. From an attacker's perspective, these exclusions can create writable regions that influence system behavior without triggering a secure boot failure.
- **Shared symmetric verification keys**: Some flash bootloaders rely on a symmetric key to verify the authenticity of the application image before handing over execution control. If all ECUs of a specific model and ECU type share the same symmetric key, extracting that key from one ECU may allow an attacker to generate a valid MAC for modified images on other matching ECUs.
- **Exposure of cryptographic key material**: Because these shared keys are highly valuable, flash bootloaders become attractive targets for side-channel analysis, fault injection, and other key extraction techniques. Once the symmetric key is recovered, an attacker may be able to create an image that passes authentication and boots on the target ECU.

A successful bypass of secure boot can become a steppingstone for additional discovery work. For example, attacker-controlled firmware may allow deeper inspection of ECU behavior, extraction of secrets, or identification of weaknesses that could later be exploited remotely.

Firmware update mechanisms further complicate the picture. Typically, automotive ECUs support firmware updates via UDS diagnostic protocols. On many MCUs, flash programming requires that flash drivers execute from RAM while flash banks are being erased or written. This creates periods where code execution from RAM is allowed from ".data" sections, while memory permissions are relaxed, increasing the attack surface during update operations. Note that mechanisms like Data Execution Prevention (DEP) would not be usable in these cases because the flash routines are designed to be executed from RAM while flashing software.

Bootloader implementation flaws amplify these risks. Because bootloaders process externally supplied data during firmware updates, they are exposed to classic memory corruption vulnerabilities such as stack overflows or improper bounds checking on incoming data packets. These vulnerabilities are particularly valuable to attackers because bootloaders execute in privileged mode and are rarely updated. The bootloader is, by definition, the mechanism used to update everything else. A single exploitable flaw can therefore enable persistent compromise, allowing an attacker to implant backdoors, disable security features, or exfiltrate the complete firmware image.

Aside from memory safety bugs, bootloaders might have logical flaws, such as accepting an unsigned image if certain flags are set or entering flash programming mode if a specific CAN message is received during startup. If entering the programming session and issuing download

commands is not gated by a strong cryptographic authorization, any node on the CAN bus can potentially reprogram the target ECU. Additional complexity arises from performance optimizations in the update process. Bootloaders often include decompression libraries to accelerate firmware downloads by transferring compressed images over the network. These libraries introduce their own attack surface. If decompression occurs before authentication or integrity verification, attackers can exploit this ordering through crafted payloads such as compression bombs, leading to memory exhaustion or buffer overflows.

A final weakness to consider is the ability to jump into the bootloader from a compromised application. Since the flash bootloader is still present while the application is running, a code injection attack in the application can redirect execution to the flash bootloader's memory space, either through a direct branch instruction or through manipulating global memory flags that indicate enablement of boot mode after issuing a soft reset. The bootloader then assumes that a valid authentication sequence has taken place by the application in the previous cycle and may allow a non-authorized diagnostic client to perform a software download session.

Taken together, insecure bootloaders and firmware update mechanisms combine several high-risk properties: privileged execution, direct exposure to external input, complex state transitions, and long-lived code that is difficult to patch. For an offensive security engineer, this layer is often one of the most attractive targets, as a single weakness can yield durable, device-wide compromise with minimal need for chaining additional vulnerabilities.

AUTOSAR-related weaknesses

As we saw in the *Software architecture and execution model* section, the ECU software is structured as a layered system with the bootloader at the bottom and the application at the top while AUTOSAR plays the role of the OS, MCAL and middleware services. A weakness in any of these layers can provide an attacker with an initial foothold. Because code is fetched directly from flash, exploit mitigations common in general-purpose systems such as address space layout randomization (ASLR) are not available. Some defenses, including data execution prevention (DEP) and stack canaries, are increasingly used, but their coverage is inconsistent. As a result, MCU-based ECUs remain susceptible to classic memory corruption weaknesses, particularly because they operate with a flat memory model and direct physical memory access. The lack of virtualization and limited isolation makes the consequences of such flaws more severe than in systems with stronger separation of privileges.

Across AUTOSAR Classic systems, improper configuration emerges as the dominant risk factor shaping the ECU's security posture. Decisions about which CAN identifiers are accepted, which diagnostic services are protected by access control restrictions, which communication interfaces are left enabled in production, and which messages are authenticated can be detrimental to security if improperly chosen. Therefore, when looking for weaknesses in AUTOSAR based systems, it pays to inspect the AUTOSAR configuration. Examples of weak

configuration include overly permissive CAN acceptance filters, unrestricted diagnostic services, unsecured TCP/IP endpoints on Ethernet-capable ECUs, and network management parameters that allow unintended wake-up behavior.

The widespread adoption of AUTOSAR Classic introduces an additional systemic risk in the form of architectural monoculture. A vulnerability discovered in a standardized module such as the CAN driver, transport layer, or memory management stack can propagate across vehicle platforms and manufacturers especially when the software supplier is the same.

While AUTOSAR Classic contributes to the simplification of the development and porting of ECU software, standardization itself can become a liability from a security perspective. Because module and service identifiers, configuration formats such as ARXML, and the structures of Basic Software and the RTE are public and consistent across suppliers, an attacker who obtains a firmware dump can easily parse and identify communication stacks, diagnostic modules, crypto drivers, and memory-management services. This draws our attention toward implementation flaws particularly in AUTOSAR modules with network exposure such as network drivers (CAN, LIN, Ethernet), the transport layer, and diagnostic message handlers. Typical secure coding violations like missing input validation, buffer overflows, integer overflows, and others are especially worrisome there because most software runs in supervisor mode with limited access separation.

As we saw in the *Software architecture and execution model* section, the operating system defenses are typically designed to address safety concerns to prevent non-malicious corruption and freedom of interference between ASIL and QM code. Without carefully enforced MPU region partitioning, a single buffer overflow in a peripheral-facing task, such as a CAN message handler, may corrupt memory across the entire ECU and interfere with safety-critical control logic. Safety-critical code itself may still contain memory corruption bugs, and once execution reaches that domain, MPU-based isolation may not offer sufficient protection. As a result, attackers can defeat freedom-from-interference guarantees by exploiting vulnerabilities within "trusted" partitions, enabling return-oriented or jump-oriented programming attacks.

In a similar fashion, a stack overflow in a UDS handler can result in redirecting execution into the bootloader, which in turn can be used to perform an erase or flash write operation. Erase is especially useful when cryptographic keys are stored in normal flash, as deleting them may trick the software to treat the cryptographic key as one that has been set to a default value (all ones or all zeros).

Lack of software access control for the HPSE

Many MCU-based ECUs historically operated without a notion of least privilege. It is sometimes argued that hardening MCU software is futile, based on the assumption that once arbitrary code execution is achieved, all security guarantees are lost. This assumption can

easily be disputed by considering the security model of an MCU that supports high-value assets protected by the HPSE.

The HPSE is expected to have exclusive access to secure flash storage containing long-term cryptographic keys. The security model assumes that even if the main application software is compromised, these secrets should remain protected. In practice, this guarantee depends on both the robustness of the HPSE firmware and the shared memory protocol between the application software and the HPSE. Input validation flaws or memory corruption vulnerabilities within the HPSE firmware coupled with improper implementation of the shared memory communication protocol may allow an attacker to inject and execute their own code within the HPSE environment. Once that boundary is crossed, extracting keys becomes trivial, as the firmware itself has legitimate access to secure storage containing these keys.

The impact of such failures is amplified when keys are global or shared across a class of ECUs. Even when keys are device-unique, large-scale extraction across multiple units can enable reconstruction of common secrets usable against other vehicles. In this context, weak software-based access controls do not merely enable local compromise; they undermine the foundational assumptions of isolation that higher-level security mechanisms depend on.

Leftover debug code or backdoors

It is common for ECU firmware to include special-purpose code intended for development, testing, and manufacturing. When this code is not fully removed or disabled before production, it can introduce serious security weaknesses. In its mildest form, leftover debug functionality may expose verbose error logs that reveal memory contents or internal state at the point of failure. While useful to developers during bring-up and troubleshooting, such information can also provide attackers with precise insight into memory layout, control flow, or fault conditions.

In more severe cases, development artifacts manifest as explicit backdoor functionality. A common example is the presence of an undocumented memory read diagnostic service that was reserved for factory diagnostics resulting in the dumping of arbitrary ranges of the physical memory space. Other examples include maintenance routines that bypass ignition or vehicle-state checks when a specific CAN message is received, allowing diagnostic services to be enabled while the vehicle is in motion [12]. These behaviors are typically intended for assembly-line programming or service environments, but when left accessible in production firmware, they violate critical safety and security assumptions.

Moreover, manufacturing software used for factory setup and end-of-line testing is expected to be erased or permanently disabled before the ECU transitions into operational mode. When it is not, it provides attackers with powerful, privileged functionality that was never intended

to be exposed in the field. From a security perspective, leftover debug code and undocumented backdoors represent an attractive target during both the discovery and exploitation phases.

When working within an AUTOSAR-based system, it pays to look for components such as the Diagnostic Log and Trace module (DLT) which can further aid reverse engineering by providing structured runtime logging with well-defined message formats. These formats often include module identifiers, error codes, function names, or memory addresses. When DLT remains active in production firmware, the log strings can guide an attacker directly to specific BSW components that exhibit special faults that are interesting for exploitation. Even inactive DLT configurations often leave behind string tables, message templates, or lookup structures that help map internal functionality and reconstruct the ECU's control flow [13].

XCP and legacy protocols in production

Similar to debug code, the continued presence of legacy calibration and diagnostic protocols such as KWP2000, CCP, and XCP in modern MCU-based ECUs represents a systemic security weakness. They typically lack authentication and encryption, allowing an attacker to abuse legitimate calibration or diagnostic commands to perform unauthorized memory reads and writes. When left accessible, they provide entry points that can be used to extract firmware and modify execution flow.

While XCP capabilities are essential during development, they become a liability if they persist in production firmware. If XCP over CAN or XCP over Ethernet remains reachable in a deployed vehicle, an attacker with bus access can issue feature-discovery commands to determine whether calibration, programming, or data acquisition services are still active. Even when calibration protocol access is disabled, recovered firmware images may reveal that XCP handlers and programming routines remain compiled into the binary. From a reverse-engineering perspective, XCP functionality is often straightforward to identify. Command identifiers, dispatch tables, and grouped handlers for calibration, page management, DAQ, and programming are typically clustered in memory. Function pointer arrays are frequently used to map command values to handler routines. These indirection layers expand the number of paths into privileged functionality that can prove useful for attack chain construction.

Calibratable security

Many MCU-based ECUs expose calibration parameters that directly influence security behavior. These knobs are typically introduced to simplify engineering workflows during development, where flexibility and observability are prioritized. Such calibration values may govern features like enabling verbose error logging, relaxed diagnostic client authentication checks, secure boot bypass modes, or disabled network message authentication. When these parameters persist in production firmware, they become high-value targets for attackers seeking to weaken an ECU's security posture without defeating stronger cryptographic protections.

A recurring structural weakness is the use of signed calibration images that allow the downgrade of security features without being bound to a specific hardware instance. While such images are needed in development or in-field troubleshooting, not restricting them to a specific device means an attacker who gets hold of such a signed image can simply reflash it to their target device and immediately disable critical security features. Without a binding between the calibration artifacts and an immutable device-specific characteristic, the system implicitly trusts security configurations that were never intended to be portable across devices.

Calibration data also tends to receive less scrutiny than security-critical code. Once an attacker gains calibration write access, individual values can be modified to relax or disable protections by changing only a few configuration bytes, without touching executable code. It can be challenging to find the right calibration parameter that would allow a security downgrade attack. That is where having access to debug code on the target can prove valuable as it reduces the time to discover the right knob that disables specific security features.

Absent or weak cryptographic protection

As we saw in the *Hardware weaknesses* section, firmware image encryption in internal flash is not possible for MCUs. However, that does not mean that during software update events, the firmware images should be left in plaintext while in transit over a non-secure channel. Unfortunately, this remains one of the weaknesses of an active diagnostic flash programming session. Since the binaries are segmented into diagnostic frames over the CAN bus, an attacker who can sniff the bus during a download session can reconstruct the firmware in plaintext.

Moreover, when supporting software updates through OTA, one would expect the update image to be encrypted from the source to the destination ECU. However, weaknesses persist where decryption may occur at an earlier point such as in the Telematics Unit or the gateway ECU, leaving the images unencrypted and even unsigned once they have been processed by the vehicle interface element. This weakness creates an opportunity for both exfiltration and tampering of such images before they are installed on the target ECU.

Common cryptographic protection weaknesses include the following:

- **Plaintext diagnostic flashing**: Firmware transferred over CAN or another in-vehicle network may be captured and reconstructed if the update session is not encrypted
- **Early OTA decryption**: Update images may be decrypted by an intermediate ECU, leaving plaintext firmware exposed inside the vehicle network
- **Missing or incomplete authentication**: Firmware may be encrypted but not properly signed or authenticated, allowing attackers to modify images without detection
- **Weak protection after vehicle ingress**: Even if the external OTA channel is protected, the internal path from the telematics unit or gateway to the target ECU may expose the image to sniffing or tampering

When encryption and authentication are supported, poor key management can introduce a second class of serious weaknesses. Many ECUs store long-term secrets either in normal flash or, when available, in secure storage backed by an HPSE or similar hardware security module. If keys are stored in normal flash, they can often be recovered through firmware extraction. In one documented case, a bootloader used a hard-coded AES key embedded directly in the firmware binary to authenticate update images. Once attackers identified the key, they could craft malicious updates that passed authentication.

Key management weaknesses commonly include the following:

- **Hard-coded symmetric keys**: Keys embedded in firmware binaries may be recovered through reverse engineering and reused to authenticate malicious images.
- **Shared keys across ECU families**: A key extracted from one ECU may apply to other ECUs of the same model, platform, or supplier implementation.
- **Mutable trust anchors**: Even though the public key used for secure boot is not confidential, it must remain immutable. If attackers can reflash the sector containing the public key, they may be able to replace the trust anchor and boot their own firmware.
- **Insufficient hardware-backed protection**: Long-term secrets stored outside secure hardware may be exposed through firmware extraction, debug access, or memory disclosure.

MCU-based ECUs also suffer from common cryptographic design and implementation mistakes. Limited randomness can lead to predictable diagnostic challenges, repeated nonces can undermine otherwise sound protocols, and missing key-usage policies can expose secrets through oracle-style attacks. Weak constructions such as XOR-based encoding or proprietary ciphers are mistakenly treated as protective measures, creating a false sense of security. Public research has repeatedly shown that proprietary cryptography in automotive systems, particularly in immobilizers and key fob ECUs, can be reverse-engineered and broken [33].

When it comes to diagnostic access control, cryptographic implementation shortcuts further blur the boundary between protection and backdoor behavior. A common example is the reuse of fixed or default seed and key material for UDS SecurityAccess across an entire vehicle line. Once attackers extract the algorithm and seed/key pairs from firmware through reverse engineering, they can unlock diagnostic access on any ECU sharing that implementation. Even when a cryptographic method is used to enforce challenge-response mechanisms, relying on non-cryptographically secure random sources for the challenge value can enable attackers to coerce the ECU to repeat a challenge generation to reuse previously captured response values.

Note

The field of cryptographic implementation weaknesses is a rich domain. You are highly encouraged to consult the MITRE CWE website for a full list of cryptographic weaknesses that must be avoided. Additionally, NIST publishes several recommendation standards that can help avoid common pitfalls [14][15].

Now that we have studied a reasonable sample of hardware and security weaknesses that plague MCU-based ECUs, we will switch focus to building effective attack chains that will leverage some of these weaknesses to achieve the attack objective.

Penetration testing techniques and attack chain construction

Attacking an MCU-based ECU rarely consists of a single exploit. Instead, it is typically a chained process that moves from initial access to sustained control of the device. A successful attack chain often requires a combination of physical access, offline analysis, and network-based exploitation. A common pattern involves extracting firmware through physical means, identifying weaknesses through reverse engineering, patching firmware to test exploits, and demonstrating the exploit over the target through an attack that mimics real-life hacks. Depending on the scope of the penetration testing engagement, the exploitation work may terminate at a software component level, hardware component level, a target ECU, or even the full vehicle level.

A vehicle OEM, for example, may be interested to see how the discovered weaknesses in an ECU allowed real control of the vehicle. This may require demonstrating lateral movement to other ECUs, like the gateway, to target a specific vehicle function not directly reachable by the compromised ECU. An ECU supplier, on the other hand, may be interested in seeing exploitation violating their security goals around specific assets, such as the ability to execute arbitrary code and the defeat of existing ECU-level security controls. In the case of a chip supplier, the test scope is usually limited to a specific MCU, where the exploitation needs to demonstrate the defeat of certain secure hardware mechanisms, like bypassing a JTAG locking mechanism or extracting cryptographic keys from the HPSE.

Depending on whether the penetration testing type is a black, gray or white box level, certain sub-steps will be skipped. For example, in a white box penetration testing engagement, the ECU hardware will be supplied along with data sheets that detail the MCU type, PCB component list, and potentially a manifest of software components used or even the binaries

themselves. On the other end of the spectrum, a black box penetration test may start with nothing more than the sealed ECU enclosure.

In the following section, rather than break down the differences at each step based on the penetration test type, we will assume a black box penetration testing engagement where very limited information is provided which essentially mimics the scenario of a real-life hacker attempting to compromise the target.

Step 1: Physical reconnaissance and access

When the scope is the ECU level, the attack typically begins with hardware reconnaissance. The attacker identifies the target ECU, either in a vehicle or as a standalone unit obtained through secondary markets such as salvage yards. Physical inspection focuses on the printed circuit board, including MCU markings, external memory devices, connector pinouts, and exposed test points. The objective is to understand how the ECU is powered, how it communicates, and which interfaces may provide access to internal functionality.

Attackers map connector pins to power, ground, CAN or LIN buses, Ethernet switches, and potential programming or debug interfaces using a combination of multimeters, oscilloscopes, and logic analyzers. Publicly available datasheets are often used to identify JTAG, SWD, or proprietary debug ports, as well as special boot mode pins that alter startup behavior. Even before opening the ECU, partial reconnaissance can be performed through the OBD-II port by querying identifiers such as part numbers or software versions, which may indirectly reveal the MCU family or firmware lineage.

If a debug interface appears accessible, an attacker may cautiously test whether it is locked. If it is protected, fault injection techniques such as voltage, clock, or electromagnetic glitching may be evaluated as a bypass. The goal of this phase is information gathering. A successful outcome is a clear hardware blueprint that contains a list of all the essential hardware components and discovered hardware interfaces. At this stage, building special rigs is useful to further probe interesting interfaces like JTAG, UART, CAN, SPI, etc. in preparation for the next phase.

> **Note**
>
> When enumerating hardware components on a live system, care must be taken to avoid damaging the hardware by inducing shorts or mishandling test pads or the wiring harness.

Step 2: Firmware extraction or acquisition

With a hardware map established, the attacker's next objective is firmware acquisition. Access to firmware is a prerequisite for enabling offline analysis. Several extraction paths are commonly used. If a debug interface is unlocked, firmware can often be read directly from

memory. If it is locked, glitch attacks may be used to temporarily disable protections during read operations.

When firmware is stored in external memory, attackers can desolder the flash device or read it in-circuit using a suitable programmer. This approach requires knowledge of the memory device type and interface but avoids interacting with the MCU's security mechanisms altogether. Diagnostic interfaces present another avenue. Some ECUs expose memory read by address or firmware upload functionality through UDS services (service $35). Weak authentication schemes or poorly implemented seed-key algorithms can sometimes be abused to retrieve full firmware images.

Bootloaders, application code, or configuration regions each provide insight into execution flow, security assumptions, and potential weaknesses. Once extracted, even a partial firmware image represents a major advantage. It converts the problem from blind probing into a white-box analysis exercise. At this point, the attacker is one step away from gaining the equivalent of source-level visibility into the ECU's behavior.

Step 3: Reverse engineering and vulnerability discovery

Armed with the firmware image and perhaps calibration data, the attacker proceeds to reverse engineering. This is the step where the bulk of security analysis is performed to discover vulnerabilities. The binary is typically loaded into a disassembler or analysis framework configured for the correct CPU architecture such as Ghidra or IDA Pro. Initial analysis focuses on identifying software components, including diagnostic handlers, communication stacks, hidden backdoors, firmware update logic, and cryptographic routines. This is where knowledge of software anatomy will prove most useful. As we saw in the earlier section, if the software being used is AUTOSAR, then the attacker will have access to the open specifications from the AUTOSAR standard to identify critical functions that may be useful to exploit or probe for further analysis.

String references often provide early guidance into areas of interest, for example, finding the ascii characters `"AUTH_FAIL"` gives the attacker a hint that there exists an authentication routine which controls a security sensitive operation. Diagnostic messages, error codes, or mode indicators can also reveal the presence of security checks and state machines. Attackers search for known protocol identifiers, firmware download handlers, and memory copy routines that may be vulnerable to misuse. Similarly, software update mechanisms are examined closely to determine how new code is accepted and whether authenticity checks are enforced correctly.

One common technique is to search for lookup tables that contain callback function addresses for diagnostic routines. If the engagement scope includes the HPSE, then the attacker will look

for functions that interface with the HPSE and data sections for crypto job parameter exchange with the main processor.

Special attention is paid to the presence of hard-coded credentials such as default passwords, fixed seed key challenge pairs, or even plaintext cryptographic keys. The discovery of such credentials is a strong indicator that a more sophisticated exploit is possible when the abuse of such credentials is chained with other attack steps.

When the code is decompiled, it is possible to run static code analyzers to spot software security weaknesses. Using LLM-based tools, it is feasible to filter through these warnings to zero in on the true positive coding defects that may prove exploitable. Once a full map of the software functions and data sections has been adequately completed, the attacker will try to produce a control flow graph to evaluate the usability of any discovered memory corruption vulnerabilities. Code paths that process external input are scrutinized for improper bounds checking or unsafe buffer handling.

In some cases, if the ECU is fully functional under Hardware in the loop (HIL) setup, fuzzing can be used to complement static analysis by triggering crashes or anomalous behavior. This can be done at the network level through crafted CAN or diagnostic traffic or through the setup of fuzzing harnesses for software components. In either case, a crash usually indicates that the code contains a potential memory corruption vulnerability that can be abused. As you notice in this pattern, the process of reverse engineering is very much like searching for a needle in a haystack. Having the right tools and applying efficient techniques for filtering the signal from the noise is essential to keep the engagement on track and avoid chasing dead ends.

To accelerate reverse engineering and exploitation efforts, attackers usually need a way to install modified firmware on the target to test their findings and rule out dead ends as they attempt to perform deeper discovery. Finding ways to bypass flash programming access controls is critical whether done over diagnostics or by using a serial programmer over JTAG.

Once a method is discovered for reliable firmware patching, this phase turns into an iterative process, where hypotheses formed during analysis are tested on the target, refined, and re-tested until we have a sufficient set of exploitable weaknesses.

Note

Defensive security engineers frequently argue about the significance of physical attacks. During the reverse engineering phase, physical attacks are extremely useful for discovering vulnerabilities that can be chained for a truly remote or local network-based attack later.

By the end of the reverse engineering phase, we must have a prioritized set of exploitable weaknesses, each with a defined impact such as the ability to perform unauthenticated firmware updates or bypass secure boot. At this stage, the attacker can shift from exploration to deliberate exploit planning.

Note

For a useful workshop on reverse engineering, we recommend this instructional video: `https://www.youtube.com/watch?v=c5AwWOVHkEQ`

Step 4: Exploitation

Exploitation is meant to convert theoretical security weaknesses into successful attacks that align with the attack objectives. It is common here to chain multiple security weaknesses to carry out the proof of exploitation. For example, demonstrating that the vehicle bus is accessible remotely via the gateway ECU can be a first step in an attack chain targeting a deeply embedded ECU such as the brake control module. A second step may require triggering a memory corruption vulnerability through a specially crafted CAN or diagnostic command that overwrites control data and redirects execution.

Note

Achieving reliable code execution on MCU-based ECUs often means either injecting a shellcode payload into RAM or hijacking the control flow using existing code (Return Oriented Programming). For instance, the attacker might use a stack overflow to redirect execution into the bootloader's flash reprogramming function, effectively forcing the ECU to erase or rewrite parts of flash with code the attacker provides. Once execution is under the attacker's control, even temporarily, the ECU security model is effectively defeated.

During this phase, you may realize that several valid vulnerabilities which were discovered earlier are impractical to exploit within the engagement period. This can be due to other security controls that are hard to defeat. Still, reporting your findings will be valuable as a determined attacker may find ways to exploit them under different circumstances.

In the case of testing a specific component like an MCU, or a set of embedded software modules, demonstrating the presence of exploitable security weaknesses may be the final step in the engagement. For example, demonstrating that a weakness exists in the HPSE firmware of the MCU vendor that leads to the leakage of cryptographic keys through side channel timing

analysis from the main CPU side, would be considered a successful completion of the penetration testing engagement. Here we assume the security objective of the HPSE firmware vendor was to protect cryptographic keys from leakage through a compromised application running on the main CPU. Similarly, if the scope of the test were a specific flash bootloader, then demonstrating that it is possible to enter the flash programming mode and complete a successful flash programming session by circumventing authentication protocols would be considered a successful completion of the penetration testing engagement. Here we assume the security objective of the flash bootloader vendor is to prevent unauthorized access to flash programming functions.

In the case of testing a complete ECU or even a specific vehicle model, the penetration testing engagement will likely proceed using the steps listed below until the attack objectives have been realized at the ECU or vehicle levels respectively.

Step 5: Escalation and persistence

The next step after demonstrating the success of individual exploits is to attempt escalation of control and persistence. With MCU-based ECUs, escalation can simply mean the ability to run code without restrictions. If the exploitation occurs in an OS partition that does not have access to the CAN bus, then escalation may involve the ability to influence the other OS partitions that have network connectivity. This may involve the abuse of improper MPU configuration or the abuse of coding vulnerabilities in the operating system itself. However, in most cases, for MCU-based ECUs, escalation is achieved through the execution of one's own firmware or application software. For example, once the attacker can demonstrate the ability to reprogram software, they can simply download their own version of software that enables direct access to the network.

Escalation and persistence are not always synonymous. If the attack objective requires the compromise of a highly privileged entity like the HPSE, then the attacker will now need to demonstrate that a compromised application running on the main process can influence the HPSE to achieve the attack objective. This may involve injecting code into the HPSE or altering its control flow via the shared memory interface. Persistence on the other hand involves the demonstration of successful update of flash memory. Here it is important to differentiate between vulnerabilities that enable the update of ECUs in the possession of the attacker vs. vulnerabilities that enable the update of any ECU even if it is only remotely accessible. For instance, if a vulnerability exists in the ECU which allows an OTA package to be programmed without a valid signature, then that would certainly be considered a persistent attack. An attacker who can set up a fake OTA update server may be able to push invalidly signed images to the target ECU without the ability of detection. With persistence, it is safe to say that the ECU is now fully under continuous control of the attacker.

Step 6: Lateral movement across the vehicle network

When performing vehicle level penetration testing, lateral movement is frequently necessary to demonstrate the achievement of the attack objectives. Here, a compromised ECU is used as a steppingstone to target other ECUs. For example, an attacker who gains control of a network-connected ECU can inject traffic on shared network segments to impact other ECUs. This is where network discovery techniques (which we saw in chapter 6) come in handy. Lateral movement can also mean jumping off from the vehicle to the operational environment such as the EVSE or even the cloud backend.

With one ECU compromised, attackers often pivot to others. Where in-vehicle networks implicitly trust internal traffic, lateral movement can be straightforward. A compromised ECU can impersonate other nodes, open diagnostic sessions, or relay malicious messages across network segments.

Similarly, gateways intended to enforce segmentation turn into liabilities if compromised. Once bypassed, attackers can propagate attacks across other network buses such as powertrain, chassis, and body networks. This strategy was demonstrated in remote vehicle compromises where attackers entered through a telematics unit and then issued internal CAN commands to safety-critical ECUs. Lateral movement multiplies impact and complicates detection by causing coordinated misbehavior across multiple modules.

End-to-end attack perspective

A full attack chain may combine physical access, firmware extraction, logic flaws, and network exploitation. While not every attack requires this level of sophistication, layered defenses tend to push attackers toward chained techniques. By systematically progressing from reconnaissance to lateral movement, attackers peel back each layer of an MCU-based ECU's security model. With the discovery of exploitable security weaknesses, the penetration tester can then present their findings along with a proof of exploit that shows the steps involved and how they abided by the original security assumptions agreed upon with the ECU supplier or OEM.

Now that we have established how these attack chains are constructed, we can examine real-world cases where similar sequences were executed, grounding this framework in observed exploits.

Case studies of MCU-based ECU compromises

To solidify the concepts presented so far, we examine three case studies of publicly disclosed compromises involving MCU-based ECUs. Each case highlights different attack vectors, the attack steps, and outcomes. These are drawn from security research presented at venues like Black Hat to illustrate the progression of car hacking over the past decade, each mapped to the

MITRE ATT&CK framework for structured understanding. Each case study relies on more than one security weakness. You are encouraged to spot the security weaknesses using the knowledge from the prior section on MCU-based weaknesses.

Case study 1: Bypassing an ECU bootloader password via EM fault injection

This case study targets a powertrain control ECU from a 2019 Corvette which was publicized to be resilient to thieves and tuners [13]. An attacker who wanted to modify the powertrain ECU to gain more performance needed a way to modify the software and calibration images. The researcher demonstrated that using a precisely timed electromagnetic fault injection attack essentially defeated the Boot Assist Module (BAM) authentication method to allow tampering with the ECU software and calibration.

Note

Source: `https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Bam-The-Bam-Electromagnetic-Fault-Injection-And-Automotive-Systems.pdf`

Figure 7.9 shows the sequence of steps performed to carry out the attack objective.

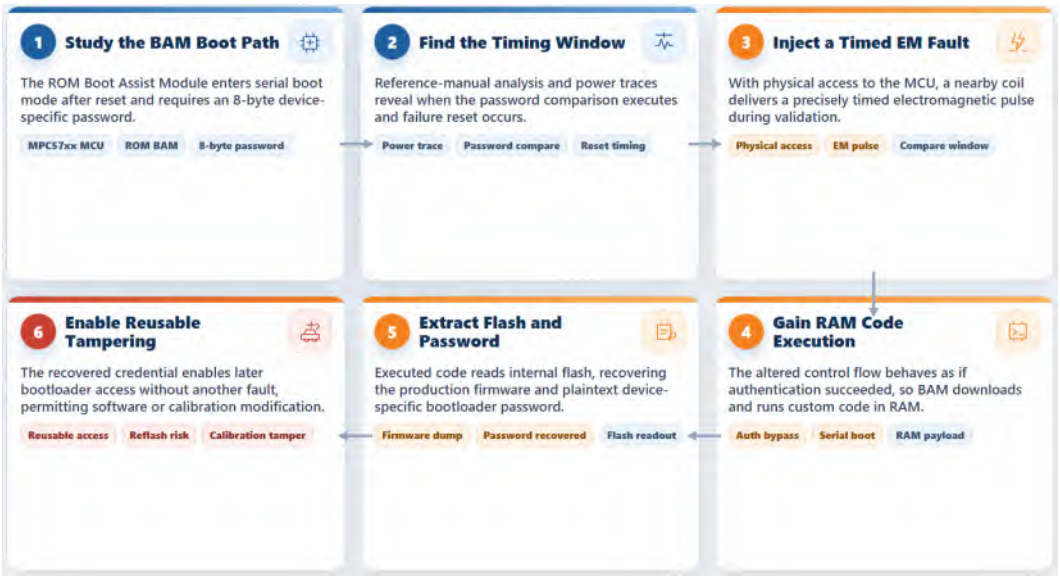


Figure 7.9: Sequence of steps for retrieving powertrain ECU firmware and serial bootloader password

Discovery and reconnaissance

The central target of this case study is the BAM, which is a first-stage bootloader that sets up the chip before handing control to the user code. The NXP/Freescale MPC57xx-series MCU implements the BAM in ROM, which executes immediately after reset. When a specific hardware pin is asserted, the Boot Assist Module enters a serial bootloader mode that accepts data over the serial interface, downloads code into RAM, and then executes that code. The particular ECU under evaluation relies on the MCU built-in bootloader password mechanism to prevent unauthorized firmware modification. The bootloader, in turn, requires an 8-byte password. In locked production ECUs, this password is device-specific and differs from the publicly documented default. If an incorrect password is supplied, the MCU resets after a short processing interval to thwart brute-force attempts.

The researcher began by studying publicly available NXP reference manuals to understand the boot sequence, the role of the Boot Assist Module, and the password comparison logical flow. They relied on power analysis to infer control-flow timing where the password comparison routine was being executed. This preparatory work was critical, as successful fault injection required sub-instruction-level timing accuracy.

Initial access

Using a ChipSHOUTER as the electromagnetic fault injection tool, the researcher placed a small copper coil near the MCU package. To see the diagram of the EMFI rig, please refer to [16]. Based on the power analysis from the discovery phase, they synchronized a high-energy electromagnetic pulse to the moment the bootloader compared the supplied password against the stored value. When timed correctly, the fault caused a control flow deviation that resulted in the password being accepted.

As a result, the bootloader proceeded as if a valid password had been provided. This granted the researcher immediate code execution within the bootloader context, allowing them to leverage BAM to download code into RAM which was subsequently executed.

From a tactic perspective, this step combines execution and privilege escalation achieved through hardware fault injection. While the password authentication mechanism was not cryptographically broken, the protection mechanism was effectively bypassed by manipulating the processor's execution flow.

Exploitation and persistence

After successfully bypassing the bootloader password check, the researcher gained the ability to download and execute custom code in RAM via the Boot Assist Module's serial bootloader. This execution context allowed read access to the MCU's internal flash memory, enabling extraction of the production firmware image from the ECU.

While the demonstration did not deploy persistent malware or modify flash contents, it showed that reflashing or permanent modification of firmware would be technically feasible once flash access is obtained. Thanks to the fact that the password was stored in internal flash, once the firmware was recovered, so was the device-specific password. With this password extracted, subsequent bootloader access no longer required fault injection. Legitimate authentication using the recovered credential was sufficient to enter serial boot mode. This transformed the attack from a repeated physical fault injection requirement into a reusable access method based on credential compromise, allowing the attacker full ability of software image tampering.

Impact

This case highlights a broader lesson for MCU-based systems: security mechanisms that depend solely on correct execution flow can be subverted when an attacker can induce hardware faults. Fault injection can invalidate foundational assumptions about control flow, allowing higher-level protections to be bypassed through physical means. Another lesson relates to how the password was stored. Using a device-unique password is a good practice but storing it in normal flash in plaintext makes extracting it a trivial task once code execution was established on the target device. As a result of these weaknesses, the claims of the ECU being immune to tampering and tuning attacks were proven false, allowing engine tuners to get more performance out of these systems than originally intended by the OEM.

Tip

A more recent talk that leverages the above attack to exploit the Boot Assist Module made some interesting discoveries about the initial findings of the Blackhat talk and is worth watching: `https://www.youtube.com/watch?v=URmI1VVilek&t=1798s`

Case study 2: Ways to exploit an insecure bootloader (DEF CON)

This security talk examines how automotive flash bootloaders accessed via UDS over CAN may be abused to modify the ECU firmware when an adversary hijacks or initiates an unauthorized programming session.

While theoretical in nature, the talk was based on surveying numerous flash bootloader implementations and hinted at real-life security weaknesses without mentioning specific OEMs or ECU suppliers that are impacted. An attacker who exploits the weaknesses mentioned in the talk can disable the normal application of the ECU by forcing it into programming mode, reflash unauthorized firmware, or intercept proprietary firmware images in transit. More critically, the bootloader itself may be replaced with a malicious variant,

sometimes referred to as a bootloader trojan. Once this occurs, the attacker gains durable control of the ECU that persists across power cycles and normal software updates. At that point, recovery often requires physical intervention or module replacement.

> **Note**
>
> **Source:**
>
> CAR HACKING VILLAGE - Flash Bootloaders Exposing Automotive ECU updates - TIB AV-Portal `https://av.tib.eu/media/39853`

Now, let's take a closer look at the weaknesses and their mapping to the attack phases.

Discovery and reconnaissance

The researcher described how one can observe diagnostic traffic relating to a flash programming session on the in-vehicle network. By monitoring the CAN traffic related to firmware transfer sequences and diagnostic message flows, it is possible to examine the order of operations, timing behavior, and authentication exchanges used during programming. Since firmware is transmitted without encryption, it is possible to capture the firmware image by sniffing the CAN bus during the flash programming session which would then enable firmware reconstruction.

The researcher shared scenarios in which the programming sequence could be intercepted or manipulated in real time. By positioning an intermediary between the diagnostic tester and the ECU (such as a CANalyzer tool), it is possible for diagnostic messages to be replayed, delayed, or altered during flashing.

Typical weaknesses to expect to find during the discovery phase are the following:

- Lack of authentication before entry into the programming session. Typically, ECU vendors are instructed to perform plausibility checks in the application before accepting a request to transition to programming mode based on vehicle speed.
- Lack of mandatory signature verification over firmware images before completing an update. In many cases integrity checks rely on checksums or cyclic redundancy checks (CRCs) which are easily calculated.
- Weak seed/key algorithms to unlock the critical flash programming diagnostic services

Initial access

Using a physical connection through the OBD-II port and a tool that supports the UDS protocol over CAN, initial access is established. This access allows direct communication with the ECU

to initiate a flash programming session. The study emphasized that such access is often assumed to be benign which is precisely why it becomes a practical entry point.

The researcher also analyzed architectures that support over-the-air updates. With MCU-based ECUs, the natural component to interface with the OTA client is the flash bootloader. Typically, an OTA client that runs in the application space handles communication with a vehicle gateway component such as the Telematics Unit, while the flash bootloader performs the actual software upload and download requests. In this context, the researcher described OTA as functionally equivalent to placing a diagnostic tester on a remote network. This highlighted the fact that weaknesses in the flash programming protocol and bootloader design are exposed to a larger attack surface beyond the physical boundary of the ECU thanks to the OTA clients, which can be used to establish an initial foothold.

Exploitation

By establishing a programming session with the ECU, an attacker can simply suspend the normal application. Naturally, entry into the programming session while the vehicle is moving would be considered a safety violation as the ECU will lose its normal functionality, which is effectively a denial-of-service attack.

Beyond denial of service, the talk further explained that portions of the diagnostic commands could be replayed due to predictable seed-key behavior with the implementation of UDS service $27 (SecurityAccessControl). The researcher also highlighted how traffic could be intercepted and modified during the firmware transfer, enabling injection or alteration of update payloads. When digital signatures are not used, the attacker only needs to update the firmware's integrity check values to ensure the modified firmware is accepted by the bootloader after the download is complete.

In addition to the ECU itself, the researcher highlighted weaknesses in the flashing toolchain. These programming tools often contain firmware images and, in some cases, seed-key material or proprietary algorithms. This basically means that reverse engineering flashing tools can lead to adverse effects as the lost credentials and extracted images could be abused on a larger scale.

Persistence

Clearly, being able to intercept a flash programming session can result in programming one's own firmware. In theory, the OEM can still issue a software update to restore the firmware to a valid version. However, the most severe outcome discussed in the talk is the ability to replace the bootloader itself. If the bootloader update mechanism does not require a valid digital signature over the bootloader image, then one can use the same flash programming sequence to replace the bootloader with a malicious version. By overwriting the boot component with attacker-controlled code, permanent persistence is achieved. This form of compromise

survives reprogramming of the main firmware and can selectively allow or block future updates.

Once persistence is established at the boot component level, detection becomes difficult. Standard diagnostic routines to read out the software versions may report spoofed values, while the malicious bootloader quietly complies with the attacker's intent. In many cases, remediation requires specialized recovery procedures or outright ECU replacement, with clear implications for cost, safety, and recall scope.

Impact

This theoretical study illustrates why firmware update paths in MCU-based ECUs demand the same rigor as primary runtime functions. Bootloaders are not passive utilities but rather powerful, privileged components that, when improperly secured, offer attackers a direct path to persistent system control. Moreover, the interface to OTA clients makes the flash bootloaders especially vulnerable to remote attacks.

It's noteworthy that in recent years OEMs have significantly improved the security of the flash bootloader specifications by requiring cryptographic signatures over software update images, and cryptographically secure challenge-response mechanisms to protect diagnostic services through service $27. Still, the ability to observe these exploits should be a reminder that weaknesses in implementing security controls at the bootloader level will remain critical and must be taken seriously.

Case study 3: My car, my keys: obtaining CAN bus SecOC signing keys

This case study documents how the researchers recovered SecOC cryptographic keys from a power steering ECU in a vehicle platform that began enforcing cryptographic authentication on steering-related CAN messages around the 2020 model year.

To achieve this, the researchers first extracted the firmware using a bypass of debug protections through fault injection. With firmware in hand, they reverse engineered both the application and the bootloader. This analysis revealed that the bootloader supported downloading a signed and encrypted payload into RAM and executing it through a routine call. By abusing this mechanism, the researchers were able to execute attacker-controlled code that reads sensitive key material from RAM. Once the keys were recovered, the researchers were able to generate valid authenticated steering messages accepted by the target ECU and noted that code execution could potentially be repurposed into a signing-oracle-style capability.

Figure 7.10 shows the sequence of steps performed to carry out the attack objective.



Figure 7.10: SecOC cryptographic key exfiltration steps

Note

Source: `https://www.youtube.com/watch?v=8958gH3KD3Y`

Discovery and reconnaissance

In the first phase of discovery, the researchers observed that newer model years of their target vehicles introduced SecOC protection on certain CAN messages, with a portion of each CAN frame dedicated to a truncated cryptographic message authentication code (MAC). Based on this observation, they selected a power steering ECU as the primary target, reasoning that it validated steer-by-wire messages and was likely implemented on an MCU with fewer hardening features.

To map the hardware components, the researchers acquired a standalone ECU, disassembled it, and identified the MCU as an RH850/P1M-E (R7F701381). Through public document search, they discovered a serial programming interface over UART which could be used for firmware extraction. The interface was locked via protection ID codes, but prior research [17] had shown how to bypass that through glitch attacks, so they applied fault injection against the MCU's internal core-voltage regulator domain. This involved removing a bypass capacitor and glitching both core rails using N-FETs. Using a low-cost setup built around a Raspberry Pi Pico and external hardware, they injected a precisely timed glitch during the debug protocol exchange. This caused the MCU to return a positive response that unlocked the serial programming interface which enabled them to dump firmware.

In the second phase of discovery, the researchers performed reverse engineering of the dumped firmware using Ghidra-based workflows. They analyzed the application and bootloader to understand code structure, diagnostic handlers, CAN parsing logic, the SecOC implementation, and key-handling behavior. They also inspected the contents of dataflash to look for hardcoded secrets.

Through this analysis, they confirmed that SecOC keys were loaded in RAM rather than relying on an HPSE for continuous key protection. They also verified that keys were not reusable across vehicles and were at least VIN-scoped. While this limited the fleet-wide portability of the attack, it would still enable full control for the targeted ECU. So, the attack objective was now clear: find a way to dump the contents of RAM to extract the SecOC keys.

In the third stage of discovery, the researchers searched for XCP and CCP handlers, as well as UDS services that may be useful in RAM content extraction. They found no XCP or CCP support, so they turned their attention to the UDS services of the flash bootloader. The main interesting finding there was the support for the download and execution of flash routines in RAM via service $31 (RoutineControl). So, the thought process was to create a code injection RAM blob that would be downloaded into RAM via the UDS service to carry out further exploitation.

But there were multiple obstacles that they needed to overcome first. This blob was protected by a CRC32 for integrity, an AES-CMAC for authenticity, and was encrypted using AES-CBC with a derived key. Calculating the CRC32 of a new blob is trivial but calculating a valid AES-CMAC and encrypting the blob requires knowledge of the cryptographic keys. Luckily, through firmware reverse engineering, the researchers were able to discover that the keys needed to calculate the CMAC of the RAM blob and perform the CBC encryption were derived from values that were hardcoded in the recovered firmware.

Now they had all the pieces needed to construct a malicious payload. They wrote a C payload that copied selected RAM regions and transferred them to the CAN controller, so the extracted RAM contents became observable on the CAN bus. A Python tool assembled the final blob by computing the CMAC, handling encryption, and wrapping the required hardcoded values.

Initial access

Initial access was achieved by leveraging standard UDS diagnostic functionality to transition the ECU into bootloader mode and authenticate via SecurityAccess (service $27) using a recovered symmetric AES key (through firmware reverse engineering). With valid authentication in place, the researcher gained access as a trusted diagnostic client and was permitted to invoke routine control and download routines that are normally intended for ECU reflashing and maintenance. This allowed the encrypted RAM blob to be transferred and executed without triggering error conditions.

Exploitation

Once running in RAM, exploitation was trivial. The injected payload directly accessed internal memory and exfiltrated cryptographic assets, including SecOC MAC keys, by copying their contents into CAN registers for observation on the bus. The payload then resets the ECU to resume normal operation, signaling the end of the exploitation phase.

Persistence

The researchers emphasized that their demonstrated objective was key extraction rather than installing persistent malware. However, they explicitly discussed post-exploitation implications. Once the relevant SecOC keys for a given ECU or VIN scope are recovered, the attacker gains a durable capability rooted in secrets rather than implants. With these keys, valid authenticated messages can be generated repeatedly, and in some configurations, the ECU could potentially be re-keyed using the recovered `Master_ECU_KEY`.

The researchers also noted that the same bootloader execution capability could enable persistence with additional effort, for example by hijacking the flash programming session rather than limiting the exploit to dumping RAM contents.

Impact

The primary impact demonstrated by this case study was network control of safety-critical steering functions. By recovering SecOC key material, they were able to generate valid cryptographically protected steering control messages accepted by the target ECU within its key scope. Throughout the analysis, the researchers highlighted several key weaknesses.

They demonstrated that serial programming interface access on the target MCU could be bypassed via glitching. They confirmed that critical SecOC key material resided in RAM and that, in at least two vehicle models, an AES key referred to as the SE key was stored in plaintext in flash. Knowledge of that key allowed them to create an authenticated RAM blob which the bootloader happily downloaded into RAM and executed under the assumption that it was the expected flash routines normally downloaded into RAM during the flash programming session. Finally, they observed gaps in SecOC anti-replay behavior, such as trip counter handling that did not prevent replay as expected.

Once keys are extracted, cryptographic protection no longer prevents message injection for that ECU. With the recovery of the flash routines, AES CMAC key, flash routines encryption key, and the Master ECU key, it was possible to download arbitrary code into RAM that was executed via the flash bootloader and to potentially perform SecOC key updates.

This case further demonstrates that even though secure on-board communication was enabled for that vehicle, defeating it was possible due to oversights in basic key management practices.

And, these three case studies gave us a taste of what real-life attacks look like against MCU-based ECUs.

Note

For more interesting case studies, please check out these resources:

- AUTOSAR reverse engineering: `https://www.youtube.com/watch?v=JjYzkVK9dks`
- Seed key remote attack on trucks: Blind Wireless Seed Key Exchange
- glitching RH850: `https://cc-sw.com/renesas-rh850-attacks-part-5-of-7-the-real-deal-glitching-a-locked-rh850/`
- Hacking ESP32 chips: `https://www.youtube.com/watch?v=c5AwWOVHkEQ`

If you are an offensive security engineer, you now have a reasonable perspective on the typical weaknesses you will encounter and the attack techniques you can apply against MCU-based ECUs. For defensive security engineers, those weaknesses serve as the low-hanging fruit to tackle. Additionally, there are some common security best practices that can help reduce the success of those attacks as we will see next.

Best practices for securing MCU-based ECUs

Defending microcontroller-based ECUs against the classes of attacks described in this chapter is both challenging and unavoidable. Automakers and suppliers must assume that motivated adversaries will target MCUs directly, including through physical access, and must therefore adopt a multi-layered security strategy which we frequently refer to as the defense in depth strategy. The goal is not to make attacks impossible, but rather to significantly raise their costs. The following mitigations represent established best practices for hardening MCU-based ECUs.

Locking down debug and development interfaces

All production ECUs should have debug and development interfaces such as JTAG and SWD permanently disabled or securely locked. This typically requires programming one-time-programmable memory or configuration bits that prevent debug and test mode access after the ECU enters operational mode. If debug access must be retained for legitimate servicing, it should be protected using cryptographically secure challenge-response mechanisms rather than fixed passwords.

Any residual backdoor mechanisms, such as GPIO strap combinations that force boot or test modes, should be eliminated or be protected by robust tamper-resistant measures. NHTSA guidance [18] explicitly recommends eliminating developer-level access in production systems unless absolutely necessary, and even then, only under strict control. Physical countermeasures such as removing test pads, potting debug pins, or applying epoxy can further increase the difficulty of abuse.

These controls are critical because real-world attacks have repeatedly demonstrated that physical access to an ECU can yield firmware extraction or execution control, either through direct debug access or via fault injection and side-channel techniques [29][32]. As a result, defenders must design with the assumption that ECUs will eventually fall into attacker hands and rely on robust protections rather than security through obscurity.

Implementing secure boot and firmware authentication

Secure boot is a foundational defense for maintaining ECU software integrity. On every boot, the MCU should verify the authenticity and integrity of its firmware images and calibration data using cryptographically secure digital signatures with the right key strength for the expected vehicle lifetime. This can be implemented in immutable ROM or a first-stage bootloader whose code flash has been locked, or via an HPSE that acts as the hardware-anchored root of trust. If verification fails, the ECU should refuse to transition control to the user application and enter a fail-safe state instead.

As we have seen in this chapter, fault injection attacks can result in the bypass of boot checks under certain conditions. Therefore, secure boot design must be hardened against common attacks such as glitching, TOCTU issues, root public key tampering, and downgrade attacks.

This can be done using anti-glitch defensive coding practices, static code analysis, and robust security testing, storing root public keys in HPSE-controlled secure storage, and protecting security calibration options with digital signatures that are enforced by the root of trust or an entity on the chain of trust.

Enforcing access control and memory protection

MCUs provide memory protection mechanisms that are often underutilized for security purposes. Designers should configure the MPU or MMU (if available) to compartmentalize software components based on trust and security criticality, not safety only. For example, CAN communication stacks should be isolated from control algorithms so that a buffer overflow in message parsing cannot result in code injection into safety-critical OS partitions.

When supported by the RTOS, privilege separation should be enforced so that externally exposed services such as diagnostics and network input handling run in lower-privilege modes. This limits the impact of exploitation and can turn a full compromise into a contained denial-of-service.

Additional software hardening measures such as stack canaries and compiler-based buffer overflow detection (e.g., StackGuard in GCC) should be enabled wherever feasible. While these measures introduce modest overhead, they significantly increase robustness against common memory-safety vulnerabilities.

When it comes to the HPSE firmware, it must be developed with a higher assurance target than normal software with special attention to hardening the shared memory communication protocol with the main CPU and the configuration of DMA controllers. HPSE firmware should enable DEP, stack protection, and strict separation from the main application core. Ideally, key material should never be directly visible to the HPSE CPU itself, and cryptographic operations should be handled exclusively by hardware engines that expose only opaque key handles.

Hardening diagnostic and update interfaces

As we have seen so far, diagnostic protocols such as UDS are powerful by design and therefore must be treated as high-risk attack surfaces. All sensitive diagnostic services, including ECU reset, programming mode entry, and memory access, should require strong authentication. Where UDS SecurityAccess is used, the underlying challenge-response algorithm must be resistant to brute force and reverse engineering, and keys should be unique per vehicle wherever possible. Where UDS service 0x29 is used, certificate validation must include full chain verification, including issuer validation and revocation status where supported.

Beyond validating authenticity, the system must explicitly bind authenticated identities to well-defined roles. Each role should be authorized to invoke only a minimal and explicitly enumerated set of diagnostic services. Replay and downgrade resistance are critical requirements as well. Authentication exchanges must incorporate fresh nonces to prevent reuse of previously captured messages.

Diagnostic functionality should also be context aware. Many ECUs already restrict programming operations based on vehicle state, such as ignition position or vehicle motion, and these restrictions should be consistently enforced. Rate limiting and lockout mechanisms may also be applied to prevent brute-force attacks against authentication services. Furthermore, firmware update mechanisms, whether dealer-based or OTA, must ensure both authenticity and confidentiality of the update images.

Employing robust cryptographic modules

Modern automotive MCUs increasingly include HPSEs designed to isolate cryptographic operations and key storage from normal application functions. These modules should be used to perform signature verification, authentication, and encryption without exposing plaintext keys to the main application processor. Even if the primary ECU software is compromised, an HPSE can prevent key extraction and misuse by restricting operations to well-defined cryptographic APIs.

HPSE firmware shall incorporate constant-time implementations to prevent side channel leakage. Similarly, OEMs shall plan for regular key rotation and avoid key reuse across large fleets. Cryptographic key leakage through fault injection should also be considered by minimizing secret-dependent control flow and leveraging hardware accelerators designed to resist side channel analysis. In fact, leaving long-lived plaintext keys in normal flash or RAM is no longer defensible.

Integrating fault injection countermeasures

As we have seen in this chapter, fault injection attacks are no longer theoretical. ECU designers must account for voltage, clock, electromagnetic, and other physical manipulation techniques. Chip suppliers should not leave the burden on the software alone to handle all such attacks. This is especially true for hardware interfaces like built-in serial programming, and JTAG where software has very little control to resist fault injection.

Many automotive MCUs provide built-in monitors for voltage, frequency, and temperature anomalies, which allows software to leverage these features, especially in the early boot stages. There, defensive coding techniques should be applied to reduce the success rate of glitch attacks by applying nested conditional checks and the usage of constants with large Hamming distance [19]. This reduces the likelihood that a single fault can flip an authorization outcome. Bootloaders especially should be designed so that bypassing security checks requires multiple precise faults rather than a single glitch.

Additional protections include tamper-detection sensors, coatings, or meshes that detect invasive attacks such as laser or EM injection. When tampering is detected, the HPSE may be instructed to wipe sensitive keys or enter a lockdown state. While such responses carry operational risk, they may be justified for high-impact domains such as gateways or theft prevention ECUs.

Summary

Microcontroller-based ECUs were once considered too constrained and too embedded to warrant serious attacker attention. As demonstrated throughout this chapter, that assumption no longer holds. MCU-based ECUs are both vulnerable and attractive targets, where successful attacks have already occurred in practice. These attacks span a wide spectrum, ranging from software exploitation over in-vehicle networks such as CAN to sophisticated fault injection techniques. The result is a threat landscape that includes local attackers with physical access, close-proximity attackers leveraging OBD-II or internal network access, and remote attackers who chain vulnerabilities to reach deeply embedded ECUs.

The potential impact of ECU compromise is substantial. Because these controllers directly govern safety-critical vehicle functions, a successful attack can result in loss of control,

degradation or disablement of safety mechanisms, or the trigger of unsafe behavior during operation. Even when safety is not immediately affected, attackers may extract sensitive data such as immobilizer key material, bypass emissions or tuning restrictions, or deliberately brick ECUs to cause financial and operational cost to the OEM. These realities underscore the necessity of a layered security approach. Hardware, software, and architectural defenses must work together so that failures are contained rather than cascade.

For offensive security engineers, the MCU domain offers a rich attack space that continues to be rewarding to study. Understanding the hardware, software, and architectural makeup of these systems is the best path for uncovering security weaknesses and building exploits. Luckily, techniques we learn for attacking MCU-based ECUs are largely portable when attempting to penetrate HPCs. This will be the topic of our next chapter.

References

[1] Infineon Technologies AG, *Infineon AURIX™ Microcontrollers: Intelligent Solutions for Automotive & Off Highway Applications*, Oct. 8, 2021. [Online]. Available: `https://www.infineon.com/assets/row/public/documents/10/43/infineon-aurix-automotive-presentations-en.pdf`

[2] AUTOSAR Specification of the Operating System. [Online]. Available: `https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_OS.pdf`

[3] MISRA Consortium, *MISRA C:2025: Guidelines for the Use of the C Language in Critical Systems*. Norwich, U.K.: MISRA Consortium Ltd., Mar. 2025. Available: `https://misra.org.uk/product/misra-c2025/`

[4] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security*, vol. 15, no. 1, pp. 1–34, Mar. 2012.

[5] SAE International, *J3101_202002: Hardware Protected Security for Ground Vehicles*, Feb. 2020. Available: `https://www.sae.org/standards/content/j3101_202002/`. Accessed: Apr. 5, 2026.

[6] Vector Informatik GmbH, *Flash Bootloader – ECU Programming via CAN, CAN FD, Ethernet, LIN and FlexRay*. Available: `https://www.vector.com/us/en/products/products-a-z/embedded-software/flash-bootloader/`

[7] Vector Informatik GmbH, *Flash Bootloader OEM Technical Reference: Vector UDS (SLP3) – Download Sequence Specification*, ver. 5.2, 2024. Available: `https://cdn.vector.com/cms/content/products/Flash_Bootloader/Docs/Vector_Flash_Bootloader_Technical_Reference.pdf`

[8] AUTOSAR Specification of Crypto Driver. Available: https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_CryptoDriver.pdf

[9] Z. Chen, S. L. Thomas, and F. D. Garcia, "MetaEmu: An architecture agnostic rehosting framework for automotive firmware," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, New York, NY, USA: Association for Computing Machinery, 2022, pp. 515–529. doi: 10.1145/3548606.3559338.

[10] Contag, Moritz, et al. "How they did it: An analysis of emission defeat devices in modern automobiles." 2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017.

[11] AUTOSAR Specification of Diagnostic Log and Trace, Available: https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_DiagnosticLogAndTrace.pdf

[12] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 447–462.

[13] E. Seo, H. E. Moon, and H. Yun, "Off-By-One 2025 Day 2," YouTube video, Off-By-One Comms Team. [Online]. Available: https://www.youtube.com/watch?v=LA7HNvqO1oc

[14] National Institute of Standards and Technology (NIST), Recommendation for Random Number Generation Using Deterministic Random Bit Generators (SP 800-90A). Gaithersburg, MD, USA: NIST. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-90a/final

[15] NIST, Recommendation for Key Management: Part 1—General (SP 800-57 Part 1). Gaithersburg, MD, USA: NIST. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final

[16] https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Bam-The-Bam-Electromagnetic-Fault-Injection-And-Automotive-Systems.pdf

[17] Renes'hack, Available: https://www.collshade.fr/articles/reneshack/rx_glitch_article.html

[18] National Highway Traffic Safety Administration (NHTSA), Cybersecurity Best Practices for the Safety of Modern Vehicles. Washington, DC, USA: U.S. Department of Transportation, Sep. 2022. [Online]. Available: https://www.nhtsa.gov/sites/nhtsa.gov/files/2022-09/cybersecurity-best-practices-safety-modern-vehicles-2022-pre-final-tag_0_0.pdf

[19] M. Witteman, "Secure application programming in the presence of side channel attacks," Riscure, White Paper, Aug. 2017. [Online]. Available: https://

riscureprodstorage.blob.core.windows.net/production/2017/08/
Riscure_Whitepaper_Side_Channel_Patterns.pdf

[20] P. Nasahl and N. Timmers, "Attacking AUTOSAR using software and hardware attacks," Riscure, White Paper, 2019. [Online]. Available: https://
riscureprodstorage.blob.core.windows.net/production/2019/12/
Riscure_Whitepaper_Attacking_AUTOSAR_using_Software_and_Hardware_Attacks.pdf

[21] P. Lapczynski, "Flash bootloaders: Exposing automotive ECU updates," presented at DEF CON 26 Car Hacking Village, Las Vegas, NV, USA, 2018. [Online]. Available: https://
av.tib.eu/media/39853

[22] Earls, "Arm's Cortex-R core tightens its grip on automotive-grade silicon," Electronic Design, Feb. 18, 2025. [Online]. Available: https://www.electronicdesign.com/markets/
automotive/article/55268816/electronic-design-arms-cortex-r-core-will-it-
dominate-in-automotive-grade-silicon

[23] Steurich, "Automotive ECUs: Architecture considerations to implement secure software updates over the air," Embedded Computing Design, Aug. 3, 2017. [Online]. Available:
https://embeddedcomputing.com/application/automotive/automotive-ecus-
architecture-considerations-to-implement-secure-software-updates-over-the-air

[24] J. Yoshida, "Infineon: Breaking down automotive attacks," EE Times, Jul. 15, 2013. [Online]. Available: https://www.eetimes.com/infineon-breaking-down-automotive-attacks

[25] Saß, Xhani Marvin, Richard Mitev, and Ahmad-Reza Sadeghi. "*Oops..! I Glitched It Again! How to {Multi-Glitch} the {Glitching-Protections} on {ARM}{TrustZone-M}*." 32nd USENIX Security Symposium (USENIX Security 23). 2023.

[26] Van den Herrewegen, Jan. Automotive firmware extraction and analysis techniques. Diss. University of Birmingham, 2021.

https://motordna.io/static/stickerlook/images/wp-automotive-security.pdf

[27] M. Dibaei, X. Zheng, K. Jiang, R. Abbas, S. Liu, Y. Zhang, Y. Xiang, and S. Yu, "Attacks and defences on intelligent connected vehicles: A survey," Digital Communications and Networks, vol. 6, no. 4, pp. 399–421, 2020. [Online]. Available: https://www.sciencedirect.com/
science/article/pii/S235286481930197X

[28] Cschirp. (2010). *Motorsteuerung VW Golf TDI innen* [Photograph]. Wikimedia Commons. Licensed under Creative Commons Attribution-Share Alike 3.0 Germany. https://
commons.wikimedia.org/wiki/File:Motorsteuerung_VW_Golf_TDI_innen.jpg

[29] Obermaier, Johannes, and Stefan Tatschner. "Shedding too much light on a microcontroller's firmware protection." *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. 2017.

[30] Wiersma, Nils, and Ramiro Pareja. "Safety!= security: On the resilience of asil-d certified microcontrollers against fault injection attacks." *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2017.

[31] O'Flynn, Colin and Chen, Zhizhang. (2014). ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. IACR Cryptology EPrint Archive. 8622. 10.1007/978-3-319-10175-0_17.

[32] Wouters, Lennert, et al. "Dismantling DST80-based immobiliser systems." *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.2 (2020): 99-127.

[33] Verdult, Roel, Flavio D. Garcia, and Baris Ege. "Dismantling megamos crypto: Wirelessly lockpicking a vehicle immobilizer." *Supplement to the Proceedings of 22nd USENIX Security Symposium (Supplement to USENIX Security 15)*. 2015.

[34] O'Flynn, Colin. "*Low-Level Automotive ECU Security: Vehicle Vulnerabilities*." Circuit Cellar, Available: `https://circuitcellar.com/research-design-hub/design-solutions/low-level-automotive-ecu-security/`.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

8

Attacking the High-Performance ECUs

Modern vehicles are increasingly built around a few centralized **High-Performance Computers (HPCs)** that consolidate functions of many of the MCU-based ECUs (Microcontroller Unit-based Electronic Control Unit) we studied in the previous chapter. Unlike those ECUs that rely on a constrained RTOS with a single tenant, HPCs leverage high performance **System-on-Chip** (**SoC**) devices. These SoCs contain heterogeneous CPU cores that may be partitioned across multiple software domains or tenants. Within the same chip, some cores may run rich operating systems such as Linux, QNX, or Android, while others run RTOS-based software for time-critical control functions. Their diverse code base, expansive hardware interfaces, and their positioning within the vehicle E/E architecture produce a much larger attack surface when compared to traditional ECUs, making them a more attractive attack target. Typically, HPCs host multiple vehicle functions such as infotainment, instrument cluster, ADAS, and connectivity domains, leveraging virtualization for isolation. Their complexity, combined with the dynamic nature of the Software Defined Vehicles (SDVs) in which they are deployed, continues to make security robustness difficult to achieve.

In this chapter, we will dive into the offensive security domain of HPCs building upon what we have learned in *Chapter 7*. As we did before, we'll begin with an overview of the HPC hardware architecture and common software stacks. We then examine their common weaknesses both in hardware and software. Then, we outline methodologies for building attack chains against these high-performance devices, from initial recon, and firmware extraction up to exploitation and persistence. Several case studies are then shared to illustrate real-world compromises of HPCs and their impact. Finally, we will also discuss security best practices and defenses to thwart the most common attacks while emphasizing the need for a layered approach to

security. By the end of this chapter, you will gain insights into the techniques by which HPCs are compromised and how to make those attacks less effective.

In the rest of this chapter, we will cover the following main topics:

- Introducing high-performance ECUs
- Understanding the hardware and software anatomy
- Identifying common weaknesses in HPCs
- Exploring penetration testing techniques
- Case studies of publicly disclosed compromises involving HPCs
- Best practices for securing HPCs

Introducing high-performance ECUs

The term high performance ECU came into existence with the emergence of the SDV and the trend of transitioning from the highly distributed E/E architecture to the zonal and centralized one. This resulted in a shift of functionality from the traditional MCU-based design to SoC-based systems. Fueled by increased demand for high computation power for autonomous driving and cockpit systems, which require expanded networking interfaces and advanced connectivity applications, HPCs gained significant penetration into the automotive market and continue to do so. What started as limited use for Telematics and Infotainment gradually expanded into the central gateway, ADAS and most recently the vehicle central computer.

High performance ECUs are differentiated from MCU-based ECUs by their rich operating systems and PC-like networking interfaces. Unlike MCU-based systems where real-time performance is highly critical, HPCs can tolerate variations in real-time performance. This is not to say that determinism is not important for HPCs, but it certainly is not as stringent as the hard real-time performance of systems like engine controllers or airbag ECUs.

Moreover, HPCs are increasingly consolidating multiple vehicle domains into a single hardware-based design such as infotainment, instrument cluster, external connectivity, and even advanced driving assistance. This consolidation yields immense complexity, requiring robust isolation to ensure a compromise in one domain such as infotainment cannot easily propagate to safety-critical controls. To achieve that, HPC designs typically employ Type-1 (bare-metal) hypervisors to run multiple guest OS instances side-by-side on the same SoC. These hypervisors ensure strong isolation between mixed-criticality partitions, for example, an ASIL-B instrument cluster alongside an Android VM that handles IVI functions.

A Type-1 hypervisor (QNX Hypervisor, Xen, Green Hills Integrity Multivisor, etc.) provides each VM with virtual CPUs, memory partitions, and virtual devices. It can also mediate inter-VM communication and hardware access to prevent interference. BlackBerry's QNX Hypervisor, for

instance, supports VirtIO paravirtualized devices for network and storage sharing between VMs [3], and dedicates real hardware (GPU, CAN controller, etc.) to VMs as needed. This type of software architecture is illustrated in *Figure 8.2*, which we will examine in more detail in the *Software architecture of HPCs* section later in the chapter.

Since most HPCs are running Linux, or Android, penetration testers who are used to evaluating enterprise systems and mobile devices can carry over their skillset relatively easily. But before we can learn those penetration testing techniques, we must take a deeper look at the hardware and software anatomy of HPCs which we will do next.

Understanding the hardware and software anatomy

To gain a better perspective about the attack surface of an HPC, we must first familiarize ourselves with the typical hardware and software architecture of such systems.

Hardware components of HPCs

When examining the PCB board of an HPC, the components that usually stick out are the advanced SoCs, diverse memory and storage technologies, and a broad array of connectivity, sensor interfaces, and debugging ports, as illustrated in *Figure 8.1*.

Automotive SoC and companion MCU

HPCs are built around high-performance **System-on-Chip** (**SoC**) devices that often incorporate advanced CPU clusters. For example, you may find high-performance cores such as ARM A78 alongside embedded cores based on RISC-V or ARM Cortex-R/M architectures. In tandem with the SoC, it is common to pair it with a companion **Microcontroller Unit** (**MCU**) that serves as a safety monitor. This MCU (e.g., Infineon AURIX, Renesas RH850) is dedicated to power management, fail-safe operations, and can act as a vehicle network interface, thus ensuring that the overall system can continue to operate even if the SoC component fails.

Memory and external storage

Unlike MCU-based systems where flash and RAM are embedded within the chip, the HPC relies on external volatile and non-volatile memory. HPCs typically include high-speed DDR4/DDR5 DRAM, with capacities of several gigabytes shared across different compute environments within SoC. For persistent storage, HPCs use a range of flash technologies including traditional NAND, eMMC/UFS flash, and even Non-Volatile Memory Express (NVMe) to support large video data recording, for example. While NOR flash typically stores bootloaders and RTOS firmware, eMMC memory modules are typically used for larger OS images and filesystems.

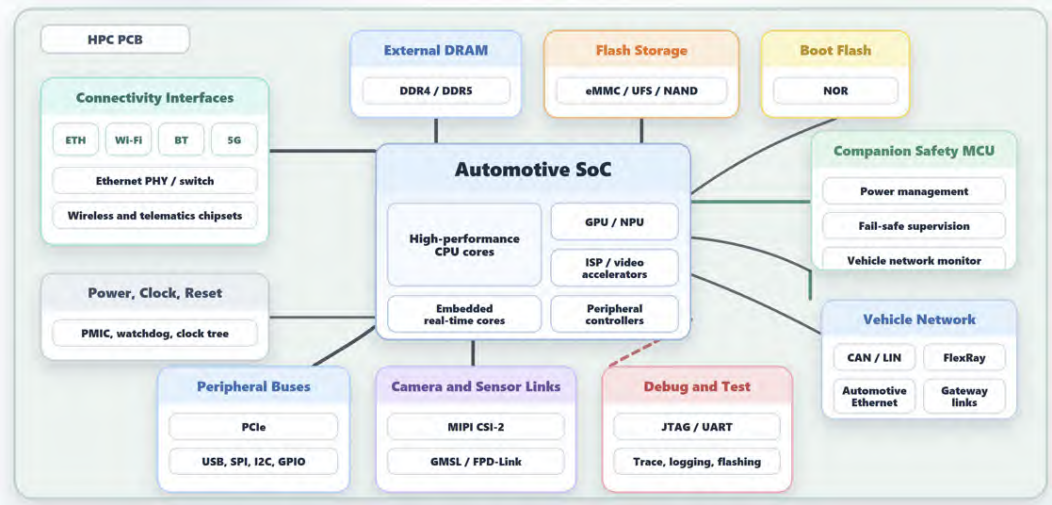


Figure 8.1: Simplified block diagram of the HPC hardware

Connectivity interfaces and peripheral buses

In addition to multi-Gigabit Ethernet controllers and integrated automotive Ethernet switches, modern HPCs feature advanced connectivity options such as Bluetooth/Wi-Fi chipsets and in some cases even cellular modems. These enable communications for telematics, OTA updates, and V2X services. As we saw in *Chapter 5*, finding a vulnerability in any communication interface either due to improper configuration, protocol weakness or a defect in the software drivers will serve as the first entry point to deliver malicious code to the target device. Since the HPC is typically the first ECU exposed to these vulnerabilities, it is naturally the primary target for such attacks.

On the other hand, peripheral buses like PCIe allow the linking of multiple SoCs in multi-chip configurations as well as interfacing with NVMe devices or even companion MCUs. Other interfaces such as USB, CAN, and LIN offer connectivity to user devices and in-vehicle network protocols. Moreover, debug and test interfaces, such as UART and JTAG, are often present for development and maintenance, making them a common target for reconnaissance, firmware extraction, and reverse engineering.

Camera and sensor interfaces

For combined cockpit and ADAS applications, HPCs include interfaces like **Mobile Industry Processor Interface (MIPI)** CSI-2 for cameras, along with serial interfaces such as **Gigabit Multimedia Serial Link (GMSL)** or **Flat Panel Display (FPD)**-Link. These high-bandwidth data streams are processed by integrated ISPs or dedicated capture units, often assigned to an

RTOS domain within the HPC to meet real-time requirements. Now that we have a brief overview of the hardware components of an HPC, let's turn our attention to the software part.

Software architecture of HPCs

The software stack in an HPC is layered and diverse, mirroring the multiple runtime execution environments that co-exist in a single computing platform while adhering to stringent safety and security requirements.

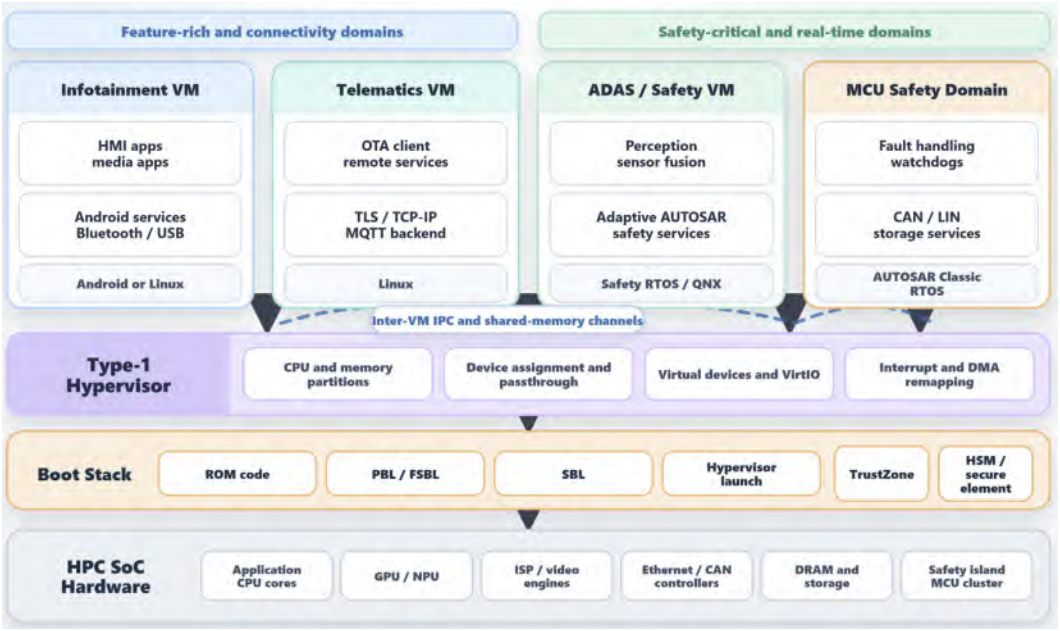


Figure 8.2: Typical software block diagram of an HPC

Next, we explore the various components of the software stack of an HPC.

Bootloaders and hypervisors

Upon power-on, ROM code performs the initial chip setup and launches the first-stage bootloader, which may boot one or more real-time cores that manage various hardware engines before reaching the stage where the hypervisor is loaded. The hypervisor sets up memory partitions, assigns CPU cores and hardware devices to VMs, and then sequentially boots each **virtual machine** (**VM**) while retaining the higher privilege level. Automotive systems often use Type-1 hypervisors such as QNX Hypervisor, Xen, Green Hills INTEGRITY Multivisor, or Wind River Helix. The hypervisor also provides a minimal monitor layer to handle inter-VM communication and manage device access via techniques like para-virtualization.

Guest operating systems and software partitioning

HPCs typically host multiple guest operating systems that are separated into software partitions while being aligned with functional, safety, and security requirements. Each guest OS executes inside its own virtual machine and is assigned a dedicated set of CPU cores, memory regions, and access to hardware devices by the hypervisor. This partition is typically enforced using hardware virtualization support (stage-2 address translation, device assignment, and interrupt remapping) and forms the primary boundary between safety-critical and non-safety-critical workloads.

Safety-critical domains

Functions requiring high automotive safety integrity are commonly placed into dedicated real-time or safety-certified guest environments. These domains typically execute a safety-certified operating system such as QNX and are configured with strict temporal and spatial isolation. For example, an ADAS or perception fusion stack may run inside a safety VM with dedicated CPU cores, bounded execution time, and tightly restricted access to peripherals. In addition, many platforms include a safety island or lockstep microcontroller cluster inside the SoC that executes independently from the main application cores and supervises safety-related behavior such as fault detection, watchdog handling, and fail-safe transitions.

Feature-rich and connectivity domains

Non-safety domains typically host operating systems such as Linux or Android and support infotainment, telematics, connectivity, user interfaces, and backend communication. A telematics VM may manage cellular connectivity, over-the-air updates, and backend messaging, while an infotainment VM may host Android applications, multimedia pipelines, Bluetooth services, and user-facing frameworks. In some designs, additional Linux-based partitions are dedicated to camera processing, surround-view composition, or instrument-cluster rendering. These domains are expected to be separated from safety-critical workloads because they usually expose a much larger externally reachable attack surface due to wireless interfaces, media parsers, and complex application frameworks.

Mixed-criticality execution model

This multi-VM design results in a mixed-criticality execution environment in which domains with fundamentally different threat models and certification assumptions coexist on the same silicon. Feature-rich domains typically evolve faster, receive more frequent updates, and depend on large open-source software stacks. Safety-critical domains, by contrast, change slowly and are validated under strict safety processes. From an attacker's perspective, this asymmetry is central to attack-chain construction. Initial access is far more likely to be achieved in a high-complexity domain such as infotainment or telematics, with subsequent steps attempting to cross isolation boundaries into more trusted or safety-relevant partitions.

As a result, guest operating system partitioning in HPCs should not be viewed as a simple virtualization deployment, but as a layered isolation strategy that combines hardware enforcement, hypervisor policy, device assignment, and carefully constrained service interfaces. The security of the overall platform depends not only on the strength of the hypervisor, but also on the correctness and minimality of the partitioning model that governs how guest operating systems interact with hardware and with each other. Minimality simply means that each guest is assigned only the memory, peripherals, shared resources, and communication paths required for its intended function. This principle is discussed further in the *Secure partitioning in hypervisor-based HPC architectures* section.

MCU software domains

As we mentioned in *Chapter 7*, the MCU software architecture can fit entirely inside an HPC to handle functions such as safety error handling in a functional safety island to achieve higher ASIL targets. Therefore, it is not uncommon to find one or more AUTOSAR classic instances running within an HPC. There are, however, some differences in how that software is booted and operated from a pure MCU-based ECU.

For one, MCU software in an HPC is booted from external storage into DRAM or SRAM that is partitioned for that MCU execution environment. The bootloader stack that is responsible for booting the entire HPC will handle booting the MCU software as well. Rather than fetching instructions from internal flash, the MCU software will fetch instructions from its dedicated RAM partitions. When it comes to handling persistent storage, instead of relying on embedded dataflash, the MCU software must rely on storage services from the HPC to read and write data normally to a robust storage medium like NOR flash. Besides that, the MCU domain will communicate with the rest of the vehicle in a similar fashion as a normal MCU by utilizing CAN, LIN and Ethernet. It will also support an IPC protocol for communicating with the application domain within the HPC itself for example, to issue commands in case of a fault.

Inter-process and inter-VM communication

With multiple operating systems and processors sharing hardware resources, robust inter-process communication (IPC) mechanisms become essential. Hypervisors typically implement virtual communication channels, using methods such as shared memory buffers to facilitate secure data exchange between VMs. Additional driver software within the VM handles logical channel setup and service authentication to produce an end-to-end communication solution. However, if these channels are not carefully designed and isolated, they can become attack conduits, enabling lateral movement across functional domains (e.g. infotainment to ADAS).

Network communication protocols

HPCs feature a rich set of networking protocols that aim to address several communication use cases. For in-vehicle communication, we typically find SOME/IP, which enables

communication with signal-based ECUs over Ethernet. Of course, the usual CAN drivers and FlexRay drivers, along with their respective signal-based communication layers, are also used. For external connectivity with the backend or with other HPCs like the In-Vehicle-Infotainment (IVI), we typically find TLS with TCP/IP software stacks. If MACsec is used, we expect to find WP-supplicants and, of course, Ethernet drivers. Other communication software we encounter includes MQTT clients for communicating with the backend and possibly DDS.

Device access, passthrough, and debug tools

HPC software often uses device passthrough for performance-critical peripherals, such as granting a Linux VM direct access to an Ethernet controller or allowing the Android VM exclusive control of a USB interface. While this minimizes overhead, it also means vulnerabilities in guest drivers can facilitate exploitation of the underlying hardware without much intervention from an additional monitor like the hypervisor. Moreover, HPCs frequently expose interfaces for flashing, debugging, profiling, and logging (such as SSH, Telnet, GDB, verbose logging). Although invaluable during development, if these tools or interfaces are left enabled in production, they become useful attack vectors to perform reconnaissance and establish initial access.

Special hardware engine firmware

HPCs manage a range of special hardware that is needed for their typical applications like image signal processors (ISPs), video decoders/encoders, deep learning inference engines, power/temperature management, and embedded GPUs. Normally, the hardware engine will be paired with an embedded core like RISC-V that will handle setup of the hardware and orchestrate job execution. These peripherals, along with their specialized firmware, expand the attack surface in ways that are normally overlooked, which we will see later in this chapter.

Now that we have surveyed the typical hardware and software components of HPC, we will turn our attention to the common weaknesses that apply to these systems.

Identifying common weaknesses in HPCs

To list all the security weaknesses that an HPC may exhibit is not feasible in a single chapter. Therefore, to motivate the topic, we will instead highlight some of the most common security weaknesses that a penetration tester should consider when evaluating an HPC.

Naturally, some weaknesses will overlap with MCU-based ones. When they do, we will avoid repeating the basic common aspects that were mentioned in the previous chapter and focus on unique aspects only.

Hardware weaknesses

As we will see throughout this section, some hardware security weaknesses are useful during the reconnaissance phase as they help in exfiltrating data and establishing initial access, while others are useful when constructing attack chains during active exploitation. You are encouraged to think along these two categories when evaluating the usefulness of a discovered security weakness.

Insecure debug and peripheral access

As discussed in the previous chapter, during reconnaissance attackers may exploit exposed hardware features to extract firmware or cryptographic secrets without having fully compromised the system. One fundamental weakness in this phase is the presence of debug and test interfaces such as JTAG or vendor-specific debugging ports. These interfaces, intended for development and manufacturing, provide low-level access to the controller's internals.

In automotive HPCs, ECU suppliers are expected to disable or restrict debug interfaces in production devices, commonly by blowing fuse bits or programming **one-time programmable** (**OTP**) memory. However, fuse- and OTP-based protections are not infallible. Prior research has demonstrated multiple classes of attacks against such mechanisms, including voltage or clock glitching to bypass fuse checks and semi-invasive attacks against anti-fuse structures [2]. As a result, the presence of debug interfaces, even if nominally disabled, must still be considered a potential attack surface for determined adversaries.

Other weaknesses arise from lifecycle-management and debug-protection errors: As platforms move from manufacturing to production, debug capabilities are expected to become more restrictive. However, if those restrictions are not correctly applied, privileged debug features may remain accessible or unlockable, significantly reducing the effort required for firmware extraction or system introspection [42].

Unprotected USB ports represent another attractive reconnaissance vector. While USB interfaces are often required during manufacturing or servicing to install firmware, leaving them enabled or insufficiently restricted during normal operation constitutes a security weakness. When coupled with misconfigured DMA channels, a malicious USB device may be able to perform unauthorized memory accesses or code injection, providing a foothold for further information extraction or system compromise [4]. In systems where USB is required during normal operation, such as IVI platforms, weak isolation of USB DMA transactions can similarly be abused as an attack primitive.

Another reconnaissance-aiding weakness lies in how HPCs handle external storage and memory interfaces. Due to their reliance on off-chip non-volatile storage and RAM, an attacker with physical possession of the HPC can directly probe SPI flash or eMMC devices, or tap DDR memory buses using interposers, to capture firmware images or data in transit. If code and

secrets are transferred over these buses in plaintext, an adversary intercepting them faces little resistance in exfiltrating sensitive material [5][3]. In practice, unless off-chip memory contents and interface buses are encrypted and authenticated, attackers may desolder memory devices or snoop on memory buses to obtain large volumes of firmware and data.

Weak hardware isolation

Hardware weaknesses that break software isolation assumptions are at the top of the list of what attackers try to exploit when targeting HPCs.

One common weakness arises from coarse-grained MMU configurations or overly permissive memory mappings: While modern HPCs typically rely on MMUs (multiple-stage address translation under virtualization) to isolate virtual machines and processes within a VM, configuration errors can easily result in access scopes that exceed what a given workload requires. Examples include mapping large contiguous physical memory regions instead of narrowly scoped buffers, reusing generic page tables across domains, or enabling shared memory regions for convenience during development that are left in production. Such misconfigurations expand the attack surface by granting virtual machines or user-space components visibility into memory regions belonging to higher-privilege or higher-criticality domains [6][7].

In addition to memory isolation, many automotive HPCs implement hardware firewall mechanisms or register access filters to protect memory-mapped registers and critical control interfaces. These filters are intended to restrict which bus masters, privilege levels, or execution environments can access specific register ranges, for example, those controlling clocking, reset logic, safety mechanisms, or peripheral configuration. When correctly configured, such register filters form a crucial last line of defense against accidental or malicious reconfiguration of critical hardware.

In practice, applying least-privilege access control consistently across these mechanisms is difficult at scale. Automotive HPCs may expose thousands of memory-mapped registers across CPUs, GPUs, accelerators, and peripheral controllers, each with distinct access requirements across boot stages, operating systems, hypervisors, and safety monitors. Coordinating those permissions across multiple software teams and lifecycle stages introduces significant challenges that frequently result in mistakes. If least-privilege principles are not rigorously enforced, register filters and access-control policies can remain broader than intended and weaken isolation boundaries.

Once such weaknesses exist, they become highly valuable post-compromise primitives. An attacker who gains code execution in a lower-criticality virtual machine or user-space environment may leverage overly broad MMU mappings or misconfigured register access controls to escalate their access. This can include modifying peripheral configuration registers,

disabling safety checks, reprogramming DMA engines, or interfering with isolation mechanisms themselves. Importantly, these attacks do not require exploiting software vulnerabilities in isolation code; they simply abuse legitimate hardware access that was incorrectly configured.

From a system design perspective, weak hardware isolation erodes the assumptions underpinning virtualization, mixed-ASIL deployments, and defense-in-depth strategies. Even when hypervisors and operating systems are correctly implemented, misconfigured MMUs or hardware firewalls can collapse isolation boundaries at the silicon level. For this reason, isolation failures caused by coarse-grained memory mappings or permissive register access should be treated as architectural security risks rather than implementation bugs, particularly in safety-relevant automotive HPC platforms.

GPU weaknesses

GPUs are frequently integrated into HPCs to accelerate perception, visualization, and AI workloads typically across mixed-criticality domains. GPUs are designed to execute highly parallel workloads using programmable shader or compute kernels, with execution largely orchestrated by a complex driver and runtime stack, unlike CPUs which directly execute general-purpose programs with fine-grained control flow.

That programmability makes GPU execution control fundamentally different from the conventional CPU model of page-based execute permissions. Prior research has shown that user-space processes can abuse GPU command submission paths to download and execute malicious GPU workloads [8][9].

In the absence of hardware-enforced execute protection, any memory region from which the GPU fetches instructions becomes executable by definition. As a result, if an attacker can corrupt or influence GPU memory, either directly or indirectly through user-space APIs, they may be able to inject malicious GPU kernels that are subsequently executed by the GPU without violating any architectural protection rules.

This risk is compounded by the fact that GPU memory is often shared across multiple execution contexts and is accessed through complex driver stacks. Weak isolation between GPU contexts, overly permissive buffer sharing mechanisms, or vulnerabilities in user-space GPU APIs can enable unintended writes into executable GPU memory regions. Prior research has demonstrated that user-space processes can abuse GPU command submission interfaces to upload and execute arbitrary GPU code, even in environments where CPU-side execution controls are intact [8][9]. In such cases, the lack of an NX-like protection on GPU memory directly enables code execution rather than merely data corruption.

From a system security perspective, malicious GPU kernel execution is particularly concerning because GPUs often operate with elevated privileges relative to system memory and

peripherals. GPUs may have broad DMA access to system RAM, access to shared memory regions with the CPU, or visibility into framebuffers and sensor data streams. A malicious GPU kernel can therefore act as a stealthy execution context that is largely invisible to traditional CPU-based security monitoring tools.

Academic work has shown that GPU-based malware can persist undetected, perform covert memory access, and even assist in extracting secrets from CPU memory via DMA or side-channel techniques [8]. If GPU memory is accessible from non-safety-critical user space and lacks enforceable execute protections, a compromised non-critical workload could potentially inject GPU code that manipulates data belonging to safety-relevant functions. This undermines assumptions about partitioning between ASIL and non-ASIL domains, particularly in architectures where the GPU is treated as a shared accelerator rather than a strictly sandboxed peripheral. As a result, GPU non-execute weaknesses should be treated as a structural risk in systems that rely on GPUs for critical workloads, especially when combined with weak driver isolation, permissive DMA access, or shared memory architectures.

Side-channel leakage

Like any computer system, HPCs are susceptible to a range of side-channel attacks that aid adversaries in reconnaissance by extracting secrets through indirect means. These attacks exploit unintentional emissions of the system such as timing, power consumption, and electromagnetic radiation. One classical example which we saw in *Chapter 7* is Differential Power Analysis (DPA). By measuring the HPC's power consumption while it performs cryptographic operations, an attacker can statistically infer the cryptographic keys being used. This technique has been shown to extract AES encryption keys with relatively modest measurement effort under laboratory conditions. By capturing power traces and applying statistical analysis, secret keys can be deduced incrementally, as demonstrated by numerous academic attacks on real devices [10][11].

The risk becomes more severe when cryptographic keys are shared globally across all HPC instances. An attacker who can obtain physical access to an HPC unit, for example from a salvage vehicle or as an aftermarket part, can perform such analysis offline to recover secrets that are reused across a fleet. These secrets can then be leveraged during later exploitation phases against other vehicles that rely on the same material. Typical weaknesses that contribute to the feasibility of secret extraction via side-channel analysis include the absence of physical or logical countermeasures, such as shielding or noise generation, which would otherwise significantly increase the number of traces required to recover a key [12]. With careful software design, it may be possible to render trace collection impractical by rotating keys or limiting the cryptoperiod, thereby reducing the window during which sufficient measurements can be obtained [13].

Another contributing weakness is the lack of hardware acceleration for certain cryptographic algorithms, which forces systems to rely on software-based implementations. Such implementations are often more susceptible to timing side-channel attacks, particularly when executed on general-purpose CPU cores, as variations in execution time or memory access patterns can leak information about secret-dependent operations [14][15]. Relying on software cryptographic implementations is even more prevalent in HPCs than MCU-based ECUs due to the rich execution environment that relies on advanced services like TLS requiring high throughput for session data encryption and authentication.

Cryptographic key handling

Weaknesses in cryptographic key handling frequently stem from how and where secrets are stored, derived, and exposed within the SoC. Fuse arrays, for example, are commonly used for chip configuration and as roots for key derivation. However, they are susceptible to exfiltration through invasive physical attacks. If a global secret is stored directly in eFuse, the risk of exposure is particularly high, as attackers with sufficient capabilities can extract fuse contents using techniques such as electron microscopy. Once such a global secret is recovered, it undermines the security of all devices that rely on it. Moreover, weak implementations of fuse-based lifecycle management can result in security downgrades if a state transition does not adequately restrict access to cryptographic keys. For example, transitioning from normal state to RMA should render certain cryptographic keys unusable.

A related weakness is the reliance on shared RTL-level keys to derive chip-specific secrets. These keys are hardwired in silicon and represent high-value targets: an attacker who recovers them can often break the security of an entire device class rather than a single unit. This creates a disproportionate impact from a single successful extraction and significantly weakens fleet-wide security guarantees.

Another common weakness pattern involves the storage of cryptographic keys in general-purpose flash memory rather than in dedicated secure storage. Keys stored in normal flash are vulnerable not only to extraction but also to tampering, replacement, or rollback. During reconnaissance, attackers routinely scan flash contents for credentials or cryptographic material. Even when such keys are encrypted, weak firmware version controls or missing rollback protections can still allow attackers to restore older, weaker states, such as a pre-production or debug-enabled image, unless a robust anti-rollback mechanism is enforced [16].

While advances have been made in mandating the use of **Hardware Platform Security Engine** (**HPSE**) technologies in embedded SoCs, weaknesses can still arise from poor architectural partitioning. In some designs, cryptographic keys are buffered or handled directly by HPSE firmware running within the secure execution environment rather than confined strictly within dedicated cryptographic engines and exposed to firmware only via handle-based access. This expands the trusted computing base and increases risk: if an attacker with

physical access discovers a vulnerability in the HPSE firmware, they may be able to extract sensitive keys. Depending on the role of those keys, this can enable decryption of proprietary IP or the generation of forged digital signatures. In the later sections, we will see how poorly designed HPSE firmware can exacerbate this security risk.

CPU vulnerabilities

If you follow CPU security disclosures regularly, you will not be surprised to learn that CPU-related vulnerabilities are frequently published by major CPU core vendors. This is not accidental. Modern CPU designs are under constant pressure to extract more performance through aggressive speculation, caching, and parallelism. When these optimizations stretch or violate fundamental security assumptions, they create fertile ground for exploitation. While Spectre- and Meltdown-class vulnerabilities are now widely understood at a high level, they represent only a subset of a broader and continuously evolving problem space. New classes of vulnerabilities regularly emerge that can alter the effective security model of the software stack in its entirety [17][18].

Many of these issues stem from subtle weaknesses in microarchitectural mechanisms that software implicitly relies upon. Examples include speculative execution paths that bypass pointer authentication code (PAC) checks [19], incomplete or delayed invalidation of translation lookaside buffer (TLB) entries across context switches [18], or inadequate containment of speculative state across privilege boundaries. The recurring theme is exposure: either secrets managed by the CPU become architecturally observable, or they become indirectly inferable through side-channel analysis. In other cases, performance optimizations allow code executing at one CPU privilege level to access, infer, or influence code or data belonging to a more privileged level. Cache behavior is a particularly common contributor, as shared caches enable well-known Flush+Reload and Prime+Probe style attacks that can leak information across execution environments that are otherwise expected to be isolated [20].

Note

Why are CPU vulnerabilities critical?

Many software security models assume that the underlying hardware faithfully enforces its advertised security guarantees. In practice, hardware design weaknesses create conditions under which those guarantees are violated. When this happens, core isolation primitives begin to erode the operating system process isolation, virtual machine separation, as well as the boundaries between the trusted-to-non-trusted domains. From an attacker's perspective, discovering such weaknesses allows for a complete break of hardware-dependent software security primitives.

Microarchitectural vulnerabilities become especially relevant once an attacker has achieved code execution on a high-performance compute (HPC) platform and seeks to escalate impact by observing or influencing other execution contexts. Spectre and Meltdown-style attacks are often introduced as reconnaissance techniques, but in post-compromise scenarios their impact is more direct. These attacks can bypass the normal privilege separation enforced by the CPU, allowing unprivileged code to transiently read or infer data belonging to more privileged contexts, such as the kernel or, in architectures with shared caches or memory interfaces, adjacent trusted execution environments or secure co-processors [21][22].

Meltdown illustrates this risk clearly. By exploiting out-of-order execution, an attacker can transiently load privileged kernel memory into a microarchitectural register state that unprivileged code should never access and then recover that data using a cache side channel after the CPU discards the illegal operation [23]. While Meltdown does not grant kernel execution or true privilege escalation, it provides a powerful privilege-bypassing information disclosure primitive: the attacker gains visibility into kernel memory without exploiting a single software defect in the operating system, relying solely on a CPU design flaw. This becomes quite relevant to HPCs where mixed criticality VMs as well as applications within a VM are relying on VM and process isolation to satisfy safety and security claims.

Susceptibility to fault injection

As discussed previously in the context of MCU-based ECUs, fault injection attacks can be highly effective for bypassing secure boot mechanisms, unlocking debug interfaces, or extracting cryptographic secrets. In HPC platforms, the attack surface for fault injection is often compounded by increased hardware complexity, longer and more fragmented boot chains, and the presence of multiple debug subsystems. Modern HPCs may expose several independent debug infrastructures such as ARM CoreSight or RISC-V debug modules, each associated with distinct privilege domains and runtime execution environments. This diversity increases the number of security checks and state transitions that can potentially be targeted by precisely timed faults.

Given limited interaction with the HPC such as the ability to issue commands, influence inputs, or execute a user-level payload, an attacker can synchronize fault injections to specific security-critical operations. A particularly attractive target is the secure boot process both due to its value and the relative ease of glitching during initial boot stages where only one or more cores are active, and the execution flow is predetermined. Recent work has shown that carefully synchronized fault injection can bypass secure-boot verification and cause a platform to boot a compromised image [24].

Once an attacker succeeds in loading malicious firmware during the boot process, they effectively gain the highest level of privilege available on the HPC. At this point, security guarantees enforced at higher software layers collapse. The attacker can implant persistent

backdoors, reconfigure isolation mechanisms, enable hidden debug functionality, or weaken cryptographic protections.

ROM weaknesses

As discussed in *Chapter 7*, the software involved in implementing secure boot is an attractive target for exploitation. This remains true for HPCs, with the key difference being the increased complexity and diversity of the bootloader stack. In HPC architectures, one or more immutable ROM components typically initiate the boot process for multiple runtime execution environments, such as application CPUs and GPUs. These ROMs load mutable firmware, often within a trusted execution environment such as the HPSE, which is then responsible for authenticating and booting software across additional execution environments including low-level firmware, safety islands, and general-purpose compute clusters.

Due to the number and heterogeneity of these components, secure boot processes are frequently pipelined or parallelized to reduce overall boot time. While this improves startup performance, it also increases the number of trust boundaries and intermediate states that must be secured. One notable weakness in such designs is the lack of revocation support for the root public key used during digital signature verification. Because this root key is typically embedded in immutable ROM or fuses, compromise of the corresponding private key, for example through a backend or signing infrastructure breach, allows attackers to produce firmware images that will be accepted as legitimate indefinitely. In such cases, affected devices may be unable to recover through software updates alone, as the trust anchor itself cannot be replaced.

Another hardware-related weakness arises from vulnerabilities in ROM code itself. ROM contents are immutable after manufacturing and therefore cannot be patched through conventional software updates. Ambiguity regarding whether ROM should be treated as hardware or software often results in inconsistent application of secure coding practices, limited static analysis, and insufficient security review during development. While some SoCs provide limited ROM patching mechanisms using OTP or similar memory, these mechanisms are constrained in size and in the number of patches they can support. As a result, vulnerabilities in ROM code tend to persist for the lifetime of the product, making early security validation of ROM implementations especially important [43].

Software weaknesses

Software weaknesses in HPCs largely stem from their complex boot stacks, the virtualized environment, and the rich operating systems. Since the MCU architecture is typically embedded inside an HPC, they also inherit the software weaknesses that we discussed in the previous chapter as well. Throughout this section, we will explore the common software weaknesses.

Basic implementation flaws

Before diving into software-specific weaknesses, it is important to cover the fundamental software implementation flaws. These weaknesses are frequently exploited to establish initial access and to construct chained attack paths across complex systems. Each year, MITRE publishes a ranked list of the most prevalent weakness classes observed in reported vulnerabilities [46].

Two implementation weaknesses consistently appear near the top of that list: buffer overflows and integer overflows. These weaknesses are not exotic. They are well understood, widely documented, and yet continue to appear in production-grade automotive software. To demonstrate software implementation weaknesses, we will study the famous integer overflow weakness which will result in a buffer overflow. You are encouraged to study other weaknesses from the Top 25 list in greater depth, as the underlying concepts surface repeatedly across nearly every software layer of the HPC software stack.

Understanding integer and buffer overflows

An integer overflow occurs when an arithmetic operation generates a numeric value that exceeds the maximum representable value for a given data type. For instance, an 8-bit unsigned integer can represent values from 0 to 255. If we attempt to add 1 to 255, the result wraps around to 0 instead of becoming 256 due to the limited range of that data type.

Consider a program designed to calculate the memory required to store several **Diagnostic Trouble Codes (DTCs)** in data flash memory as shown in *Figure 8.3*. Each record occupies 64 bytes, and the program expects the diagnostic client to specify how many records they wish to store. It then calculates the total size and allocates the necessary memory.

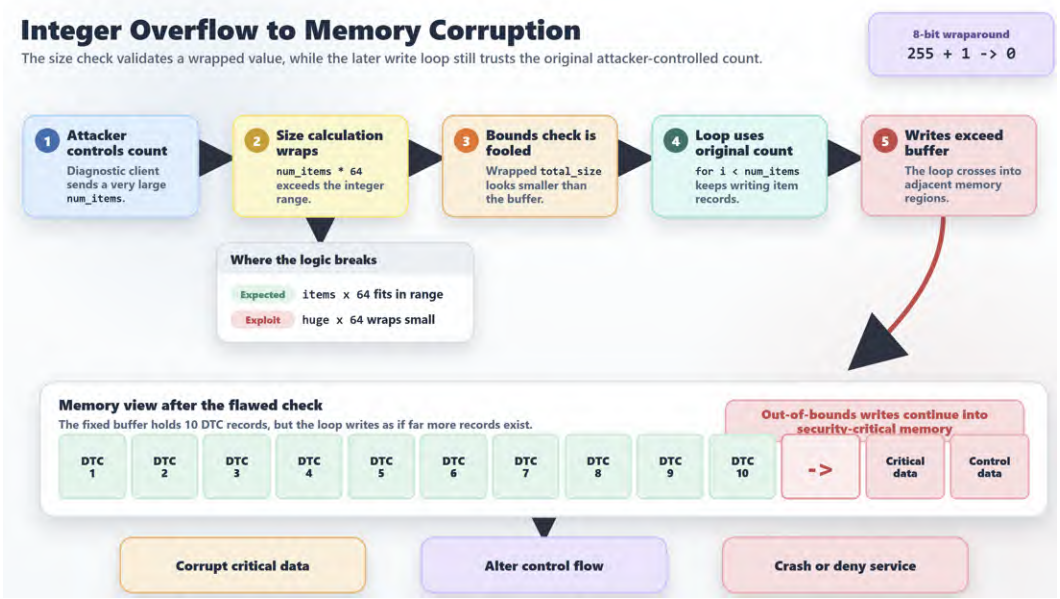


Figure 8.3: Illustrating memory corruption due to an integer overflow implementation flaw

Here's a simplified code snippet that demonstrates how the flaw results in memory corruption:

```
#define ITEM_SIZE 64
#define MAX_ITEMS 10
#define BUFFER_SIZE (ITEM_SIZE * MAX_ITEMS)

// Buffer for storing item data
char buffer[BUFFER_SIZE];

int main() {
    unsigned int num_items;
    printf("Enter the number of items: ");
scanf("%u",#_items);

    // Integer overflow vulnerability
    unsigned int total_size = num_items * ITEM_SIZE;

    // Check to avoid writing beyond the buffer - but the check is flawed due to
integer overflow
    if (total_size > BUFFER_SIZE) {
        printf("Too many items!\n");
```

```
        return -1;
    }

    // Process items - this is where the overflow can be exploited
    for (unsigned int i = 0; i < num_items; i++) {
        // Simulating processing of each item and filling the buffer
        // Vulnerable code: writes data beyond the buffer if num_items is too
large due to overflow
        printf("Processing item %u\n", i);
memset(&buffer[i*ITEM_SIZE],0,ITEM_SIZE); // Fill item slot with zeroes
    }

    return 0;
}
```

In this code, an attacker could input a very large number for `num_items`. If `num_items` is large enough, the multiplication `num_items * ITEM_SIZE` can cause an integer overflow, resulting in a smaller `total_size` than expected. Consequently, the check if `(total_size > BUFFER_SIZE)` will fail to detect a buffer overflow, allowing the program to proceed with insufficient buffer space.

Exploitation

By exploiting this integer overflow, an attacker can cause the program to write beyond the bounds of the allocated buffer. In the for loop, the program writes data into the buffer based on the user-supplied `num_items`, potentially overwriting adjacent memory regions. This is the famous buffer overflow which is leveraged to inject malicious payloads into memory and overflow the stack or the heap. This out-of-bounds write can lead to various adverse effects, such as the following:

- **Corrupting critical data**: Overwriting important variables or control structures, leading to unpredictable system behavior
- **Altering control flow**: Modifying function pointers or return addresses, potentially allowing execution of unintended code
- **System crashes**: Causing access violations, resulting in denial of service

In addition to the above flaws, lack of proper input validation and the existence of race conditions continue to be useful weaknesses to inject code into memory and abuse privileged services. Learning what each weakness is and how it can be exploited is a mandatory skill for penetration testers looking to chain weaknesses to pull off sophisticated exploits.

Note

Software implementation flaws are not limited to HPC software but are equally relevant in MCU-based ECUs that we studied in *Chapter 7*.

Insecure bootloader and OS boot configuration

A weakness in the design or implementation of any component in the boot stack can break the chain of trust and result in booting malicious code. One such weakness is lack of read-only memory protection of memory regions initialized by an early boot component which are left writable after control is transferred to the next stage. This can create opportunities for memory corruption by an attacker who has limited access to memory during early boot. Similarly, if a boot component fails to correctly restrict peripheral access to the intended bus master or execution environment, the system may transition into the operational state with an insecure hardware configuration.

In many systems, a bootloader such as U-Boot, UEFI, or a custom second-stage loader is responsible for loading the Guest Operating system such as Linux or Android. To support development and testing, many devices, including automotive HPCs, allow secure boot to be bypassed if the HPC is still in a pre-production state. If an attacker gains physical access to an HPC and the bootloader is either left unlocked or can be unlocked without proper authorization, they can boot a modified GuestOS. This effectively bypasses higher-level security mechanisms enforced by the original OS. Once an attacker gains the ability to execute arbitrary code during boot, they can install persistent modifications such as bootkits or rootkits that survive normal software updates.

Bootloaders are also susceptible to common implementation flaws as well as fault-injection-assisted bypasses of their protection logic. Research on embedded bootloaders shows that vulnerable design and implementation patterns can be combined with binary analysis and glitching techniques to defeat bootloader security mechanisms and gain control of protected memory [44].

Less sophisticated but equally impactful attacks exploit misconfigurations in the boot chain. For example, if the root public key used to verify digital signatures during boot is mistakenly provisioned with a development key rather than a production key, attackers may be able to obtain the corresponding private key, which is often widely distributed for development and testing purposes. This would allow unauthorized firmware images to be accepted as legitimate. For this reason, penetration testers are commonly tasked with reviewing boot software and provisioning processes to identify such weaknesses before the product ships.

Even when the chain of trust itself remains intact, failure to initialize the system into a secure runtime state represents another recurring weakness in HPC platforms. Many security assumptions depend on static configuration of hardware partitioning, memory regions, and peripheral access controls during early boot. Overly permissive configurations may leave the system exposed if a later-stage application becomes compromised through a software vulnerability.

Finally, insecure boot enforcement and weaknesses in hardware trust anchors remain critical concerns in post-compromise scenarios. Secure boot is intended to ensure that only firmware signed by the vehicle manufacturer or ECU supplier can persist across reboots. However, incomplete secure boot coverage can negate this guarantee. In some designs, early boot stages are verified, but later components such as the operating system or application images are loaded without signature verification due to startup timing or performance considerations. Attackers can target these unverified stages to install persistent malware.

Unnecessary services and expanded attack surfaces

HPCs often include a variety of services, daemons, debug interfaces, and OS packages that are not required for the vehicle's primary functionality. Each unnecessary component expands the attack surface by opening ports, handling external input, exposing privileged operations, or providing development functionality that should not remain enabled in production.

Common examples include default web servers, outdated SSH or telnet services, sample applications, demo code, vendor-specific maintenance services, and manufacturing or development daemons. Each service may contain its own vulnerabilities, weak authentication, unsafe defaults, or excessive privileges. For this reason, attackers and red-team testers systematically enumerate listening ports, running processes, installed packages, and exposed management interfaces on an HPC to identify weak entry points.

Debug and console interfaces deserve particular attention because they often provide powerful access with fewer safeguards than production services. Exposed shell prompts, unsecured UART consoles, open telnet services, or weakly protected debug endpoints can give attackers direct interaction with the operating system. Similarly, Linux debug interfaces such as `/sys/kernel/debug` or debugfs can expose dangerous kernel-level controls if permissions are misconfigured. For example, CVE-2019-14100 in the Snapdragon Automotive platform allowed register writes through debugfs; the vendor later disabled register writing through debugfs by default to close the issue [37].

Overprivileged services and permission misconfigurations

Least privilege is a fundamental security principle. When HPC software components run with more permission than their function requires, they create opportunities for attackers to amplify the impact of an initial compromise. In practice, it is not uncommon to find daemons,

services, or applications on HPCs running with root or system-level privileges without a strong functional justification.

For example, exploiting a buffer overflow in an infotainment media player might normally allow an attacker to crash or control only that process. If the media player runs as root, however, the same vulnerability can grant root-level access within the guest VM. If safety-critical applications or sensitive services share that VM, the attacker's impact can expand significantly. For this reason, penetration testers often treat elevated services as high-value targets.

Improper file and directory permissions create similar risks. Linux and Android rely heavily on discretionary access control (DAC), and in distributed HPC development environments, permission settings can easily become inconsistent. Unrestricted read access, unnecessary write permissions, or executable access on sensitive files can enable lateral movement or privilege escalation after a local compromise. For example, globally writable log files, calibration data, system configuration directories such as `/etc`, or application directories with 0777 permissions may allow attackers to alter system behavior, plant executables, or inject content later consumed by privileged services.

Leftover development artifacts are another common escalation path. Production HPC images may inadvertently include debugging tools, helper binaries, scripts, or test utilities with `setuid-root` permissions or permissive access control lists. Attackers actively search for these artifacts because they can provide a direct path to elevated privileges.

Mandatory access control (MAC) mechanisms are intended to reduce this risk by enforcing stronger isolation between processes and resources. Linux-based HPCs may use SELinux, commonly adopted in Android, or AppArmor to restrict what each process can access. For example, SELinux policies can prevent an infotainment application from reading system configuration files or stop a Bluetooth service from accessing sensor data except through approved interfaces.

In practice, however, MAC protections are often misconfigured, overly permissive, or disabled for development convenience. SELinux permissive mode, for example, logs policy violations but does not enforce them, removing an important layer of defense. Similar risks arise when development or userdebug builds are deployed without proper lockdown, since these builds may allow MAC enforcement to be disabled or bypassed more easily.

Android-specific IPC and debug interface vulnerabilities

Android, which is the OS of choice for infotainment systems, brings its own set of software weaknesses that attackers can exploit, particularly in its inter-process communication and debugging features. One notable vector is the Android Binder IPC mechanism. Binder is a kernel driver and framework that allows high-level system services and apps to communicate.

It is central to Android's architecture as nearly all privileged system APIs (telephony, media, etc.) are exposed via Binder. Consequently, any kernel bug in the Binder driver can have serious consequences.

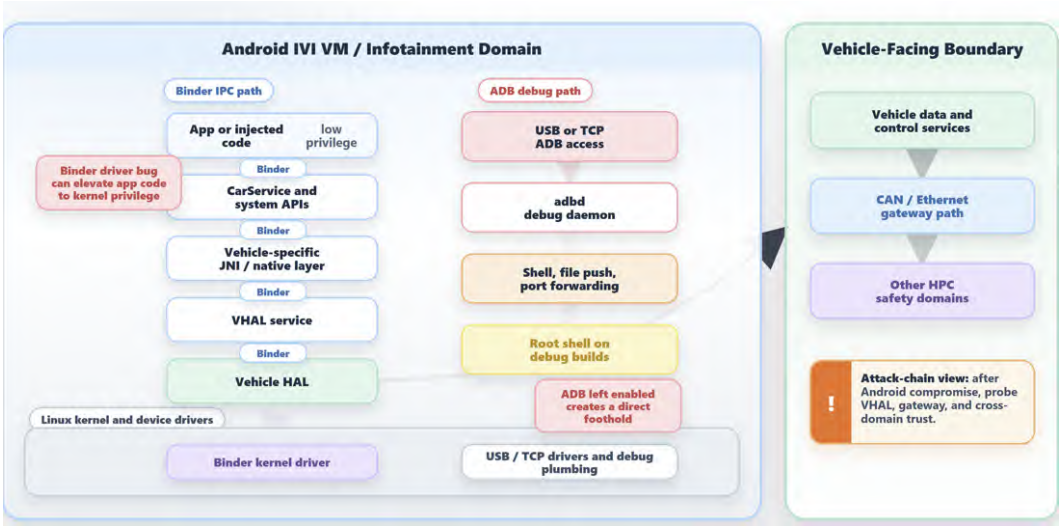


Figure 8.4: Binder exposes vehicle services, while ADB exposes direct shell access

A prominent example is CVE-2019-2215, a use-after-free vulnerability in the Binder driver. This flaw allowed the elevation of privilege from an application to the Linux kernel [38] without user interaction. In late 2019, this Binder bug was exploited in the wild as a zero-day (notably used by spyware to break out of the Android sandbox). Consequently, if an attacker manages to install or inject a malicious app into an Android-based IVI system (through social engineering, supply chain compromise, or an earlier stage exploit), a Binder vulnerability like CVE-2019-2215 could let that app take over the Android domain within the HPC and interfere maliciously with the vehicle through the vehicle HAL. Even without app installation, Binder is accessible to any code already running on the device, so a foothold via some other service can be escalated via such a kernel bug. Pen testers are encouraged to try known Android exploits against automotive IVI units, as these are essentially Android devices. Once the Android environment has been compromised, it's desirable to test how resilient the rest of the HPC domains are to malicious interference originating from the Android environment.

As mentioned earlier with Linux, debug and developer interfaces can be quite effective to establish a foothold in the system, and that is no different for Android. In Android, the primary debug interface is ADB (Android Debug Bridge). ADB allows developers to connect to the device (usually over USB or TCP) and get a root shell or upload files, etc., especially on development builds. In production, ADB is normally disabled or requires user approval and pairing. That is why as a penetration tester, you should always look for ADB being

inadvertently left behind. An open ADB connection (e.g., via USB port in the car or Wi-Fi ADB left active) could let an attacker with physical access simply spawn a shell on the HPC. If the device is running a debug build (often the case in pre-release or if the vendor neglected to switch to a release build), that ADB shell might run as root by default.

HPSE firmware weaknesses

As discussed in *Chapter 7*, a dedicated Hardware Protected Security Environment (HPSE) is intended to reduce the risk of cryptographic key exposure by securely storing keys and executing sensitive operations within a protected execution environment. In contrast to traditional MCU-based architectures, however, an HPSE in an HPC is often required to serve multiple tenants that are mutually distrustful. As a result, weaknesses in how guest VMs gain access to the HPSE services can lead to unintended exposure of one tenant's cryptographic material to another. HPSE firmware weaknesses most commonly manifest in access control and key handling logic. If the firmware does not correctly enforce isolation between requesting entities, keys may be accessible beyond their intended scope. In this sense, the security of the HPSE depends not only on hardware protections but also on the correctness of the firmware that mediates access to cryptographic services.

As noted previously, the presence of an HPSE does not guarantee that all cryptographic operations are performed within it. Automotive suppliers may still rely on software-based cryptographic implementations for performance or integration reasons, particularly in HPCs where throughput demands are high. This reintroduces the risk of cryptographic key exposure through attacks on main system memory, including physical memory probing or DMA-based attacks [39]. The performance impact of routing all cryptographic operations through the HPSE is often more pronounced in HPC architecture, increasing the likelihood that software crypto is used to augment or bypass secure hardware, thereby weakening the overall security posture. Another reason for the continued reliance on software crypto is the constant emergence of new cryptographic algorithms that may not yet be supported in hardware such as PQC and Chinese Crypto algorithms. When software cryptographic solutions are unavoidable, implementations that ignore timing side channel analysis and speculative execution are especially vulnerable to attacks that seek to expose the cryptographic keys.

Weak inter-VM isolation and resource sharing

Beyond complete guest-to-host escapes, a more common class of weaknesses arises from incomplete or improperly enforced isolation between virtual machines. A hypervisor is responsible for enforcing strict separation of memory, I/O, and execution resources so that one VM cannot interfere with another. If this isolation is misconfigured, a compromised low-trust VM may directly affect higher-criticality workloads.

A fundamental requirement is that a guest must not be able to access memory belonging to the hypervisor or to other guests. This separation is typically enforced through hardware

virtualization extensions and MMUs. However, configuration errors or poorly constrained shared-memory communication paths can undermine these guarantees. For example, when guests exchange data through shared-memory mechanisms, the partitioning model must strictly limit which regions are shared and how signaling occurs [33]. Even in the absence of direct vulnerability, covert channels may arise through the observation of shared CPU cores and caches.

In safety-critical systems, temporal isolation is just as important as spatial isolation. If CPU scheduling is not deterministic or properly budgeted, a compromised VM may starve a safety-critical VM of CPU cycles. From a penetration testing perspective, identifying such misconfigurations provides a practical avenue to demonstrate denial-of-service conditions against safety functions which is a great concern with higher levels of autonomous driving objectives. Automotive hypervisors typically offer mechanisms for temporal isolation, such as CPU time budgets or scheduling partitions. However, configuration errors can still lead to violations of freedom-from-interference requirements in the time domain, resulting in missed deadlines for safety-critical tasks.

Another shared resource that can be a source of exploitable weaknesses is DMA. Improper DMA protection can allow a compromised VM to program a pass-through device to overwrite memory belonging to the hypervisor or another VM. Without restricting the access of a DMA pass-through device to host physical memory that is explicitly granted, a bus-mastering peripheral can become an effective mechanism for violating inter-VM isolation.

Kernel vulnerabilities and outdated software versions

The widespread adoption of open-source software (OSS) within automotive HPCs, particularly through reliance on off-the-shelf components such as the Linux or Android kernel and common system libraries, introduces a substantial and well-known attack surface. When such components are integrated without rigorous vulnerability assessment and patch management strategy, they provide attackers with ready-made exploitation opportunities.

In practice, scanning recovered binaries for known vulnerabilities in OSS components remains one of the most effective techniques penetration testers use to identify viable attack paths. These flaws are frequently exploitable due to the use of fixed or long-term kernel versions that lag behind upstream security patches. Given the need for safety assessments of patches, it is common to find unpatched vulnerabilities that the OEM is simply unable to patch in a timely fashion. As a result, it is not uncommon to find dozens of known CVEs in a deployed HPC software stack, with publicly available exploits remaining active long after disclosure.

Well-known kernel vulnerabilities such as Dirty COW (CVE-2016-5195) and Dirty Pipe (CVE-2022-0847) illustrate this risk. Dirty COW was a race condition that affected Linux-based operating systems, including Android, and allowed local privilege escalation to root [34].

Dirty Pipe, disclosed in 2022, similarly allows local users to gain root privileges on Linux 5.8 and later by injecting data into read-only, root-owned files [35]. Vulnerabilities of this class enable attackers who have achieved any form of local code execution, even as an unprivileged user or application, to escalate privileges and take full control of the HPC, effectively breaking out of a sandboxed execution environment.

Widely used system utilities can also contain severe vulnerabilities. A notable example is CVE-2021-3156, known as Baron Samedit, a heap overflow in the sudo utility that went undiscovered for nearly a decade and can be exploited to obtain root privileges [36]. While exploitation requires some initial code execution, such flaws demonstrate how unpatched CVEs in standard Linux components can directly translate into high-impact automotive exploits.

Guest-to-host escape vulnerabilities

In a virtualized HPC, it is important to distinguish between two different isolation failures: process isolation escape and guest-to-host escape. A process isolation escape occurs when malicious code breaks out of an application sandbox or process boundary within the same guest operating system. This may give the attacker greater privileges inside that guest OS, but it does not necessarily compromise the hypervisor or other VMs.

A guest-to-host escape, sometimes called a GuestOS escape or hypervisor escape, is more severe. In this case, malicious code executing inside a guest VM exploits a vulnerability in the virtualized device model, hypercall interface, shared memory mechanism, virtual I/O path, or hypervisor component to break out of the guest OS boundary. A successful escape allows code execution at the hypervisor or host privilege level, collapsing the VM isolation model and potentially enabling access to other guest VMs.

Hypervisor escape vulnerabilities are well documented in enterprise virtualization and are equally impactful in automotive systems. For example, in early 2025, multiple vulnerabilities in VMware's hypervisor were chained to allow a VM administrator to escape a guest virtual machine and gain control of the hypervisor and host system [30]. Vulnerabilities such as heap overflows enabled an attacker with elevated privileges in one VM to execute code within the hypervisor's VMX process, demonstrating how a single guest compromise can escalate into full system takeover.

Each interface exposed by the hypervisor to guest operating systems represents a potential attack surface. This includes hypercalls, shared memory mechanisms, and virtual device emulation logic. A recent automotive example is CVE-2024-38420, a high-severity vulnerability in a virtual input device [31]. In this case, improper validation of device configuration data allowed a guest VM to trigger memory corruption in the hypervisor. Such a flaw could potentially lead to code execution at the host privilege level. In an automotive HPC

where an infotainment VM shares hardware with an ADAS VM, an analogous vulnerability could allow a compromised application within the non-safety-critical domain to gain control over the safety-critical domain.

Once hypervisor binaries are extracted from an HPC, vulnerability analysis of the hypervisor software becomes a natural next step. Penetration testers focus on all guest-hypervisor interaction points, including hypercalls, device emulation paths, VM exit handling, and shared memory protocols. While memory safety issues are a primary concern,logic errors can be equally dangerous. For example, IBM's PowerVM hypervisor contained a flaw where a crafted sequence of hypervisor calls from a guest could escape its partition, demonstrating that subtle state machine errors can also enable guest-to-host escapes [32].

Missing or weak authentication checks during updates

In many HPC designs, the SoC manages updates for its own software domains as well as other subsystems connected to it. If authentication checks are absent or incomplete, a compromised SoC can push malicious firmware to itself and any other ECU that depends on it for receiving software updates. A penetration tester should pay extra attention to software update services in an HPC to look for ways the software update can be abused to install malicious software. These weaknesses can span the entire communication pipeline starting with the backend and ending with the edge device that is being updated.

Typical areas of investigation include these:

- Scope of update package authentication and decryption
- **Time-of-Check to Time-of-Use** (**TOCTOU**) issues between validating update block authenticity and writing images to storage
- Lack of compatibility checking among images in a single package
- Lack of redundant bootchains to aid in recovery if the update fails
- Lack of restrictions on software reboot capabilities as a result of software update completion
- The reader is encouraged to study common weaknesses that plague software update mechanisms by understanding the Uptane Threat Model [45]

Now that we are familiar with the typical security weaknesses that are found in HPCs, it's time to put that knowledge to use and explore how these weaknesses can be abused through penetration testing techniques.

Exploring penetration testing techniques

Many of the penetration testing techniques presented for MCU-based ECUs apply to HPCs as well. While the recipe is almost the same, there are some unique tactics for each type of ECU which we will highlight throughout this section.

Even as HPCs vary in hardware and software architecture, attackers and penetration testers alike almost always follow a common blueprint for discovering vulnerabilities and designing exploits as shown in *Figure 8.5*.

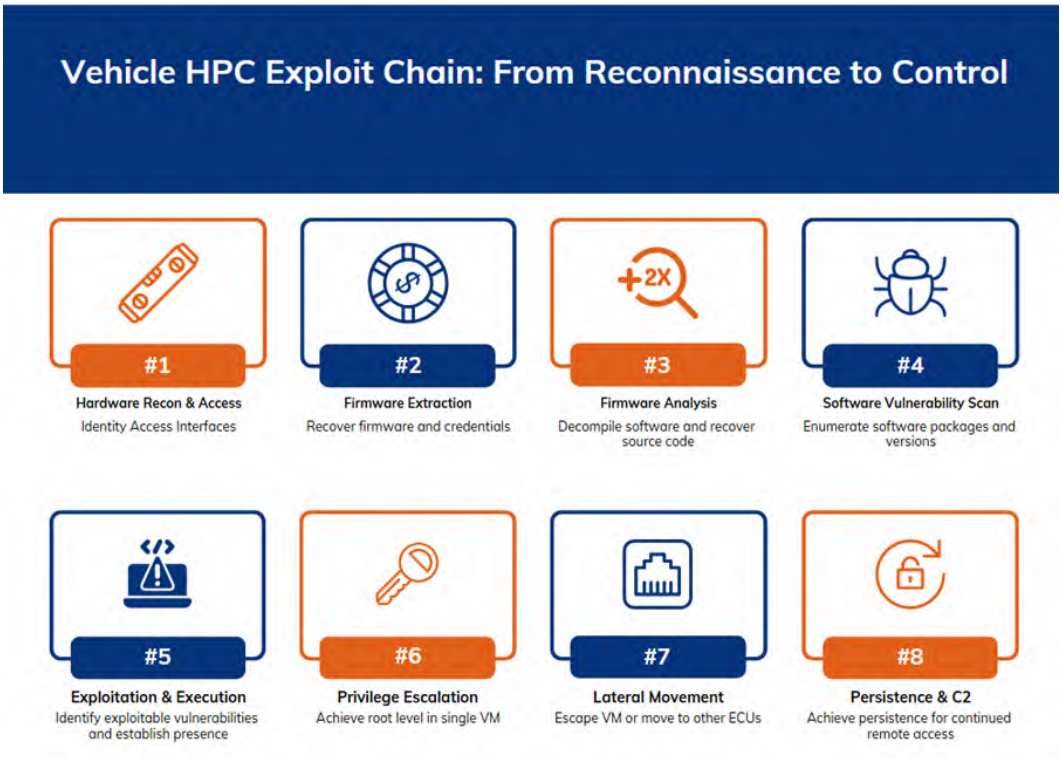


Figure 8.5: Typical flow for a pentest campaign against an HPC

Starting with the reconnaissance phase, attackers will look for exposed/unprotected interfaces to dump firmware and exfiltrate secrets. Given the software complexity and plethora of connectivity protocols, attackers will then look for software vulnerabilities that can be exploited through a connectivity interface to deliver a malicious payload. In some instances, they may exploit a communication protocol weakness without the need for a software vulnerability. From there, privilege escalation and lateral movement will ensue until the attacker is able to run arbitrary code and achieve persistence to carry out their attack objective.

Let's take a closer look at how attackers can leverage some of the weaknesses presented earlier to build effective attack chains.

Step 1: Reconnaissance and initial access

Penetration testers begin by scrutinizing the vehicle's HPC hardware for any avenue of entry, especially when performing black box testing. Often this means physically examining circuit boards for debug and communication interfaces such as JTAG ports, UART serial ports, and any accessible USB connectors. As we saw in the hardware weaknesses section earlier, JTAG is a standard chip-level debugging port that allows direct reading or writing of device memory. Testers carefully probe for exposed JTAG pads or header pins on the HPC's board, using tools like JTAGulator [40] to identify the correct pinouts. A successful JTAG connection might let them halt the processor and examine the contents of memory.

UART serial ports are another attractive target as connecting to a UART console can reveal a boot log or login prompt. In some cases, manufacturers leave a maintenance shell accessible via UART. By entering known default credentials or interrupting the boot sequence, testers might gain an initial foothold with at least limited command access. Even when these interfaces are meant to be disabled (for instance, via software locks or blown fuse bits to deactivate JTAG), skilled attackers can attempt glitching as we saw in the previous chapter to bypass such locks and enable debug or test modes. Through this meticulous hardware reconnaissance, the penetration testing team sets the stage for deeper exploitation to retrieve the firmware and software, maybe even credentials, and establish initial access into the HPC.

Another common technique during the reconnaissance stage is to enumerate all communication protocols and networking services that may allow initial access. Ideally, we want to create a networking map of the HPC relative to the rest of the E/E architecture to determine which sensor inputs it receives and which controllers it communicates with for actuation and control. Here, we will look for the typical CAN, LIN, FlexRay and Ethernet interfaces along with wireless access for example, through WiFi/Bluetooth and cellular modems. Ethernet switches are a common target as identifying the ethernet switch and tracing through the wiring harness can reveal the in-vehicle networking infrastructure.

With a map of the HPC's networking interfaces, testers can start probing the HPC attack surfaces such as network services listening on communication interfaces, APIs and daemons running on the controller, and any user interfaces (e.g. infotainment apps or Bluetooth handlers) that process external input. By enumerating all interfaces, the pentester sets the stage for later exploitation once a set of vulnerabilities has been confirmed.

> **Note**
>
> JTAG pins may be hidden within the PCB layers or hard to access due to BGA packaging. Testers will start with easily accessible interfaces and then target harder methods if these attempts are unsuccessful.

Step 2: Code extraction and analysis

Once physical or networking access yields a pathway in, the next goal is to obtain the HPC's firmware and software for analysis. Using the unlocked JTAG or bootloader interface, penetration testers dump the contents of flash memory that store a mixture of firmware binaries, configuration files, scripts, and file system contents such as the Linux RootFS. If direct access is not possible, the penetration tester can try physical extraction by desoldering storage chips like NOR flash chips and eMMC or even inserting interposers on DRAM chips to dump the booted software. The end goal here is code extraction, after which the testers can dissect the HPC's software code base off device. They often employ tools to unpack file systems and binaries (for example, using binwalk and custom scripts) to reveal the HPC's software components.

At this point, we are typically looking for useful configuration files that may contain passwords, encryption keys, as well as software packages that may contain known vulnerabilities. Here the team performs software composition analysis (SCA), which is essentially a deep audit of third-party and open-source components within the HPC. Tools like Black Duck are effective in enumerating open-source software packages that are found in the extracted binaries. Uncovering any outdated or vulnerable libraries and executables remains one of the most effective and easy ways to start building an attack chain.

In practice, this means comparing the code contents against vulnerability databases for known CVEs. While doing so, it's common to discover, for instance, an out-of-date cryptographic library, development tools, left-over shell scripts, or even an Android or Linux framework component with known vulnerabilities. Even configuration mistakes pop up during firmware analysis like hardcoded debug credentials, unnecessary services left enabled with world executable permissions, or insecure default settings that can act as pivot points. Each finding in the dumped images is triaged for exploit potential as the team plans the next phase of the attack.

Step 3: Reverse engineering to uncover software vulnerabilities

Besides open-source vulnerabilities, the penetration tester will perform firmware reverse engineering to discover zero-day vulnerabilities due to implementation bugs. Here, memory

corruption bugs, such as buffer overflows or use-after-free errors, are prime targets in any native code component.

First, the recovered binaries must be de-compiled to produce code that is easy to inspect for security vulnerabilities. As we saw in *Chapter 7*, tools like Ghidra can be useful to achieve that objective. With the code in hand, the penetration tester can apply **static analysis security testing** (**SAST**) to scan for security weaknesses. Using AI agents is especially useful in this step to filter the signal from the noise by focusing only on the weaknesses that are truly exploitable. In addition to SAST, building an environment that enables fuzzing is also a useful technique for discovering vulnerabilities.

For instance, the team may fuzz-test a software service like a video decoder to see if malformed inputs can crash the service or overwrite memory. Any crash is investigated to see if it can be turned into a reliable code execution method. Code injection flaws are another avenue, where the HPC software takes input and mistakenly treats it as code or a command. This could be as simple as a poorly sanitized command-line passed to a shell. Fuzzing may also reveal race conditions, such as **time-of-check-to-time-of-use** (**TOCTOU**) flaws. A TOCTOU bug might allow testers to, say, swap out a file or change permissions between the time the HPC's software checks it and uses it, leading to privilege escalation. The end goal of this vulnerability hunting is to discover as many exploitable flaws as possible which can later be chained to achieve a larger attack objective.

> **Note**
>
> When source code is retrieved or decompiled, AI tools can prove quite effective in analyzing code dumps to identify software vulnerabilities. This can be useful in accelerating the search for zero-day vulnerabilities.

Step 4: Exploitation and code execution

After identifying enough security weaknesses, the team must develop an exploit to demonstrate that the attack objective was achieved. Often, the initial exploit might only yield limited access, for example, running code in a less-privileged context or hijacking a single process. But even achieving non-root command execution is a critical milestone. The narrative now shifts to crafting payloads that the HPC will run. In one scenario, suppose testers find a buffer overflow in an HPC's network service. They could craft a carefully structured input (often a long string of data) that overwrites the program's return address, redirecting execution to the tester's payload. The payload might be a small piece of shellcode that opens a remote shell or adds a new user account.

In another scenario, perhaps an Android-based HPC has an app with a code injection flaw allowing the testers to exploit it to drop into the **Android Debug Bridge** (**ADB**) or to spawn a Unix shell on the device. The testers refine their exploit, so it works repeatably without crashing the system. If the HPC has exploit mitigations like **ASLR** (**Address Space Layout Randomization**), **Pointer Authentication Codes** (**PAC**) and **DEP** (**Data Execution Prevention**), the exploit needs to cleverly bypass those (often leveraging microarchitectural weaknesses that enable information leakage). You will notice that this process is iterative in which the tester alternates between exploitation and discovery as defenses usually get in the way requiring the tester to discover a way to bypass them before they proceed with further exploitation. Once successful, the penetration testers now have code running on the target HPC under their control. At this stage, they often have the equivalent of a user-level shell on the device with the ability to issue commands and observe responses.

The OEM may be satisfied with a pentest report that shows that the discovered vulnerabilities enabled user level code execution. If the pentest engagement has set a higher bar, then work must proceed to achieve higher execution privileges. From the perspective of the attackers, the car's HPC has been effectively hijacked, but the work is not done as the attackers' abilities might be constrained by the operating system's security layers.

Step 5: Privilege escalation in HPC environments

As we discussed in the section on weaknesses, HPCs running Linux or Android typically enforce privilege separation, meaning the initial code execution might be in a sandboxed app or as an unprivileged user process. As a result, to fully control the HPC, and by extension the critical vehicle functions it manages, the attackers must seek to escalate their privileges to higher levels. The penetration testing team shifts to finding a path to root privileges. Well-known kernel vulnerabilities become powerful tools at this juncture. If no known vulnerabilities exist, then more advanced analysis is needed to discover a zero-day vulnerability in the kernel itself.

Other avenues include exploiting misconfigured setuid binaries (programs that run as root). If the firmware analysis revealed any executable on the HPC that has the SUID bit set and is improperly coded, the testers can abuse it to spawn a root shell. Likewise, on Android HPCs, they might look for system services running with high privileges that trust inputs from their low-privilege context. Any flaw here can yield a compromise of the Android system user or even the Linux kernel underlying it. Sometimes the escalation is as straightforward as finding default credentials.

Unfortunately, it is not uncommon to encounter cases where the root password was set to "root" or a debugging SSH server was left enabled. More subtle routes include abusing kernel mode driver bugs since a vulnerability in a KMD will give attackers the same privilege as the kernel. By the end of this phase, the penetration testers aim to have unrestricted root-level

control of the HPC's operating system. This means they can do practically anything within the Guest VM such as modify files, read sensitive data, and send arbitrary network messages.

Step 6: Lateral movement across virtual machines

Depending on how well the HPC apps are isolated, gaining control of a single VM within the HPC may not always be sufficient. Typically, influencing the deeper layers of the E/E architecture requires lateral movement from the HPC to another VM or physical ECU that is connected to the in-vehicle network. In the case of a multi-VM HPC it is common for an attacker to want to move from the infotainment VM to the ADAS VM to interfere with ECUs in the safety domain.

A determined attacker with root access on one domain will probe configuration flaws to break out of that sandbox. This could involve exploiting the hypervisor or abusing shared resources between VMs. For instance, if two virtual machines share a common memory region without adequate input validation, a crafted attack could let the compromised domain inject code into the other.

The other path of lateral movement is onto the vehicle network. Assume an architecture where the HPC is separated from the deeper in-vehicle network through a central gateway. To directly communicate with lower-level ECUs controlling brakes, powertrain, or steering, the attacker must influence the gateway ECU itself. Armed with root access on one virtual machine within the HPC, attackers can attempt to reconfigure network interfaces to spoof communication with the gateway ECU. In practice, the penetration testers at this stage might write custom scripts or tools on the HPC to generate CAN frames and then test their ability to, say, unlock doors, manipulate instrument cluster readings, or even apply braking or acceleration.

Through patient lateral movement, a single compromised HPC can evolve into a full-vehicle compromise. It is common for OEMs to share the E/E architecture so pentesters can quickly validate their attack chains that span multiple ECUs. If this is not available, then careful analysis of the in-vehicle network topology is necessary to map out trust relationships and pivot points to demonstrate how an initial foothold can cascade into multi-ECU control.

Step 7: Persistence and remote command and control

As a penetration tester, if you have gotten this far, then you have likely passed the objectives of your penetration testing engagement with flying colors. However, for attackers, ensuring their presence survives a system reset is an additional crucial target. One straightforward method is to establish a persistent backdoor on the HPC.

For example, the attacker might add their own SSH key and enable the SSH service to remotely connect to the HPC at will, with authentication baked in. Additionally, they may modify startup scripts or firmware so that even under a power cycle, their malicious services or

implants launch automatically. On a Linux-based HPC, this could involve adding a script to `/etc/init.d` or a systemd service that starts a backdoor listener. In an Android-based system, they might tamper with the boot partition or an **OTA** (**Over-The-Air**) update mechanism to reinstall their payload after updates.

The testers also may set up a **command-and-control** (**C2**) channel suitable for a car's connectivity. For example, the attacker could run a lightweight agent that periodically reaches out to a C2 server (using HTTPS or even hiding in normal telemetry data) to fetch commands. Alternatively, the HPC could be instructed to open a reverse shell to a specific host whenever it's online. The idea is to simulate an attacker being able to come back days or weeks later and still have an open door into the vehicle. In effect, the car remains "*owned*" by the attacker until the entire system is patched.

In advanced engagements, penetration testers expand their scope beyond the vehicle itself, highlighting the risk of supply chain attacks. Rather than attacking the HPC in the field, a red team might attempt to tamper with the software before it's ever installed. We will talk about supply chain attacks and how to handle them next.

Advanced red team tactics: supply chain compromise

Supply chain attacks are a highly efficient way to compromise many systems at once by inserting malicious code into a widely used component [41]. This may require infiltrating a Tier-1 supplier's development network or build servers. For example, if attackers manage to infect the update package of an HPC's firmware (or compromise the code-signing infrastructure), they could potentially push a legitimate-looking yet backdoored update to all vehicles of a certain model. Such an approach bypasses many in-vehicle defenses because the malicious firmware arrives signed and trusted.

While largely hypothetical, the continuous improvement of defensive security will likely force sophisticated attackers, especially nation states, to shift to supply chain attacks by implanting insiders rather than investing heavily in defeating security controls after they have been deployed. A related tactic is targeting third-party dependencies in the car's software supply chain. As in classic IT supply-chain exploits, an attacker might contribute code to an open-source project used in the HPC or exploit a compromised library that gets included in the vehicle firmware. Once the poisoned component is integrated, every vehicle becomes vulnerable by design.

Now that we have immersed ourselves in penetration testing techniques, it's time we shift our focus to real-life hacks to see how these hypothetical methods can be put to use. This should solidify our original premise that attacks against HPCs are feasible and not merely the subject of hypothetical scenarios.

Case studies of publicly disclosed compromises involving HPCs

To solidify the concepts presented so far, we examine three case studies of publicly disclosed compromises involving HPCs. Each case highlights different attack vectors, attack steps, and outcomes. Drawn from security research, they illustrate the progression of car hacking over the past decade. As we did in prior chapters, the attack steps are mapped to the MITRE ATT&CK framework for structured understanding. As evident below, each case study relies on more than one security weakness to achieve the attack objective. You are encouraged to spot the security weaknesses in each case study using the material we have covered so far.

Case study 1: Mazda IVI

Mazda's **Connectivity Master Unit** (**CMU**) is a typical infotainment system that manages Bluetooth, navigation, and smartphone integration. In this HPC, the user-facing applications are hosted on a rich OS while CAN connectivity is handled by a companion MCU running an RTOS.

In this case study, researchers identified multiple vulnerabilities, affecting models like Mazda 3 from 2014 to 2021. These vulnerabilities stem from insufficient input sanitization, allowing attackers to execute arbitrary code with root privileges by connecting specially crafted USB devices to the system.

> **Note**
>
> Source: `https://www.zerodayinitiative.com/blog/2024/11/7/multiple-vulnerabilities-in-the-mazda-in-vehicle-infotainment-ivi-system`

Figure 8.6 illustrates this attack chain:



Figure 8.6: Attack chain from USB auto-update to Vehicle MCU interference

Discovery and reconnaissance

During the discovery phase, the researchers first established a clear understanding of the CMU hardware layout, including the primary application SoC, the companion MCU, and the external non-volatile storage devices. This initial hardware mapping was followed by firmware extraction and reverse engineering to understand the role of each major software component.

By tracing the software update execution path, the researchers identified the libraries and internal functions responsible for update handling and validation. One particularly important observation was that several operations were performed on the update package before signature verification was completed. This ordering immediately revealed a high-value inspection point for potential vulnerabilities, because any logic that processes untrusted update data prior to cryptographic verification becomes part of the effective attack surface.

The first vulnerability was discovered in the USB communication path between the CMU and Apple mobile devices. A malicious device can supply a crafted, malformed string that is passed directly into an internal database operation without proper sanitization, resulting in an SQL injection vulnerability. Because the process handling this interface executes with root privileges, the injected SQL statements are executed within a privileged context. This enables direct manipulation of the underlying SQLite database and can be leveraged for actions such as data modification, file creation, and corruption of system state. While this vulnerability does not by itself guarantee immediate arbitrary code execution, it provides a powerful primitive that can be combined with other weaknesses to escalate toward full system compromise.

Hardware examination

The CMU is implemented as a single printed circuit board integrating a Freescale controller, part number SCIMX06DAVT10AC, together with external NAND flash and eMMC storage

devices, a Renesas microcontroller, part number R5F35MCEJFF, and a Murata Bluetooth communication module. System connectivity is exposed through several interfaces, including multiple USB ports, a CAN interface, and a serial console.

From an architectural perspective, the board layout and component partitioning indicate a modular design in which application-level processing is handled by the Freescale SoC, while selected auxiliary and real-time functions are delegated to the Renesas MCU. This form of functional separation is common in automotive head units and infotainment systems, particularly when CAN access and timing-sensitive tasks are involved.

Although such partitioning is often intended to limit fault propagation and simplify system design, it does not, by itself, constitute a security boundary. As we will see in the following sections, the effectiveness of this separation ultimately depends on how trust relationships and communication paths between the application processor and the companion MCU are implemented.

Software composition analysis

The main application SoC executes a Linux-based operating system, as confirmed through serial console output and bootloader logs. During early boot, the system mounts multiple file systems, many of which are configured with read–write permissions. From an attacker's perspective, this significantly expands the post-compromise persistence and tampering surface, particularly when system services and update components reside on writable storage.

Persistent storage is distributed across mtdblock, ffx01, and mmcblk partitions. This heterogeneous storage layout enables multiple paths for filesystem-level manipulation, including modification of update artifacts, configuration files, and executable components that are later consumed by privileged services.

Console authentication behavior was not publicly documented. However, verbose boot and initialization logs exposed detailed system information, including service startup behavior and internal component naming. This level of observability materially reduces reverse-engineering effort and directly supports exploit development by revealing candidate execution paths and privileged services.

The CMU supports firmware updates through a diagnostic menu and consumes update packages in the form of `.up` files supplied via USB mass-storage devices. These update files are implemented as password-protected archives and contain critical system components, including the root filesystem, bootloaders, and configuration data. The update pipeline performs structural validation, certificate extraction, and signature verification. However, several parsing and handling operations are executed before cryptographic verification is completed, which is another clue for possible exploitation. As discussed earlier, any logic that

processes untrusted update content prior to integrity validation becomes part of the effective pre-authentication attack surface.

Reverse engineering of the update workflow identified the update services `svcjciupdatea.so` and `svcjciupdates.so`, together with supporting libraries including libjcireflashua.so and libjcisecurity.so, as the primary components responsible for loading, validating, and executing update instructions. The researchers focused on functions responsible for file discovery, archive handling, extraction, and command invocation, as these code paths directly bridge attacker-controlled update content and privileged system operations.

This design choice is critical. The update mechanism executes with elevated privileges and has direct access to system files and executable paths. Consequently, weaknesses in update package parsing or command handling immediately translate into high-impact code-execution primitives.

The following vulnerabilities were identified in the CMU software stack. Collectively, they demonstrate how peripheral access and update functionality can be chained to reach full system compromise:

- **SQL injection in DeviceManager (CVE-2024-8355 / ZDI-24-1208)**: A lack of input sanitization in the `eInsertDeviceEntry` function within `/jci/devicemanager/libdevicemanager.so` allows an attacker to inject SQL statements by spoofing iAP serial numbers. The injected input is written directly into the SQLite backend (`DeviceDatabase.db`) without validation.
  This enables manipulation of database content and can be leveraged for file creation, information disclosure, and database corruption. Because the DeviceManager component operates with elevated privileges, the impact of this weakness extends beyond data integrity and directly affects system-level security guarantees.
- **Command injection via REFLASH_DDU_FindFile (CVE-2024-8359 / ZDI-24-1191)**: The `REFLASH_DDU_FindFile` function in libjcireflashua.so constructs operating-system commands using attacker-controlled input without proper sanitization. During firmware update processing, this allows an attacker to inject arbitrary shell commands. In practical terms, this provides a reliable remote code execution primitive in the update execution path.
- **Command injection via REFLASH_DDU_ExtractFile (CVE-2024-8360 / ZDI-24-1192)**: The `REFLASH_DDU_ExtractFile` function exhibits the same class of input-handling weakness. Unsanitized parameters are incorporated into system command execution during file extraction, enabling direct command injection and leading to complete system compromise.

- **Command injection via UPDATES_ExtractFile (CVE-2024-8358 / ZDI-24-1190)**: Within `svcjciupdates.so`, the `UPDATES_ExtractFile` function processes update archives using attacker-controlled file paths and parameters without sufficient validation. This enables both path manipulation and command injection during update `extraction,again` resulting in privileged code execution.
- **Missing root of trust in the application SoC (CVE-2024-8357 / ZDI-24-1189)**: The CMU bootstrap sequence on the application SoC does not enforce authentication of boot components. As a result, an attacker with write access to the relevant storage can modify bootloader code, persist malware across reboots, and obtain execution control before the operating system is loaded.
  This fundamentally undermines any higher-level software security mechanisms implemented within the Linux environment.
- **Unsigned code execution in the VIP MCU (CVE-2024-8356 / ZDI-24-1188)**: The auxiliary VIP MCU responsible for CAN and LIN communication accepts firmware images without signature verification. An attacker can therefore replace the VIP firmware with a modified image and gain direct control over in-vehicle network communication.

This weakness collapses the intended separation between the application processing domain and vehicle communication functions. In effect, a compromise originating in the infotainment or application domain can be extended into the CAN bus, directly increasing the blast radius into safety-relevant vehicle systems.

Step-by-step breakdown of the attack chain

Exploitation of the described command injection vulnerabilities is possible because software update installation can be triggered automatically upon connecting a USB mass storage device; this is a "known feature" which is activated by simply placing an empty file named `jci-autoupdate` on the storage device.

Setup and initial exploitation

Exploiting the OS command injection vulnerabilities in the Mazda CMU is remarkably straightforward, requiring only physical access to the vehicle's USB port. An attacker prepares a FAT32-formatted USB mass storage device with a maliciously crafted file whose name contains the OS commands to be executed. The filename must end with the `.up` extension to be recognized by the software update mechanism.

For easier exploitation, the attacker can utilize vulnerability ZDI-CAN-23420, which does not impose additional requirements like crafting a valid update file. Moreover, software updates

can be triggered automatically by placing an empty file named `jci-autoupdate` on the USB device, activating the "auto-update" feature upon connection.

Execution and persistence

Once the USB device is connected to the vehicle and the system reads the crafted file, the embedded commands execute, granting the attacker access to the CMU. This access enables manipulation of the root filesystem, such as installing backdoored system components that persist across reboots. These changes ensure the attacker retains control over the compromised system indefinitely.

Lateral movement

After achieving a foothold in the CMU, the attacker can move laterally to the VIP microcontroller. This involves installing custom-crafted firmware on the VIP, granting unrestricted access to vehicle networks like CAN and LIN. While the research did not explore specific impacts, this control could theoretically enable attackers to manipulate vehicle operations, affecting safety-critical functions.

This case study demonstrated to us how common weaknesses can be relatively easily chained together to mount successful attacks on an HPC which can result in direct control of more critical components on the vehicle internal network. Next, we'll discuss the second case study which relied on more sophisticated attack techniques.

Case study 2: Tesla IVI

This case study demonstrates a remote, zero-click compromise of an IVI system by exploiting a memory-safety flaw in a proprietary Bluetooth stack that is reachable prior to authentication. By extracting and reverse-engineering the device firmware, the attackers identified a root-privileged Bluetooth service lacking common exploit mitigations and triggered a Use-After-Free condition in the L2CAP segmentation and reassembly logic while processing ACL payloads delivered over an established HCI link. Through controlled heap manipulation, they obtained arbitrary read and write primitives, leaked heap, library, and stack addresses to defeat ASLR, and redirected execution using a crafted ROP chain to achieve root code execution. The compromise enabled persistent access and remote command-and-control, with impacts ranging from IVI takeover and user data exposure to in-vehicle eavesdropping and potential pivoting into connected mobile devices.

Note

Sources:

- `https://www.youtube.com/watch?v=q6X6L82J2Mk`
- `https://2024.hexacon.fr/slides/Evdokimov-0_click_RCE_on_the_IVI_component.pdf`

This case study illustrates how a single remotely reachable vulnerability in a privileged wireless stack can escalate into full HPC-class system compromise. Let's explore the steps they took to achieve their attack objectives.

Discovery and reconnaissance

During the discovery phase, attackers conducted extensive reconnaissance on the Alpine Halo9 iLX-F509 In-Vehicle Infotainment (IVI) system. This device is an external IVI system featuring a touchscreen display and various connectivity options, including USB, **Wi-Fi, and Bluetooth**. It supports Apple CarPlay and Android Auto. The attackers focused on the device's **Bluetooth stack**, which is a proprietary implementation that they suspected was potentially less scrutinized for vulnerabilities.

To gain a deep understanding of the system, the attackers obtained the firmware from the device's embedded MultiMediaCard (eMMC) chip without desoldering it. By reverse-engineering the firmware, particularly the btapp binary responsible for Bluetooth functionality, they discovered that it was an ARM 32-bit executable running as root. The binary lacked several security mitigations, such as stack canaries and Position Independent Executable (PIE) support, making it an attractive target for exploitation. Additionally, the presence of symbols in the binary simplified the reverse-engineering process.

The attackers analyzed the Bluetooth protocols implemented by the device, focusing on the Host Controller Interface (HCI) and Logical Link Control and Adaptation Protocol (L2CAP) layers. They identified a Use-After-Free (UAF) vulnerability in the handling of HCI Asynchronous Connection-Less (ACL) data fragments within the `prh_l2_sar_data_ind` function. This vulnerability could be exploited without user interaction, qualifying it as a zero-click vulnerability.

Step-by-step breakdown of the attack chain

Let's explore each step of the attack chain in detail, as illustrated in *Figure 8.7*:

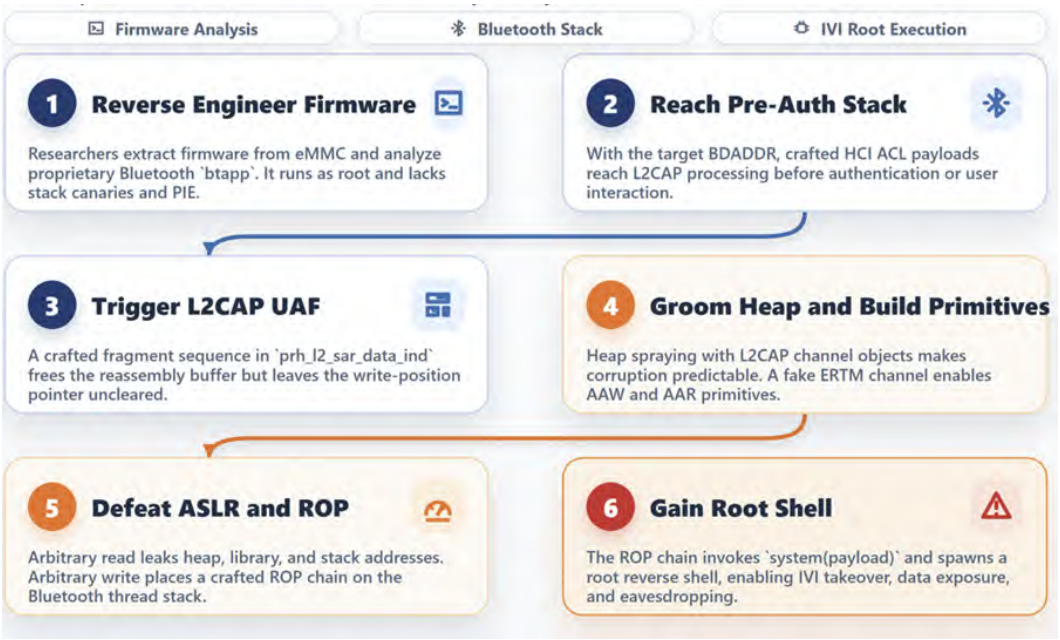


Figure 8.7: Attack chain from zero click to reverse root shell

Initial access

Initial access was obtained through a zero-click vulnerability in the Bluetooth stack. Because L2CAP traffic is processed before any authentication or user interaction takes place, an attacker can deliver specially crafted Bluetooth packets directly to the target device.

The only prerequisite for targeting the device is knowledge of its Bluetooth device address (BDADDR). In practice, this can be obtained by sniffing nearby Bluetooth traffic using tools such as Ubertooth, inferring the address from the Wi-Fi module MAC address in coexistence designs, or brute-forcing the lower bytes of the address when the upper bytes are known. From an attacker's perspective, this provides a fully remote, unauthenticated entry point into a privileged protocol stack.

Use-after-free trigger

The core vulnerability is a Use-After-Free (UAF) condition that occurs during the reassembly of fragmented Bluetooth data. The system uses an internal link object to track the state of an established connection, including a data buffer and a write-position pointer that marks where the next fragment should be stored.

The exploit is triggered by sending a specific sequence of fragments:

- An initial fragment sets up a valid data buffer. A second "start" fragment is sent that intentionally fails a length validation check
- Due to a logic error, the system frees the data buffer but fails to clear the internal write-position pointer

This leaves a dangling pointer still aimed at a location in the middle of a freed heap chunk. When the attacker sends subsequent data fragments, the system continues writing to that freed memory.

Heap overflow

The attackers then transform this UAF into a heap overflow. By providing a malicious length value in a new fragment, they trick the system into believing the current data reassembly is much larger than the original buffer size. As additional fragments arrive, the system uses the dangling pointer to write data beyond the boundaries of the original heap allocation, allowing the attackers to overwrite adjacent objects in memory.

Heap spraying and manipulation

To ensure the overflow hits a useful target, the attackers perform heap spraying. They create approximately 50 logical channel objects to fill the heap with predictable structures. This ensures that when the overflow occurs, it predictably corrupts a target channel object. The outcome of this step is a deterministic heap layout that can be reused across multiple exploitation attempts.

Arbitrary write primitive

The attackers achieve an arbitrary address write by utilizing Enhanced Retransmission Mode (ERTM), a complex channel type they "fake" by overwriting a standard channel object. By corrupting the internal data reassembly pointer within this fake ERTM structure, they can redirect the pointer to any location in the system's memory. Any subsequent data sent to that channel is then written to the attacker's chosen address. This primitive is the central building block for the remainder of the exploit, as it enables direct modification of control-flow-relevant memory and critical data structures.

Arbitrary read primitive

An arbitrary address read is obtained by abusing the ERTM retransmission logic. The attackers craft a "retransmission request" that forces the system to resend data it has already received. By using their write primitive to overwrite the internal retransmission list with a target memory address, the system is tricked into "re-sending" the contents of that memory back to the attacker. This read primitive is used to leak heap pointers, shared library base addresses, and stack addresses.

Bypassing ASLR

Using the AAR capability, the attackers leak addresses from the heap, libc, and the stack. These disclosures are sufficient to defeat Address Space Layout Randomization (ASLR). Once the base addresses of relevant modules and memory regions are known, control-flow redirection can be performed in a stable and repeatable manner. Without these leaks, the exploit would remain probabilistic and fragile.

Privilege escalation

The Bluetooth application binary (btapp) executes with root privileges. As a result, any code execution achieved within this process context immediately translates into root-level control of the system. However, execution must still be redirected to attacker-controlled logic. To achieve this, the attackers construct a return-oriented programming (ROP) chain using gadgets located in known memory regions disclosed through the address leaks.

Code execution

The attackers use the AAW primitive to overwrite the stack of the Bluetooth processing thread with a crafted ROP chain. The chain is constructed to invoke the system(payload) function. Successful execution of the chain results in a root reverse shell on top of TCP/IP being spawned on the target device, providing full interactive control of the IVI system. At this stage, the attacker has transitioned from a wireless protocol-level vulnerability to a complete system compromise.

Impact

The root reverse shell establishes a channel between the compromised IVI system and the attacker infrastructure. This channel enables remote command execution and real-time interaction with the device. From an operational standpoint, the IVI system effectively becomes a remotely managed endpoint inside the vehicle, allowing the attacker to deface the display, steal phone book information, or eavesdrop via the external microphone.

This case study demonstrated how attackers combine memory corruption, information disclosure, and control-flow manipulation to progress through an attack chain. In the following case study, we move beyond this baseline and examine an even more sophisticated intrusion scenario, highlighting the additional techniques and operational complexity required to expand the attack from the boundary of the HPC.

Case study 3: Tesla Pwn2Own 2023

This case study presents a three-stage attack chain against a Tesla infotainment system that begins with a paired-device Bluetooth heap overflow in the Bluetooth server process, escalates through a shared memory region with the Wi-Fi and Bluetooth chipset into Wi-Fi firmware

execution, and then abuses a Wi-Fi Linux driver vulnerability to achieve Linux kernel-level code execution on the infotainment system.

From this unrestricted root position, the attackers pivoted over the in-vehicle Ethernet network to the Security Gateway and exploited a Time of Check to Time of Use (TOCTOU) logic flaw in its firmware update workflow, allowing a signed update to be replaced on removable storage with a malicious image applied without re-verification.

By modifying the Security Gateway bootloader to disable software-only secure boot, the attackers obtain persistent code execution on the gateway itself and reuse existing firmware services to transmit arbitrary CAN messages to downstream ECUs. The result is a full cross-domain compromise from the multimedia HPC to the in-vehicle control network, demonstrating how chained memory safety and update logic flaws can be combined to bypass sandboxing, defeat secure boot assumptions, and enable durable, vehicle-wide control. *Figure 8.8* illustrates the end-to-end progression of this attack chain

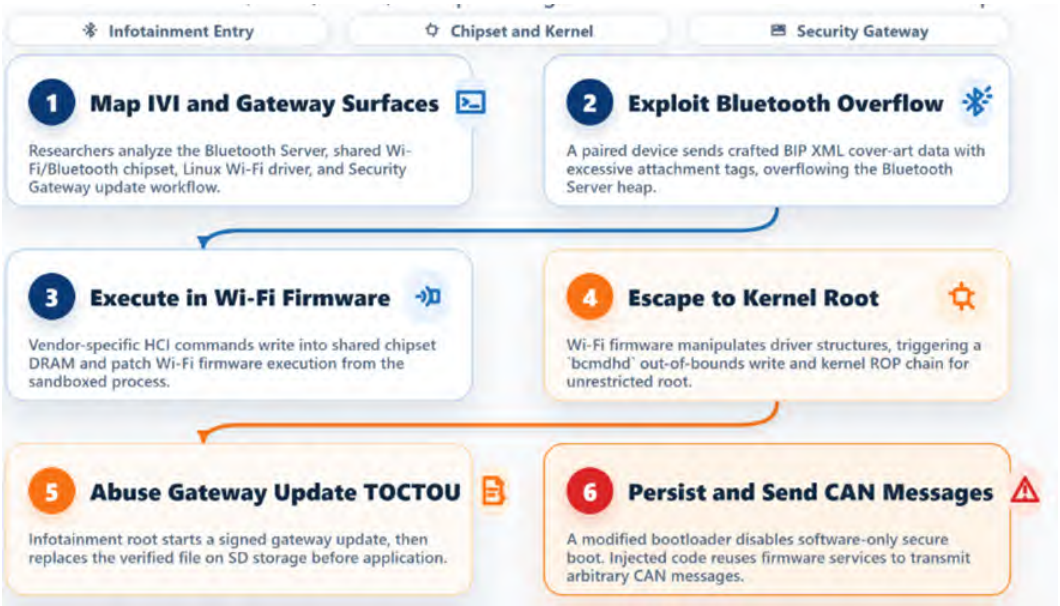


Figure 8.8: Three phases of exploit construction

Note

Source: `https://www.youtube.com/watch?v=jsZOtKuGoJk`

Discovery and reconnaissance

During the discovery phase, security researchers conducted an in-depth analysis of the Tesla Model 3's systems to uncover potential vulnerabilities that could be exploited. They focused on the vehicle's infotainment system, a critical component in the multimedia domain that interfaces with the vehicle domain via the Security Gateway which is located on the same PCB board. This infotainment domain runs on an AMD Rizen processor using a Linux operating system, equipped with Wi-Fi and Bluetooth connectivity.

The researchers obtained hardware for testing, including infotainment boards and displays, and utilized both static analysis tools (such as Ghidra and IDA Pro) and dynamic instrumentation techniques. They benefited from Tesla's security researcher program, which provided them with SSH access to the infotainment system, allowing for comprehensive analysis and testing.

Key attack surfaces identified included the following:

- **Bluetooth stack**: The Bluetooth Server process (bthserv), responsible for handling Bluetooth communications, was compiled without Position Independent Executable (PIE) support. Its large attack surface, due to being written in C and using a custom heap allocator, made it an attractive target for exploitation.
- **Wi-Fi firmware and driver**: The shared Wi-Fi/Bluetooth module presented an opportunity to transition from Bluetooth to Wi-Fi, with potential vulnerabilities in the Wi-Fi firmware and its interaction with the Linux kernel driver (bcmdhd).
- **Security gateway**: Acting as a bridge between the multimedia and vehicle domains, the Security Gateway is responsible for filtering Controller Area Network (CAN) messages and managing firmware updates for other Electronic Control Units (ECUs) within the vehicle.

Step-by-step breakdown of the attack chain

Let's discuss each phase of this attack chain in detail.

Phase 1: Initial access

In phase 1, the attackers achieved initial access by exploiting a heap overflow vulnerability in the Bluetooth Server process. This required the attacker to have a paired device with the car's Bluetooth system, achievable through a compromised phone connected to the car or physical access to the car's touchscreen.

Here's what happened in this phase:

- **Exploitation of Bluetooth heap overflow**: A heap overflow was identified in the parsing function of the Bluetooth stack Basic Imaging Profile (BIP) within the Bluetooth Server. This profile allows the phone to send cover art images to be displayed on the screen when playing music. Due to an implementation bug in the parsing function, the researchers could send a specially crafted XML file containing an excessive number of tags (beyond the expected limit), triggering an overflow that wrote data beyond the allocated buffer's boundary.
- **Arbitrary write primitive and code execution in Bluetooth server**: Exploiting the custom heap allocator used by the Bluetooth Server, the attackers gained an arbitrary write primitive. Thanks to the absence of ASLR and Position Independent Execution (PIE) support, the addresses that needed to be overwritten were fixed in memory making the attacker's job easier. They overwrote a global variable that held a function pointer used in a timer callback within the application's main loop. By controlling this function pointer, they redirected execution flow to their injected code, achieving code execution within the Bluetooth Server process.

Phase 2: Privilege escalation to Wi-Fi firmware

In phase 2, the researchers figured out that the infotainment system was heavily sandboxed using Kafel for syscall filtering, AppArmor for mandatory access control, and Minijail for file system isolation, so compromising that process limited them to the abilities that were assigned to it and resources it already had access to. Given that the Bluetooth Server process operated with limited privileges and was properly sandboxed, the attackers sought to escalate their privileges to the kernel level. They leveraged the shared DRAM memory with the Wi-Fi/Bluetooth chipset, using vendor-specific Host Controller Interface (HCI) commands to write directly to the Wi-Fi firmware's memory. This allowed them to inject code into the Wi-Fi module from the Bluetooth Server.

The following actions took place in this phase:

- **Execution of code in Wi-Fi firmware**: By injecting their code into unused memory regions of the Wi-Fi firmware and patching the firmware's idle task to jump to their code, the attackers gained execution within the Wi-Fi firmware. This position provided a more advantageous foothold, as the Wi-Fi firmware interacts closely with the host system's kernel driver over the PCI Express interface, using Direct Memory Access (DMA) and mailbox communication mechanisms. In a way, they had to move laterally outside the SoC boundary into the WiFi chipset only to find a way back into the Linux kernel of the infotainment system!

- **Exploitation of Wi-Fi driver vulnerability**: The attackers discovered an out-of-bounds write vulnerability in the Linux Wi-Fi driver's handling of certain shared data structures between the firmware and the driver. By manipulating these structures from their code running in the Wi-Fi firmware, they induced the driver (bcmdhd) to write beyond the intended memory boundaries within the kernel's memory space.
- **Kernel code execution**: The out-of-bounds write allowed the attackers to overwrite data in kernel memory, specifically targeting the kernel stacks of processes they controlled. By overwriting return addresses on the stack, they executed a Return-Oriented Programming (ROP) chain within the kernel context, ultimately achieving arbitrary code execution at the kernel level.
- **Establishing an unrestricted root shell**: With code execution in the kernel, the attackers spawned a root-level process without the restrictions imposed by the original sandbox environment. This provided them with full control over the infotainment system, including the ability to execute arbitrary commands and access sensitive data.

Phase 3: Pivot to the security gateway

Up to this point, a security researcher would be quite happy with the results, but not these guys! This is why they embarked on phase 3 of this attack chain where they attempted to compromise the gateway, as shown in *Figure 8.9*, to reach the vehicle domain side over the CAN bus. From the unrestricted root shell on the infotainment system, the attackers communicated with the Security Gateway over the internal Ethernet network.

The Security Gateway is an MCU (Microcontroller Unit) running a real-time operating system (FreeRTOS) and is responsible for mediating communications between the infotainment system and the vehicle's critical components via CAN buses. The attack techniques that follow should sound familiar as we covered similar attack patterns in the previous chapter on MCU-based ECUs. This demonstrates how in certain cases, an attacker must compromise both the HPC and the MCU-based ECUs to achieve their attack objective.

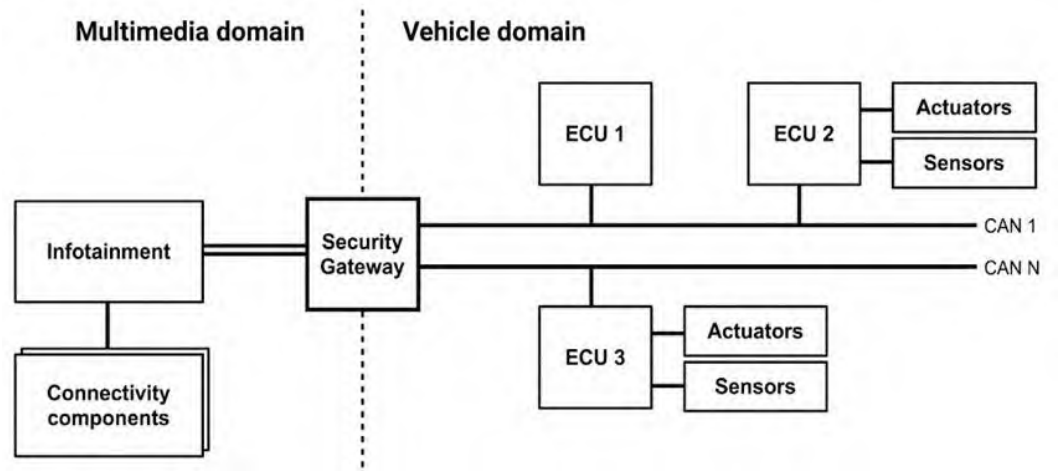


Figure 8.9: The security gateway separates the multimedia domain from the in-vehicle domain where the safety critical functions exist

- **Exploitation of Security Gateway logic flaw**: The attackers exploited a logic flaw in the security gateway's firmware update mechanism. Specifically, they manipulated the update process to bypass signature verification checks. They initiated a legitimate firmware update, which passed the signature verification and scheduled an update. Before the update was applied, they replaced the firmware file on the Security Gateway's SD card with a malicious one. Due to a flaw in the update process, the signature of the already scheduled update was not re-verified before application, allowing the malicious firmware to be loaded.
- **Code execution in Security Gateway**: By uploading a modified bootloader that disabled software-based secure boot mechanisms, the attackers gained code execution within the Security Gateway. They injected their code into the updater binary, leveraging existing functions to interact with the vehicle's CAN buses.
- **Understanding Security Gateway's firmware update mechanism**: The Security Gateway operates in different modes, with the primary ones being the following:
    - **Bootloader mode**: Responsible for initial system startup, software-based secure boot, and transitioning to other modes.
    - **Updater mode**: Used to update firmware for itself and other ECUs in the vehicle. It can fetch firmware files, verify their signatures, and apply updates.
    - **Main application mode**: Handles regular operations, including filtering CAN messages and facilitating communication between the infotainment system and vehicle components.

- **Firmware update workflow**: The typical firmware update process involves the following steps:
    1. **Initiation**: The infotainment system initiates an update by uploading the updater binary to Security Gateway's SD card using the XMODEM protocol
    2. **Reboot for update**: The infotainment system sends a "reboot for update" command via UDP to the Security Gateway
    3. **Verification and execution**: Upon rebooting, the Security Gateway's bootloader verifies the digital signature of the updater binary and, if valid, executes it
    4. **Firmware fetching**: In Updater Mode, the Security Gateway fetches the ECU firmware files from the infotainment system using the Trivial File Transfer Protocol (TFTP)
    5. **Signature verification**: Each fetched firmware file is stored on the SD card and its digital signature is verified
    6. **Application of updates**: If the signature verification passes, the update is scheduled and eventually applied to the respective ECU
- **Exploiting the logic flaw**: The attackers identified a Time-of-Check to Time-of-Use (TOCTOU) vulnerability in the firmware update process:
    - **Time-of-Check**: The Security Gateway verifies the digital signature of the firmware file immediately after fetching it via TFTP from the infotainment system
    - **Time-of-Use**: The firmware file is later used during the update application phase

The logic flaw arises because Security Gateway does not re-verify the signature of the firmware file immediately before applying the update. This allows a window of opportunity where the attackers can replace the legitimate, signed firmware file with a malicious one on the SD card after the initial signature verification but before the update is applied.

Detailed steps of the exploit

Now that we've covered the phases of the attack chain, let's look at how the attackers executed the exploit and the steps they took to achieve the same:

1. **Initiating a legitimate firmware update**: The attackers, now with root access to the infotainment system from previous steps, initiate a firmware update process for an ECU via the Security Gateway. The infotainment system provides a legitimate firmware file with a valid digital signature to the Security Gateway using TFTP.

2. **Signature verification and scheduling**: The Security Gateway fetches the firmware file, stores it on the SD card, and verifies its signature. Upon successful verification, the update is scheduled for application during the next update phase.
3. **Replacing the firmware file**: Before the update is applied, the attackers replace the legitimate firmware file on the SD card with a malicious one. They exploit the fact that the Security Gateway does not protect the SD card's contents against unauthorized modifications from the infotainment system. The attackers use their root privileges on the infotainment system to overwrite the firmware file with their own version, which contains malicious code.
4. **Application of the malicious firmware**: During the update phase, Security Gateway proceeds to apply the firmware update to the ECU using the firmware file present on the SD card. Critically, it does not re-verify the digital signature of the firmware file before application. As a result, malicious firmware is applied to the ECU without detection.
5. **Disabling software-based secure boot**: The attackers target Security Gateway's own firmware, including the bootloader, which is responsible for performing software-based secure boot checks. By using the same method, they upload a modified bootloader that has been altered to disable secure boot mechanisms. This modified bootloader allows the execution of unsigned or malicious code on the Security Gateway.
6. **Gaining code execution on the security gateway**: With the secure boot disabled, the attackers can now execute arbitrary code on the Security Gateway. They inject their code into the updater binary or create a custom updater binary that includes their payload. When the Security Gateway reboots and loads the updater binary from the SD card, it executes the attackers' code.
7. **Leveraging existing functions**: The attackers' injected code reuses existing functions within the Security Gateway's firmware to interact with the vehicle's CAN buses. By calling these functions, they can send arbitrary CAN messages to other ECUs in the vehicle.

Impact

The successful exploitation of these vulnerabilities demonstrated a sophisticated, multi-stage attack chain that traversed from a paired Bluetooth device to kernel-level execution and

ultimately compromised the vehicle's core systems. The implications of this attack chain are significant:

- **Unauthorized control of vehicle functions**: By sending arbitrary CAN messages, an attacker could manipulate critical vehicle systems, potentially affecting safety features and vehicle operation
- **Data privacy risks**: Access to the infotainment system could expose personal data, including contacts, messages, call logs, and navigation history
- **Persistent system compromise**: Modifying firmware components like the bootloader and Security Gateway could allow for a persistent presence within the vehicle's systems, surviving reboots and standard update mechanisms

If at the start of this chapter you had doubts about the technical feasibility of remote attacks impacting vehicle safety, at this point you should have a solid understanding of what is feasible and how skilled attackers chain vulnerabilities to achieve their aims.

Bonus case studies

Now that you have read our case studies, we encourage you to analyze the following set of case studies on your own to understand how the vulnerabilities were discovered, how they were exploited individually, and the process by which they were chained together to achieve the attacker's objective. While doing so, you are encouraged to think about the defensive countermeasures that could eliminate those issues from being exploited in the first place:

Topic	Links
bd-jb: Blu-ray Disc Java Sandbox Escape	`https://www.youtube.com/watch?v=jyyCOoWksbg`
Next-Gen Exploitation: Exploring the PS5 Security Landscape by Specter	`https://www.youtube.com/watch?v=HBFDjfmIUis`
XBOX One Cybersecurity Concept	`https://www.youtube.com/watch?v=U7VwtOrwceo`

Topic	Links
Penetration Testing of In-Vehicle Infotainment Systems in Connected Vehicles	`https://ieeexplore.ieee.org/document/10575976`
Hyundai Head Unit Hacking	`https://docs.google.com/presentation/d/14B_Jrxjz8zHF_HcdfbCyitxiQVvrvYQKZ60USY33rY8/edit#slide=id.p`
SIMuraI: Slicing Through the Complexity of SIM Card Security Research	`https://www.usenix.org/system/files/usenixsecurity24-lisowski.pdf`
BarraCUDA: Edge GPUs do Leak DNNWeights	`https://www.usenix.org/conference/usenixsecurity25/presentation/horvath`

Table 8.1: Additional readings and study materials

Best practices for securing HPCs

Before discussing specific countermeasures, it is important to establish a critical boundary condition. No collection of isolated best practices can meaningfully secure an automotive HPC unless they are driven by a systematic threat analysis and risk assessment process. High-performance computing platforms continuously evolve as new applications, connectivity paths, and integration points are introduced, and each new feature reshapes the attack surface. Treating security as a static checklist often creates a false sense of assurance.

That said, well-established security principles such as defense in depth, least privilege, and least functionality remain highly relevant when they are applied as part of a structured engineering process. Let's look at some of the best practices you can adopt to secure HPCs.

Minimizing the software and interface attack surface

Reducing exposed functionality remains one of the most effective defensive strategies and easiest to implement. Automotive HPC platforms are inherently complex, and every additional service, daemon, protocol stack, and listening port introduces another potential entry point.

As demonstrated in *Case study 1*, the presence of an automatically triggered USB-based update mechanism (activated through the `jci-autoupdate` marker file) materially expanded the pre-authentication attack surface and enabled direct exploitation through removable media. Disabling unattended update triggers and restricting update initiation to authenticated diagnostic workflows would have eliminated this entire exploitation vector.

Defensive engineers should perform systematic audits of the HPC software build and remove any component that is not strictly required for vehicle operation. In practice, this includes eliminating development tools, sample applications, scripting environments, and debugging services from production images.

Least functionality must be paired with least privilege. In *Case study 2*, the proprietary Bluetooth service (btapp) executed with root privileges. Running the Bluetooth stack in a dedicated, unprivileged process with a tightly scoped capability set would have limited the impact of a single memory-safety flaw from becoming an immediate full system compromise.

Services that must remain enabled should run with the minimal set of privileges required to perform their function. Development conveniences such as SSH servers or Android Debug Bridge must not be present in production images unless explicitly required, and even then, must be tightly restricted.

Default credentials and development backdoors represent a particularly high-risk class of exposure. Any credentials used during bring-up or testing must be removed or replaced with device-unique credentials prior to deployment. When the HPC platform is supplied by a third party, OEMs must explicitly validate that vendor backdoors and diagnostic shortcuts are not present in production firmware.

Beyond removing unnecessary software, necessary services must be configured as restrictively as possible. *Case study 2* demonstrates that allowing the Bluetooth stack to expose complex protocol modes and reassembly logic to unauthenticated peers significantly increases attack surface and exploitability. Defensive configuration should explicitly disable optional protocol extensions and experimental features in production builds.

Moreover, if a web service is required for in-vehicle user interfaces, it should not accept remote connections (rather support outbound communication initiation only), and mutual authentication should be enforced when external access is unavoidable. All unnecessary listening ports and network services should be disabled so that even if an attacker reaches the

HPC network interface, no unintended entry points are exposed. Physical and logical interfaces must be treated with the same discipline. USB ports should be disabled or restricted to recognized and authenticated devices, and unused hardware interfaces should be disabled.

Software bill of materials and vulnerability management

Modern HPC platforms depend heavily on third-party software components, particularly in Linux, and Android-based environments. Identifying and eliminating vulnerable software packages remains one of the most effective ways to raise the security bar in HPCs. *Case study 3* demonstrates that vulnerabilities in vendor Wi-Fi firmware and Linux drivers (for example, the bcmdhd driver, and its interaction with shared firmware memory) can be decisive in multi-stage attack chains. SBOMs must therefore include chipset firmware and kernel drivers, not only user-space packages.

All external components and their exact versions must be tracked, and vulnerability scanning should be performed continuously to identify known CVEs. Moreover, patch readiness cannot stop at production release. HPC platforms must be engineered to support in-field updates throughout the operational lifetime of the vehicle. This, in turn, requires long-term support for building environments, signing infrastructure, and updating pipelines. Maintaining the ability to rebuild and patch an HPC platform many years after release is a nontrivial organizational and engineering commitment.

Establishing hardware root of trust and secure boot

Hardware security provides the foundation upon which all higher-level defenses depend. No amount of software defenses can be sufficient if the underlying hardware itself is rife with security weaknesses. At the very base, a tamper-resistant secure boot architecture is mandatory to establish the software chain of trust. Secure boot enforces that chain of trust from the first instruction executed after reset through all bootloader stages and into the operating system and application software. An immutable boot ROM embedded in the SoC acts as the initial Root of Trust and verifies the digital signature of the next boot stage before execution.

Each subsequent stage verifies the integrity and authenticity of the following stage. If any component fails verification, execution must halt. *Case study 1* directly demonstrates the consequence of missing secure boot enforcement on the application SoC. The CMU bootstrap chain did not authenticate boot components, enabling persistent malware installation once storage write access was achieved. A hardware-anchored secure boot chain would have prevented post-exploitation persistence by enforcing integrity of the bootloader and operating system images.

Device Identifier Composition Engine (**DICE**) mechanisms are increasingly preferred for implementing secure boot and identity provisioning. Rather than relying solely on static device

keys, DICE derives device-unique cryptographic identities from measurements of hardware and software state at each stage. This layered identity construction strengthens resistance to cloning and firmware substitution attacks.

Where possible, secure boot should be complemented with remote attestation. Because automotive HPC platforms are typically connected to backend services, backend access should be conditional on verifiable claims about the platform's security state. Attestation limits the ability of a compromised vehicle to interact with backend infrastructure and reduces the risk of large-scale, fleet-wide impact.

Moreover, hardware debug and developer interfaces must be locked down prior to deployment. Open JTAG, UART consoles, or USB debug ports materially simplify firmware extraction and offline analysis. These interfaces should be disabled or protected using strong cryptographic authentication and device-unique credentials.

Using trusted execution environments and isolated security subsystems

Even with a strong secure boot chain and locked hardware interfaces, defenders must assume that software vulnerabilities will eventually allow attackers to execute code within the main operating system. Critical security services and long-term secrets must therefore be isolated from the primary execution environment. Automotive HPC platforms commonly rely on Trusted Execution Environments such as Arm TrustZone to host sensitive services including cryptographic operations, key storage, and attestation agents. Applications running in the normal world can only access these services through narrowly scoped and validated interfaces.

Many automotive SoCs further integrate dedicated security subsystems or Hardware Security Modules. SAE J3101 formalizes this model as a Hardware Protected Security Environment (HPSE). An HPSE typically contains its own processor, secure storage, and firmware and performs security-critical functions such as secure boot validation, signature verification, and key management independently of the main application processor. Isolating secrets in an HPSE significantly reduces exposure in the event of operating-system compromise. Keys stored within the HPSE are not readable through main-CPU memory access, DMA abuse, or debug interfaces.

TEEs and HSMs also enable isolated monitoring and attestation services. Integrity measurement agents running in a protected environment can detect kernel or firmware tampering that would be difficult to conceal from an attacker-controlled operating system. This layered design is a practical realization of defense in depth.

Secure partitioning in hypervisor-based HPC architectures

When a Type-1 hypervisor is used to host multiple vehicle domains on a single HPC platform, the hypervisor becomes a critical security boundary. The hypervisor itself must be minimized and hardened. Only essential virtualization functionality should execute at the highest privilege level. Clearly, optional services, debug features, and management components must be disabled in production builds.

Furthermore, each guest operating system must be strictly isolated in both memory and device access. Hardware virtualization mechanisms and second-stage address translation must prevent one virtual machine from reading or modifying another's memory. Peripheral isolation is equally important as those peripherals should be statically assigned to individual guests.

When device sharing is unavoidable, an IOMMU, referred to as an SMMU in Arm systems, must be enabled and correctly configured. Additionally, DMA protection is essential to prevent a compromised guest or device firmware from accessing memory outside its assigned region.

Hypervisor configurations should be immutable after boot wherever feasible. Guests must not be permitted to dynamically expand memory allocations or gain access to additional peripherals. Isolation barriers should be fully established before any guest is started, and fine-grained hypervisor policies should be configured using the most restrictive settings that still support functional requirements.

This type of strong partitioning ensures that compromise of a less critical domain, such as an infotainment or Android virtual machine, does not trivially enable lateral movement into safety-critical or vehicle control domains.

Memory protection and microarchitectural risk mitigation

Within each guest operating system, standard OS hardening techniques must be applied to prevent a single compromised user process from full take-over of that VM. Typical measures include strict process isolation, non-executable memory regions, and read-only mappings for critical data structures to materially increase the difficulty of exploitation.

Compile-time and runtime hardening features such as Address Space Layout Randomization, **Pointer Authentication Codes** (**PAC**), **Branch Target Identification** (**BTI**), and stack canaries should be enabled to increase the difficulty of control flow attacks. In *Case study 2*, ASLR alone was insufficient because the exploit chain combined memory corruption with an arbitrary read primitive to disclose heap, library, and stack addresses. This illustrates why ASLR must be combined with stronger control-flow and pointer-integrity protections such as PAC and BTI on supported platforms. Moreover, for Linux-based environments, mandatory access control mechanisms such as SELinux and namespace isolation can further restrict resource access for applications and containers.

Moreover, some automotive SoCs now provide transparent memory encryption for external DRAM. Encrypting memory contents at the memory controller mitigates physical attacks such as cold-boot extraction or memory probing and should be leveraged wherever possible.

Defenders must also account for speculative execution attacks where the risk of information leakage is deemed critical enough to incur the additional processing overhead for the corresponding mitigations. In such cases, CPU microcode updates and operating-system mitigations for Spectre and Meltdown-class vulnerabilities must be applied. This includes enabling mechanisms such as Kernel Page Table Isolation and appropriate speculation barriers. When virtualization is used, the hypervisor must ensure that speculative side channels cannot be exploited to escape guest isolation. If performance impact is too severe, then a risk-based assessment must be performed to determine the critical code paths that must be protected. Clearly, performance impact requires careful evaluation, but the guiding principle remains that untrusted code must not be able to infer secrets through shared microarchitectural state.

Strengthening input handling

Going beyond the hardening of the Guest OS, all external and cross-domain inputs must be treated as untrusted. This includes values used to construct file paths, database queries, and operating-system commands. *Case study 1* provides a concrete illustration of this principle. Malformed device identifiers and attacker-controlled archive parameters were passed directly into database operations and shell command construction, enabling SQL injection and multiple command-injection vulnerabilities within privileged update components.

Input handling should be implemented using APIs and libraries that enforce parameterized and bounded inputs to eliminate injection classes of vulnerabilities. Any parsing logic that feeds into command execution or privileged operations must be thoroughly tested to detect code injection vulnerabilities.

Secure in-field updates and supply chain protection

While secure boot verifies software at runtime, secure update mechanisms prevent malicious software from being introduced in the first place. *Case study 3* demonstrates a critical update-workflow weakness in the Security Gateway: firmware images were verified at fetch time but not re-verified at installation time. All update artifacts must be cryptographically authenticated and verified before any update content is processed. Attention must be paid to rollback prevention and mix and match attacks through the usage of cryptographically signed partition version tables that bind software updates together.

The same input validation rigor applies directly to software update mechanisms. The time-of-check to time-of-use flaw exploited in *Case study 3* shows that update security is not only

cryptographic but also transactional. Update state, artifacts, and metadata must be protected against modification between verification and execution.

Moreover, pull requests of OTA updates remain the preferred method as they eliminate the possibility of a malicious server pushing tampered software updates. In fact, the update pipeline must enforce integrity checks at every stage, and no command derived from update content should be executed unless authenticity and integrity have already been established. In *Case study 3*, the infotainment system effectively acted as a privileged update source for the Security Gateway over internal networks. Accounting for all update masters is crucial as we cannot assume an onboard updater has a higher level of trust simply because it initiates updates from within the vehicle. In fact, update delivery must enforce mutual authentication and authorization between domains regardless of their physical placement so that a compromised multimedia domain cannot act as a trusted update origin. Moreover, fail-safe update strategies, such as dual or redundant boot chains, are required to ensure that update failures do not render the HPC inoperable.

When it comes to supply chain security, it must be considered equally to hardware and software. Components must be sourced from trusted suppliers with controlled manufacturing and distribution processes. Software delivered by suppliers and partners must be verified for integrity and provenance using cryptographic mechanisms. Moreover, protecting signing infrastructure and strictly controlling signing keys is mandatory. Essentially, every artifact that enters the vehicle software ecosystem must be verifiable and traceable.

Summary

HPCs embody the convergence of critical vehicle functionality, rich user experiences, and cloud-integrated services into a single computational platform. As we have explored throughout this chapter, the architectural decisions that make HPCs so powerful also make them fundamentally harder to secure. With their wide range of hardware interfaces, reliance on commodity operating systems, and need to host multiple software domains under varying safety and security constraints, HPCs present a vastly expanded attack surface.

Through detailed analysis of their hardware components and layered software stacks, we have seen how weaknesses in both areas are not only common but often interdependent. For example, a poorly protected JTAG port or insecure bootloader can serve as the launchpad for privilege escalation, lateral movement, and even full vehicle compromise. Software vulnerabilities, from kernel CVEs to overprivileged services and weak hypervisor partitioning, are equally potent when combined with poor hardware-enforced isolation or default configurations.

Case studies, such as those involving Mazda and Tesla, provide real-world validation of the methods and weaknesses discussed. Each showed that seemingly minor oversights such as an

unsigned firmware path, a misconfigured debug interface, or insufficient input sanitization can be chained into sophisticated, persistent, and high-impact intrusions.

By adopting the mindset of an attacker, we examined the phases of a successful compromise, from initial access to persistence. For offensive security engineers, this should serve as the baseline for understanding and building sophisticated attack chains against real-life HPCs.

On the defensive side, while stressing the need for a systematic threat analysis-driven approach, we have outlined best practices for designing and maintaining secure HPCs. These practices include enforcing secure boot with hardware roots of trust, leveraging TrustZone or HPSEs for cryptographic material isolation, employing hardened hypervisors for mixed-criticality separation, reducing attack surfaces through least privilege, and embedding security into the software development lifecycle. Importantly, these defenses must work as defense-in-depth layers, each providing a layer of security to reduce the likelihood of a damaging breach.

Perhaps the most sobering insight from this chapter is the realization that attackers often succeed not because security mechanisms are absent, but because they are incomplete, misconfigured, or inconsistently applied. As we conclude this chapter, we understand that protecting an HPC means securing a platform that rivals enterprise servers in complexity and threat exposure, only with the added challenges of physical access, long lifetime support, and functional safety integration. This requires cross-disciplinary thinking, continuous testing, and above all, a deep understanding of cutting-edge attack techniques, so we are not blindsided by a defensive-only mindset.

References

[1] QNX Software Systems, "*QNX Hypervisor Empowers Software-Defined Vehicles with VirtIO*," Electronic Design. [Online]. Available: `https://www.electronicdesign.com/markets/automotive/video/21285003/electronic-design-qnx-hypervisor-empowers-software-defined-vehicles-with-virtio`

[2] S. Skorobogatov, Semi-Invasive Attacks: A New Approach to Hardware Security Analysis, University of Cambridge, Computer Laboratory, PhD Thesis, 2005.

[3] Keysight Technologies, "*W4641A DDR4 x16 2-wing BGA Interposer for Logic Analyzers*," Keysight Product Page. [Online]. Available: `https://www.keysight.com/us/en/product/w4641a/ddr4-x16-2-wing-bga-interposer-logic-analyzers.html`

[4] A. T. Markettos et al., "*Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrusted Peripherals*," in Proceedings of the 28th USENIX Security Symposium, 2019.

[5] Thyrasec, "*Extracting Firmware from EEPROM, Flash and eMMC*," Thyrasec Blog. [Online]. Available: `https://www.thyrasec.com/blog/extracting-firmware-from-eeprom-flash-and-emmc/`

[6] Arm Ltd., Arm Architecture Reference Manual for A-profile architecture (ARMv8-A), Chapter: Memory Management and Stage 2 Translation, Arm Ltd.

[7] G. Heiser and B. Leslie, "*The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors*," in Proceedings of the Asia-Pacific Computer Systems Architecture Conference, 2010.

[8] V. P. Ladakis et al., "*You Can Type, but You Can't Hide: A Stealthy GPU-Based Keylogger*," EuroSys 2013.

[9] Zhu, Zhiting, et al. "*Understanding the security of discrete GPUs*." Proceedings of the General Purpose GPUs. 2017. 1-11.

[10] P. Kocher, J. Jaffe, and B. Jun, "*Differential Power Analysis*," in Proceedings of CRYPTO '99, 1999.

[11] S. Mangard, E. Oswald, and T. Popp, Power Analysis Attacks: Revealing the Secrets of Smart Cards, Springer, 2007.

[12] K. Gandolfi, C. Mourtel, and F. Olivier, "*Electromagnetic Analysis: Concrete Results*," in Proceedings of CHES 2001, 2001.

[13] NIST, SP 800-57 Part 1 Rev. 5, Recommendation for Key Management - Part 1: General, National Institute of Standards and Technology, 2020.

[14] D. J. Bernstein, "*Cache-Timing Attacks on AES*," Technical Report, 2005.

[15] P. Kocher, "*Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*," in Proceedings of CRYPTO '96, 1996.

[16] MITRE EMB3D, "*MID-030: Firmware Rollback Protections*," MITRE EMB3D. [Online]. Available: `https://emb3d.mitre.org/mitigations/MID-030.html`

[17] Kocher, Paul, et al. "*Spectre attacks: Exploiting speculative execution*." Communications of the ACM 63.7 (2020): 93-101.

[18] C. Canella et al., A Systematic Evaluation of Transient Execution Attacks and Defenses, USENIX Security 2019.

[19] A. Liljestrand et al., PACMAN: Attacking ARM Pointer Authentication with Speculative Execution, ISCA 2022.

[20] Y. Yarom and K. Falkner, Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack, USENIX Security 2014.

[21] M. Schwarz et al., NetSpectre: Read Arbitrary Memory over Network, IEEE S&P 2019.

[22] Ghaniyoun, Moein, et al. "*Teesec: Pre-silicon vulnerability discovery for trusted execution environments*." Proceedings of the 50th Annual International Symposium on Computer Architecture. 2023.

[23] M. Lipp et al., "*Meltdown: Reading Kernel Memory from User Space*," in Proceedings of the 27th USENIX Security Symposium, Aug. 2018, pp. 973-990. [Online]. Available: `https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf`

[24] C. Fanjas, C. Gaine, D. Aboulkassimi, S. Pontie, and O. Potin, "*Combined Fault Injection and Real-Time Side-Channel Analysis for Android Secure-Boot Bypassing*," Cryptology ePrint Archive, Paper 2022/602, 2022. [Online]. Available: `https://eprint.iacr.org/2022/602`

[25] Gens, David. OS-level Attacks and Defenses: from Software to Hardware-based Exploits. Diss. 2019. [26] Carvalho, Paulo F., et al. "*PCI express hotplug implementation for ATCA based instrumentation.*" Fusion Engineering and Design 96 (2015): 738-741.

[27] ARM Ltd., System Memory Management Unit Architecture Specification (SMMU), ARM Architecture Reference Manual. [Online]. Available: `https://developer.arm.com/documentation/ihi0070/ha/`

[28] ARM Ltd., Secure Boot and Root of Trust, ARM Platform Security Architecture. Online. Available: `https://developer.arm.com/documentation/101892/0100/Security---Platform-Security-Architecture`

[29] ISO/SAE 21434:2021(en) Road vehicles — Cybersecurity engineering.

[30] D. Goodin, "*VMware plugs three hypervisor-hijack holes*," The Register, Mar. 4, 2025. [Online]. Available: `https://www.theregister.com/2025/03/04/vmware_plugs_three_hypervisorhijack_holes/`

[31] Qualcomm Technologies Inc., "*February 2025 Security Bulletin*," Qualcomm Security Center, Feb. 2025. [Online]. Available: `https://docs.qualcomm.com/product/publicresources/securitybulletin/february-2025-bulletin.html`

[32] IBM Corporation, "*Security Bulletin: The PowerVM hypervisor is vulnerable to a specially crafted sequence of hypervisor calls from a partition that can lead to a system crash*," IBM Support, 2021. [Online]. Available: `https://www.ibm.com/support/pages/security-bulletin-powervm-hypervisor-vulnerable-specially-crafted-sequence-hypervisor-calls-partition-can-lead-system-crash`

[33] QNX Software Systems, "*Memory sharing*," QNX Hypervisor User's Guide. [Online]. Available: https://www.qnx.com/developers/docs/8.0/com.qnx.doc.hypervisor.user/topic/share/share_mem.html

[34] Red Hat, "*Understanding and Mitigating the Dirty COW Vulnerability*," Red Hat Blog, Oct. 2016. [Online]. Available: https://www.redhat.com/en/blog/understanding-and-mitigating-dirty-cow-vulnerability

[35] L. Abrams, "*New Linux bug gives root on all major distros, exploit released*," BleepingComputer, Mar. 2022. [Online]. Available: https://www.bleepingcomputer.com/news/security/new-linux-bug-gives-root-on-all-major-distros-exploit-released/

[36] PlaxidityX, "*High Profile Linux Vulnerability May Make Vehicles Susceptible to Cyber Attack*," PlaxidityX Cybersecurity Blog, Jan. 2021. [Online]. Available: https://plaxidityx.com/blog/cyber-security-blog/high-profile-linux-vulnerability

[37] National Institute of Standards and Technology (NIST), "*CVE-2019-14100*," National Vulnerability Database, 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-14100

[38] National Institute of Standards and Technology (NIST), "*CVE-2019-2215*," National Vulnerability Database, Oct. 2019. [Online]. Available:https://nvd.nist.gov/vuln/detail/CVE-2019-2215

[39] Renesas Electronics, "*HW/SW Security Mechanisms for Future Automotive Society*," Whitepaper, 2023. [Online]. Available: https://www.renesas.com/en/document/whp/hwsw-security-mechanisms-future-automotive-society

[40] JTAGulator. [Online]. Available: https://expliot.io/jtagulator/

[41] Spiceworks, "*Supply Chain Attacks: Why Risk Management and Business Continuity Planning Are Essential*," Spiceworks Articles, Jan. 2024. [Online]. Available: https://www.spiceworks.com/it-security/cyber-risk-management/articles/supply-chain-attacks-why-risk-management-and-business-continuity-planning-are-essential/

[42] Intel, "*Intel Debug Technology*," Intel Security Guidance, updated Jul. 28, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/intel-debug-technology.html

[43] MITRE, "*CWE-1310: Missing Ability to Patch ROM Code*," Common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/data/definitions/1310.html

[44] J. Van den Herrewegen, D. Oswald, F. D. Garcia, and Q. Temeiza, "*Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis*," IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2021, no. 1, pp. 56-81, 2021. [Online]. Available: `https://doi.org/10.46586/tches.v2021.i1.56-81`

[45] Uptane, "*Threat Model*," Uptane Documentation. [Online]. Available: `https://uptane.org/learn-more/threat`. [Accessed: May 16, 2026].

[46] MITRE, "*CWE Top 25 Most Dangerous Software Weaknesses*," Common Weakness Enumeration (CWE), 2025. [Online]. Available: `https://cwe.mitre.org/top25/`. [Accessed: May 16, 2026].

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

9

Attacking Vehicle Sensors and AI/ML

Connected vehicles depend on a large number of sensors enabling real-time awareness of both vehicle conditions and the surrounding environment. Modern vehicles commonly integrate dozens of sensors, with higher-end platforms exceeding 100 sensors as more advanced features are introduced. These sensors span every domain of vehicle functionality. Powertrain and engine control units rely on deeply embedded sensors, such as temperature, pressure, and airflow sensors, to optimize performance, efficiency, and emissions. Body systems and safety features use sensors for everything from airbag deployment to cabin comfort. **Advanced Driver Assistance Systems** (**ADAS**) introduce a distinct set of external perception sensors, including cameras, radar, LiDAR, and ultrasonic sensors, to detect obstacles, lanes, and traffic signs across the various levels of driving automation. Beyond traditional sensors, modern vehicles increasingly integrate **artificial intelligence** (**AI**) / **machine learning** (**ML**)-based systems, including **large language model** (**LLM**)-based in-vehicle assistants that process voice commands and external data sources to provide navigation, communication, and vehicle control capabilities [1].

Because sensors and AI systems are foundational to vehicle safety and functionality, any compromise or manipulation of their inputs or outputs can have serious consequences. A spoofed ADAS sensor could cause the vehicle to misjudge obstacles or road conditions, while a compromised in-vehicle AI assistant could be induced to invoke vehicle functions without authorization from the legitimate user. As vehicles move toward higher levels of autonomy, the integrity of sensor inputs and AI system behavior becomes increasingly critical.

This chapter explores vehicle sensors and AI systems from an offensive security perspective. We examine the architectures of sensor types found in modern vehicles, analyze the weaknesses that arise from their physical properties, protocol design, and integration into the

vehicle, and discuss penetration testing techniques. Further, we present three case studies to illustrate these concepts in practice. The chapter concludes with best practices and defensive guidance to reduce the sensor attack surface, ensuring these vital components are robust against adversaries.

In this chapter, we will cover the following main topics:

- Overview of vehicle sensors and AI
- Understanding vehicle sensors and AI interfaces
- Common weaknesses of vehicle sensors and AI systems
- Penetration testing techniques against vehicle sensors and AI
- Case studies of real-life attacks on vehicle sensors and AI/ML systems
- Best practices for securing vehicle sensors and AI systems

Overview of vehicle sensors and AI

Vehicle sensors span a wide range of complexities. Some sensors are simple transducers electrically terminated at a nearby **Electronic Control Unit** (**ECU**), while others are full modules with onboard processors, storage, and high-bandwidth data links. A well-equipped passenger vehicle may carry dozens of individual sensing elements spanning radar, camera, LiDAR, ultrasonic, and pressure-based technologies, each contributing to a different layer of environmental awareness. Beyond physical perception, vehicles increasingly incorporate AI-driven voice assistants and in-cabin intelligence that process natural language commands, personalize the driving experience, and communicate with cloud services in real time. Each of these capabilities expand the vehicle's functionality, but at the same time expands the attack surface of the vehicle [2].

Regardless of complexity, when analyzing the security of vehicle sensors, the architectural question is the same: *where does sensing occur*, *where is the data processed*, *and how is that data transported through the E/E architecture to the control functions that consume it?* While ECUs are typically the most common targets for penetration testing, sensor makers and OEMs have been engaging penetration testers in recent years both to identify sensor vulnerabilities before pushing their product to the market and to uncover ways the rest of the vehicle can be impacted if one or more sensors are compromised. It is important to note that vehicle sensors and AI systems introduce unique security challenges beyond what we observe in traditional in-vehicle network protocols and ECUs. Sensor attacks typically operate at the physical layer, using electromagnetic signals, light, or sound to inject false data into systems that were not designed with adversarial inputs in mind. AI attacks target the inference process itself, corrupting or misleading models that drive automated decisions. Together, these attack

categories represent one of the most rapidly evolving frontiers in automotive cybersecurity [2,3].

Before diving into sensor penetration testing methodologies, it is important to enumerate the primary sensor types and AI interfaces encountered in modern connected and automated vehicles, along with their typical hardware and software anatomy and their network connectivity patterns.

Understanding vehicle sensors and AI interfaces

Sensors and AI systems in a modern vehicle encompass a wide range of technologies that differ substantially in their physical operating principles, their connectivity characteristics, and the nature of the attack surface they present. To better understand the type of sensors and AI interfaces that exist in vehicles, it is useful to categorize them according to their architectural integration into the vehicle. We use four categories to distinguish between them: **ECU-attached sensors**, **network-attached sensors**, **externally mounted sensors**, and **AI and ML systems**.

The first two categories, **ECU-attached sensors** and **network-attached sensors**, have attack surfaces that overlap with ECUs and in-vehicle communication protocols. Attacks against these sensor types primarily target ECU firmware, diagnostic interfaces, or in-vehicle network-level message injection and replay attacks. These attack surfaces and the penetration testing techniques applicable to them are covered in the chapters addressing ECU security and in-vehicle network security, respectively. Rather than repeating the study of these attack surfaces, this chapter briefly covers **ECU-attached** and **network-attached sensors** for completeness, pointing to the dedicated coverage in earlier chapters where applicable, before focusing in depth on the two remaining categories, **externally mounted sensors** and **AI and ML systems**, which introduce attack surfaces that are substantially different from those of traditional ECUs and networked components. Externally mounted sensors introduce attack surfaces in the physical signal domain that require specialized RF, optical, or acoustic equipment and knowledge to assess. AI and ML systems extend the attack surface to the inference pipeline and the model supply chain, and, in the case of LLM-based in-vehicle assistants, to the natural language inputs and external data sources the system is authorized to process. An overview of common vehicle sensor classes in a **Software-Defined Vehicle** (**SDV**) is depicted in *Figure 9.1*:

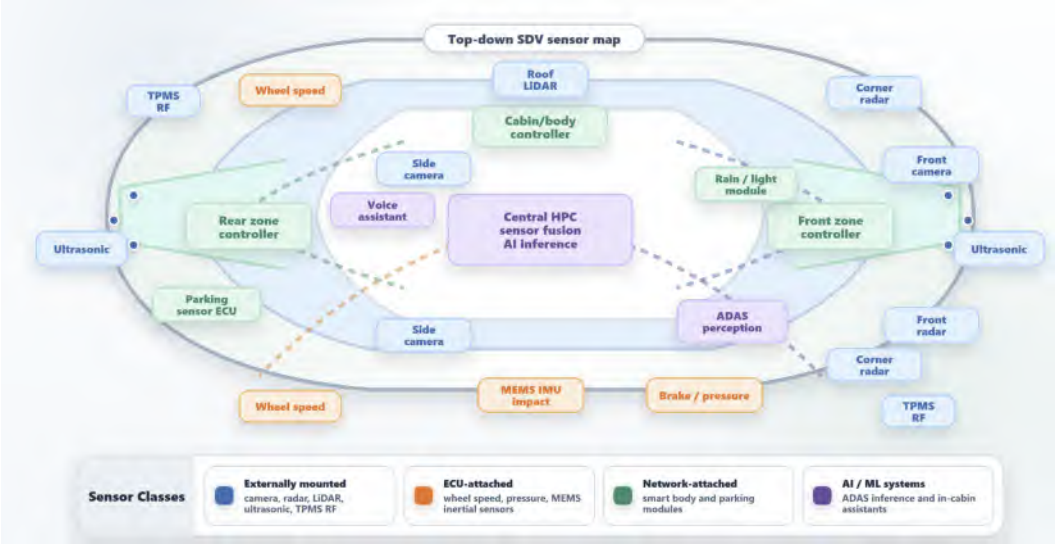


Figure 9.1: Vehicle sensor classes in a typical SDV

Let's move on to the next type of sensors.

ECU-attached and network-attached sensors

ECU-attached sensors are physically integrated into an electronic control unit or directly connected to a dedicated sensor processing module. Examples include wheel speed sensors, throttle position sensors, steering angle sensors, and airbag impact sensors. The ECU controls the hardware interface to the sensor and processes the raw signal internally before the result is made available to the vehicle network. Because the sensor data is produced and consumed within the ECU's own processing boundary, attacks against ECU-attached sensors generally proceed through the ECU itself, targeting the firmware through diagnostic interfaces, exploiting memory safety weaknesses in the signal processing code, or manipulating the physical sensor signal at the connector level. These attack categories are addressed in the context of ECU security in *Chapters 7* and *8*.

One category of ECU-attached sensor warrants specific coverage in this chapter: **Microelectromechanical Systems** (**MEMS**) inertial sensors, including accelerometers and gyroscopes. These sensors are used in functions such as electronic stability control, airbag deployment, and inertial navigation. Although their digital output is processed within the ECU, they are susceptible to manipulation through external physical stimuli directed at the sensor itself, which places them in the same threat class as the externally facing sensors covered in the sections that follow.

Network-attached sensors are standalone modules that produce processed measurements and send them over an in-vehicle network bus. Examples include parking distance sensors and body sensors such as rain and light detection modules, which transmit measurement data over **Local Interconnect Network** (**LIN**) to a single controlling ECU such as a body control module, and sensors such as wheel speed and steering angle sensors that broadcast their signals over **Controller Area Network** (**CAN**) to multiple ECUs simultaneously. From an offensive security perspective, a network-attached sensor is a network node that can be analyzed for authentication weaknesses in its message communication, for diagnostic surface exposure, and for susceptibility to denial-of-service or replay attacks on the bus. These considerations are directly similar to those applicable to any other node on the same bus and are covered in the in-vehicle network communication security in *Chapter 6*.

Externally mounted sensors

Externally mounted sensors are distinguished by the fact that their primary input comes from the physical environment outside the vehicle boundary. Many of the externally mounted sensors covered in this chapter are primary inputs to ADAS functions such as automatic emergency breaking, adaptive cruise control, and lane keeping assistance. The following subsections describe the architecture and interfaces of the externally mounted sensor types that are in scope for this chapter.

Tire pressure monitoring systems

Tire Pressure Monitoring Systems (**TPMS**) consist of a transmitter unit mounted in each wheel and a receiver module integrated into the vehicle body. Each transmitter is a low-power, battery-operated device that measures tire pressure and temperature and transmits periodic status packets using a short-range **Radio Frequency** (**RF**) signal, typically operating in the 315 MHz band in North America and the 433.92 MHz **Industrial, Scientific and Medical** (**ISM**) band in Europe. The receiver decodes incoming transmissions and forwards pressure and temperature data to a gateway or body control module over an internal CAN bus.

TPMS transmitters are identified by a unique sensor ID that the receiver learns during an enrollment or pairing procedure, initiated during tire replacement or after a wheel rotation. The sensor ID is transmitted in cleartext within the packet structure and is not cryptographically authenticated in the majority of deployed systems. The receiver relies on sensor ID matching and, in some implementations, received signal strength thresholds as its primary criteria for accepting incoming data. The one-way communication architecture means the transmitter has no knowledge of whether its packets have been received, and the receiver has no mechanism to challenge or verify the transmitter's identity.

The communication from the TPMS receiver to the body control module or gateway is typically over an in-vehicle bus, such as CAN or LIN. At this stage, the TPMS data has already been

decoded from the RF transmission and is treated as any other message on the bus. Further, the security of the internal bus communication is similar to that of other CAN-attached sensors, including the use or lack of message authentication at the bus level.

Camera systems

Camera-based perception systems typically involve multiple image sensors distributed around the vehicle perimeter, covering forward, backward, and lateral fields of view. Raw image data is transmitted from each camera module to a central processing unit over a high-bandwidth Serializer-Deserializer (SerDes) link. **Gigabit Multimedia Serial Link (GMSL)** and **Flat Panel Display Link (FPD-Link)** are the two dominant proprietary standards used in automotive camera systems.

The processing of camera data varies by system architecture. Some designs forward raw frames, or **Image Signal Processor (ISP)**-processed frames, to a centralized **High-Performance Computer (HPC)** for inference, while others perform object detection and lane recognition locally within the camera module or a dedicated ISP ECU and publish structured outputs such as object lists and lane markers over the vehicle network. The distinction is relevant from a security perspective because the attack surface for an adversarial image injection at the SerDes level differs from the attack surface for a crafted structured output injected at the in-vehicle network level.

The control channel for camera systems, which carries synchronization, exposure, and configuration messages, typically operates over **Inter-Integrated Circuit (I2C)** alongside or embedded within the SerDes link. This control path operates at a substantially lower bandwidth than the video data path.

Note

Terms:

- Image Signal Processor (ISP) is a processor that converts raw camera sensor output into usable image frames or structured metadata through operations such as noise reduction, white balance, and gamma encoding. The ISP may be embedded in the camera module or implemented as a separate ECU.
- Serializer-Deserializer (SerDes) converts parallel data to a serial stream for high-speed point-to-point transmission. In automotive cameras, proprietary standards such as GMSL and FPD-Link use SerDes links to carry video from camera modules to processing ECUs, often embedding a control channel on the same physical cable.

Radar systems

Automotive radar systems operate in the 24 GHz or 77 to 81 GHz millimeter-wave frequency range, with the higher band now dominant for long-range and medium-range detection owing to its wider available bandwidth and smaller antenna form factor. The radar module emits a **Frequency-Modulated Continuous Wave** (**FMCW**) signal, and it computes the range, velocity, and lateral angle of detected objects from the received reflections, producing an object list or a **range-Doppler map** that is forwarded to the vehicle's sensor fusion layer.

Radar modules communicate their outputs to the fusion layer over CAN, FlexRay, or Automotive Ethernet depending on the data rate requirements of the application. Long-range front radar systems, which must deliver object lists at high update rates for ADAS functions such as adaptive cruise control and automatic emergency braking, increasingly use Automotive Ethernet. Short-range corner radars used for blind-spot monitoring and cross-traffic detection may use CAN given the lower bandwidth requirements of those applications.

Note

Terms:

- Range-Doppler map is a two-dimensional radar processing output that plots detected objects by both range and Doppler velocity, serving as the basis for Constant False Alarm Rate (CFAR) detection and object classification.
- Constant False Alarm Rate (CFAR) is an adaptive radar detection algorithm that continuously adjusts its threshold relative to the local noise floor, maintaining a consistent false alarm rate across varying interference and clutter conditions.

LiDAR systems

LiDAR systems use pulsed or modulated laser light, typically at 905 nm for time-of-flight systems or 1550 nm for systems designed for higher output power at reduced eye-safety risk, to measure the range of objects in the environment by timing the interval between the outgoing pulse and the returning reflection. The result is a three-dimensional **point cloud** at update rates typically between 10 and 20 Hz for rotating multi-channel sensors or higher for solid-state systems. LiDAR point clouds are computationally intensive to process and are typically forwarded to an HPC over Automotive Ethernet. The physical receiver of a LiDAR system is an array of **Single-Photon Avalanche Diode** (**SPAD**) or **Avalanche Photodiode** (**APD**) elements that respond to incident photons at the sensor's operating wavelength, with no built-in ability to distinguish the source of those photons from the external environment.

There are two main hardware architectures in use. **Rotating mechanical sensors** use a spinning mirror or mirror assembly to sweep the laser beam across a wide horizontal field of view, producing a scan pattern that is deterministic and predictable in both angle and timing. **Solid-state sensors** use electronically steered beams with no moving parts, which are increasingly common in series production vehicles due to their improved reliability and lower profile. Both designs forward their point cloud data to an HPC over Automotive Ethernet, and both typically use a separate lower-bandwidth interface, such as CAN, for configuration, health monitoring, and firmware update management.

Note

Terms:

- Single-Photon Avalanche Diode (SPAD) is a photodetector that operates above its breakdown voltage so that a single incident photon triggers a measurable avalanche current, enabling the picosecond-precision photon timestamping used for time-of-flight distance measurement in LiDAR receivers.
- Avalanche Photodiode (APD) is a photodetector that amplifies incoming light through carrier multiplication under applied reverse bias, producing an analog output proportional to incident intensity. It is used in LiDAR receivers where high sensitivity to returning laser pulses is required.
- Point cloud is a three-dimensional dataset of discrete spatial points produced by a LiDAR sensor, each representing a surface location from which a laser pulse reflected, and is used as input to object detection, segmentation, and environment mapping.

Ultrasonic systems

Ultrasonic sensor systems operate at close range, typically up to five meters for parking applications, using sound pulses in the 40 to 58 kHz range. The sensor emits a pulse burst and measures the time of flight of the reflected echo to compute the distance to the nearest obstacle. Some implementations apply echo gating, which restricts the receiver to accepting only echoes that arrive within a timing window consistent with a physically plausible reflection distance, rejecting any signals that arrive outside this window. A typical vehicle contains between eight and twelve ultrasonic sensors distributed around the front and rear bumpers, with each sensor connected to a parking assistance ECU over LIN, CAN, or through a simple **Pulse-Width Modulated (PWM)** analog signal. Because ultrasonic sensors operate at distances well below the minimum detection range of radar and LiDAR, they are the primary

sensor type used for close-range proximity detection in low-speed parking and automated maneuver assistance.

Architecture of externally mounted sensors

From an architectural standpoint, externally mounted sensors typically follow one of two deployment models. The first model consists of front-end sensors that stream largely raw data to a centralized ECU, domain controller, or HPC for processing. A common example is a camera module that transmits pixel streams over automotive SerDes to an image signal processor or perception **System-on-Chip** (**SoC**). In such designs, the sensor front end contains minimal preprocessing logic, and the bulk of perception computation occurs in centralized compute.

The second model consists of smart sensors that perform substantial onboard processing and publish higher-level outputs. Radar and LiDAR modules frequently fall into this category, executing signal processing pipelines internally to generate object lists, tracks, or point clouds. Some advanced camera systems also incorporate onboard image processing and publish structured metadata in addition to or instead of raw frames. Ultrasonic systems often compute time-of-flight and distance measurements within a local control module before publishing results upstream.

In practice, modern platforms mix both models. Raw image streams may coexist with radar object tracks, LiDAR point clouds, and ultrasonic distance measurements, all converging at domain or zonal controllers for fusion.

Connectivity reflects these bandwidth and processing differences and is usually divided into a **data plane** and a **control plane**. The data plane transports measurement data and may include:

- SerDes links carrying camera payloads
- Automotive Ethernet interconnecting radar, LiDAR, and other high-bandwidth sensors to domain or zonal controllers

On the other hand, the control plane manages configuration, synchronization, calibration, and health reporting. It commonly relies on CAN, LIN, I2C, or vendor-specific side channels, even when the primary data path uses Ethernet or SerDes.

From a hardware and software perspective, many externally mounted sensors resemble compact ECUs. They frequently integrate:

- A **microcontroller unit** (**MCU**) or SoC
- Dedicated memory and storage

- High-speed serializers or network controllers
- Firmware responsible for sensor control, calibration, timing alignment, signal preprocessing, and data publication

AI and ML systems

The fourth architectural category is **AI and ML systems**, which encompasses the software systems that consume sensor data or other input and perform inference, planning, or interaction tasks using machine learning models. Two distinct subsystems fall within this category: the **ADAS perception and planning** stack, and **in-vehicle AI assistants**.

ADAS perception and planning

The most safety-critical AI applications are found in perception and planning stacks of ADAS and automated driving platforms. The ADAS perception stack takes sensor inputs from cameras, radar, LiDAR, and other sensing technologies and produces a model of the vehicle's environment that feeds the planning and control layers. Perception tasks performed by this stack include object detection and classification, pedestrian and cyclist recognition, lane detection, traffic sign and signal recognition, and free-space estimation. These tasks are performed by **Deep Neural Network** (**DNN**) models, typically **Convolutional Neural Networks** (**CNNs**), which is a class of architecture designed to extract spatial features from image data, or transformer-based architectures, which are trained on large, annotated datasets. The outputs of the perception stack directly influence vehicle control decisions, which makes the integrity of these models a safety-critical property. Importantly, a neural network model is not simply software in the conventional sense. It is a function approximator whose behavior is determined by the statistical properties of its training data, and whose outputs can be manipulated by carefully constructed inputs that fall outside the distribution it was trained on.

Moreover, the ADAS stack interfaces with the vehicle's sensor hardware through a defined data pipeline: raw or pre-processed sensor data received at the HPC over Ethernet is ingested by the perception model, and the resulting structured outputs are published to the control and planning systems through well-defined **Application Programming Interfaces** (**APIs**) or middleware layers. In many implementations, the inference pipeline is isolated within a dedicated execution environment on the HPC, with access to vehicle control functions controlled by a safety manager that enforces operational design domain boundaries.

In-vehicle AI assistants

A second category of AI adoption is the in-vehicle AI assistant, a natural language interface that allows the driver and passengers to control vehicle functions through voice commands supporting functions such as navigation control, media and climate management, phone call handling, and general information queries. These systems are typically built on LLM

infrastructure, either locally hosted on the vehicle or accessed remotely through cloud APIs. The assistant receives spoken input via the vehicle's microphone system, processes it through a speech recognition layer and then through the language model, and produces a response or executes a vehicle function through a defined API. In addition to voice input, the assistant may ingest content from external data sources such as navigation point of interest descriptions, incoming messages, and connected service API responses, all of which form part of the context the language model processes when generating a response or invoking a function.

The boundary between the AI assistant and the rest of the vehicle is defined by the assistant's permission model, which determines which functions the model may invoke and under what conditions. This boundary is a critical trust enforcement point. If the assistant's permissions are overly broad, or if the permission model can be bypassed through carefully crafted inputs, an attacker who can influence the assistant's inputs can indirectly invoke vehicle functions without the driver's or passengers' permission.

While the increasing reliance on vehicle sensors and AI-driven systems has significantly improved perception, automation, and driver assistance capabilities, these same technologies also introduce new security and reliability challenges. The following section examines the common weaknesses and attack surfaces associated with modern vehicle sensing and AI systems.

Common weaknesses of vehicle sensors and AI systems

The weaknesses described in this section are rooted in a combination of fundamental physical properties of the sensing technologies, protocol-level design decisions made during standardization or commercialization, and architectural choices made during vehicle-level integration. Because these sensors draw their primary input from the physical environment outside the vehicle boundary, the attack surface they present is qualitatively different from that of internally networked components: an attacker does not need access to the vehicle's electronics to influence what they perceive. By manipulating the electromagnetic, optical, or acoustic signals in the sensor's operating environment, an attacker can affect the data the sensor delivers to the rest of the vehicle without establishing any connection to the vehicle's internal network. In many cases these weaknesses do not arise from implementation errors but from design trade-offs that were considered acceptable when the threat landscape was less well understood. Understanding the root cause of each weakness is important for a penetration tester because it determines whether the weakness is exploitable under realistic conditions and what the likely impact would be, particularly given that a successful manipulation of what these sensors perceive could affect safety-critical vehicle behavior.

Acoustic signal injection weaknesses in MEMS inertial sensors

MEMS inertial sensors are susceptible to a physical-layer attack known as resonant acoustic injection. These sensors transduce physical motion into electrical signals through microscale mechanical structures, and the analog front end of a capacitive MEMS sensor can be coupled to by precisely tuned acoustic tones directed at the sensor from an external source. Under certain conditions, the interaction between the mechanical resonance frequency and the sensor's analog-to-digital conversion path causes the injected acoustic energy to produce shaped digital readings rather than noise, allowing an attacker to induce controlled deviations in the sensor output without any access to the vehicle network or ECU firmware. The feasibility of this attack depends on sensor placement, enclosure design, signal conditioning, and whether the consuming ECU applies plausibility checks to the sensor output, as the absence of such checks allows acoustically induced deviations to propagate directly into safety-relevant functions.

Unauthenticated RF transmission and vehicle tracking in TPMS

The most significant weakness of deployed TPMS systems is the lack of message authentication. TPMS transmitters communicate using one-way radio transmissions with no shared secret between the transmitter and the receiver, and no cryptographic binding between the sensor ID and the data payload. The packet structure, which typically includes sensor ID, pressure value, temperature reading, battery status, and a **Cyclic Redundancy Check** (**CRC**), can be easily decoded from a passive radio capture because the modulation schemes used are well documented in open literature and in publicly available decoder implementations. The CRC is used to detect transmission errors rather than to authenticate the source, and the receiver has no mechanism to distinguish a legitimate transmission from one crafted by an attacker-controlled device capable of transmitting at the appropriate frequency with a correctly formatted payload.

The regulatory requirement that TPMS alerts the driver within a specified time after a genuine tire pressure loss makes it susceptible to accepting incoming data with minimal filtering. That is, receivers that apply sophisticated plausibility checks risk exceeding the regulatory alert latency threshold, which reinforces the tendency to accept packets with matching sensor IDs without further validation.

Moreover, published academic work [4] demonstrated as early as 2010 that passive monitoring of TPMS signals from a stationary roadside position is sufficient to recover sensor IDs and track individual vehicles at ranges of several meters using inexpensive radio hardware, and that

replay of captured packets against target vehicles produces false pressure warning alerts in production systems.

Specific weaknesses arising from the lack of RF signal authentication in TPMS include the following:

- **Cleartext sensor identifier transmission enabling passive replay**: Each transmitted packet includes the sensor ID in an unencrypted, unauthenticated field. An attacker that can demodulate the signal can recover the sensor ID and incorporate it into a crafted packet, making replay and packet injection trivially achievable with commodity software-defined radio hardware.
- **False reading acceptance due to permissive receiver filtering**: Receivers configured to accept any packet with a matching sensor ID, without plausibility filtering on the pressure value, will accept crafted packets reporting zero pressure, above-range values, or simultaneous loss events across all four wheels, producing false warning states that may distract the driver or trigger automated responses.
- **Enrollment mechanism abuse and rogue sensor registration**: The enrollment procedure, which is triggered by tire replacement or wheel rotation events, requires the receiver to accept new sensor IDs. An attacker who can transmit a correctly formatted enrollment packet may register a rogue transmitter whose subsequent packets are treated as legitimate sensor data. Furthermore, in implementations where enrollment is initiated through a **Bluetooth Low Energy** (**BLE**) interface, prior analysis of specific production systems has revealed memory safety weaknesses in the code executed during the enrollment procedure that may be exploitable.
- **Fixed sensor ID enabling persistent vehicle tracking**: Because each sensor transmits a fixed unique identifier at regular intervals across the vehicle's operational life, a passive observer positioned along a road or in a parking area can correlate successive transmissions from the same sensor ID over time and construct a movement history of the associated vehicle without any active interaction with the vehicle.

Camera system weaknesses at the physical, interface, and inference layers

Camera-based perception systems are susceptible to weaknesses operating at three distinct levels: the **physical optical layer**, the **interface connectivity layer**, and the **machine learning inference layer**.

At the **physical** layer, the image sensor is inherently passive and accepts photons from any source in its field of view. High-intensity near-infrared illumination directed at the camera aperture can saturate the image sensor's photodiodes, producing a region of saturated pixels in

the output frame that eliminates useful visual information for the affected area. The saturation threshold depends on the sensor's dynamic range and the effectiveness of the auto-exposure system, but a sufficiently bright directed light source can overcome these compensating mechanisms. Because near-infrared light is outside the visible spectrum, this type of attack is not apparent to vehicle occupants observing the camera display, nor does it produce any visible indication to other road users.

At the **interface layer**, SerDes links carry the raw video stream from the camera to the processing unit as a continuous electrical signal. In most current deployments, this stream is not cryptographically authenticated end-to-end. Thus, an attacker with physical access to the wiring harness can inject a crafted video stream at the link level, that is, on the SerDes wire between the camera and ECU, presenting the processing ECU with entirely fabricated imagery while the physical camera module continues to function as normal. Even if authentication is implemented on the video path, weaknesses can still exist in certificate handling, frame replay protection, and partial authentication coverage. For example, systems that support region of interest authentication rather than full-frame authentication introduce an additional concern; if the authenticated region on a frame is selected without a cryptographically secure randomness source, an attacker who can predict which region will be authenticated can craft injected frames that preserve the authenticated region while substituting fabricated content in the remaining area. Additionally, the I2C control channel, which governs exposure settings, synchronization, and camera configuration, is generally more accessible with simpler hardware than the video data path and may have fewer authentication controls in place.

A separate interface-layer weakness concerns sensor identity authentication. In most deployed systems, the processing ECU has no mechanism to verify the physical identity of the camera module connected to a given SerDes port. As a result, a camera module can be replaced with a counterfeit device that presents a valid electrical interface while producing imagery that has been modified or selectively fabricated, without the processing ECU being able to distinguish the replacement from the original sensor.

At the **inference layer**, DNN-based object detectors are susceptible to adversarial inputs. The vulnerability arises from the structure of the network's learned decision boundaries, which can be exploited by adding small, carefully computed perturbations to a genuine input. In the automotive context, prior research has shown that adversarial patches such as printed stickers placed on traffic signs within the camera's field of view have caused production-grade object detectors to fail to detect pedestrians, misclassify stop signs, and generate ghost detections of obstacles, although there are no physical objects there [5,6].

> **Note**
>
> **Terms:**
>
> Adversarial patch is a compact printed image placed in a camera's field of view, optimized to cause object detection models to misclassify or fail to detect objects across varying viewing distances and angles.

Specific weaknesses corresponding to these three layers include the following:

- **Near-infrared saturation of image sensor photodiodes**: Directed near-infrared illumination from a source outside the visible spectrum can saturate pixels in the affected region of the sensor output. The resulting saturated area is indistinguishable from an overexposed frame segment and may cause the perception model to produce degraded or absent detections in the affected field of view.
- **Adversarial patch-induced object detection failure**: A compact printed adversarial patch placed on or near an object in the camera's field of view can cause the object detection model to fail to detect that object, misclassify it as a benign category, or generate false detections in surrounding areas.
- **Lack of end-to-end authentication on the SerDes video stream**: An attacker with access to the SerDes link between camera module and processing ECU can inject a crafted video stream that the processing ECU accepts as genuine camera output, because the stream carries no end-to-end cryptographic authentication. This attack path requires physical vehicle access but does not require exploitation of any software vulnerability in the processing stack.
- **I2C control channel configuration manipulation**: The I2C bus accessible at or near the camera connector can be probed and written to with readily available hardware such as USB-to-I2C adapters or low-cost microcontroller boards capable of acting as a bus master. Unauthenticated writes to camera configuration registers can modify exposure, gain, frame timing, and synchronization parameters in ways that degrade image quality or disrupt the camera module's operation without any visible external indication.

Unauthenticated RF reception and waveform predictability weaknesses in radar systems

The fundamental weakness of radar-based sensing is that the receiver cannot distinguish a genuine reflection from a crafted transmission that reproduces the signal characteristics of a genuine reflection. An automotive radar measures range by correlating the frequency

difference between the transmitted **chirp** and the received signal, that is, the **beat frequency**, to compute the distance to a detected object. An attacker who transmits a signal that produces the correct beat frequency when mixed with the radar's own transmit waveform will cause the processing pipeline to register an object at the corresponding range. Because the radar operates on mathematical properties of the received signal rather than on any form of authentication or identity verification, an injected false return is indistinguishable from a genuine reflection at the signal processing stage.

Note

Terms:

- Chirp is a radar waveform whose frequency increases or decreases linearly over a defined sweep period. The chirp slope and duration determine range resolution and maximum detection range in FMCW systems.
- Beat frequency is the difference in frequency between the transmitted chirp and the echo signal received back at the radar; because the echo arrives slightly later than the transmitted signal, the two signals differ in frequency by an amount that increases with target distance, and it is this frequency difference that the radar uses to compute range.

The feasibility of this class of attack depends on the attacker's ability to recover the radar waveform parameters. Early automotive radars using simple fixed-chirp waveforms are vulnerable to an attacker who can estimate the chirp slope and timing from a brief period of passive observation. Modern radars using randomized chirp sequences or waveform agility increase the difficulty of synchronized injection, that is, they raise the bar for the attacker but do not eliminate the threat entirely.

Specific weaknesses related to open RF reception architecture of automotive radar include the following:

- **Synchronized chirp injection creating phantom objects**: An attacker who recovers the radar's chirp parameters from a brief period of passive observation can transmit a signal that the radar's range-Doppler processing pipeline interprets as a reflection from an object at an attacker-specified range and velocity. The resulting phantom object entry in the detected object list then propagates to the sensor fusion layer as a plausible sensor input, unless cross-modal consistency checks are in place to detect the

inconsistency against inputs from other sensing technologies such as cameras or LiDAR.

- **Broadband noise jamming suppressing real object detection**: Continuous broadband interference transmitted on the radar's operating frequency elevates the noise floor seen by the CFAR algorithm, causing it to raise the detection threshold to a level that suppresses legitimate reflections from real objects. The attacker's power advantage is significant in this scenario because the attacker's signal experiences only single-path propagation loss, while a genuine reflection experiences round-trip propagation loss proportional to the fourth power of target range, meaning that the attacker requires considerably less transmit power than the radar itself to corrupt the detection process.

LiDAR physical-layer and interface weaknesses

LiDAR systems that operate on the time-of-flight principle share a physical-layer weakness with radar: the receiver has no means of distinguishing a genuine returning photon that originated from its own emitter from one delivered by an external source. A pulsed LiDAR emits outgoing laser pulses and registers returning photons that arrive within a timing window consistent with a reflection at some range. An attacker who injects photons at the sensor's operating wavelength with a controlled delay relative to the outgoing pulse causes the sensor to record a false range measurement at the distance corresponding to that delay. Because range is derived from photon arrival time alone, with no property that identifies the photon origin, the attack operates entirely at the physical layer and does not require bypassing any cryptographic or protocol-level control.

Anti-spoofing techniques based on **pulse fingerprinting** can be partially circumvented by projecting a sufficiently large number of pulses distributed across the sensor's scan cycle, overwhelming the fingerprinting mechanism through volume rather than precision [7]. Moreover, high-power near-infrared illumination directed at the sensor aperture can saturate the SPAD or APD array, creating a persistent blind sector that suppresses all returns from the affected direction.

> **Note**
>
> **Terms:**
>
> Pulse fingerprinting is a LiDAR anti-spoofing technique that encodes identifying timing characteristics into outgoing laser pulses and verifies that returning photons match those signatures before accepting them as valid range measurements.

The CAN bus-based control interface used for LiDAR configuration and health monitoring presents a further weakness: unauthenticated control messages can force mode changes, trigger firmware update sequences, or modify operational parameters that affect the sensor's detection behavior. Similarly, the Automotive Ethernet interface used to transmit point cloud data to the HPC is susceptible to lack of message authentication, replay attacks, and lack of link-layer integrity verification where these controls have not been implemented. Both interface-level weaknesses are similar to the weaknesses covered in *Chapter 6*.

Specific weaknesses related to LiDAR are as follows:

- **Timed photon injection creating false range measurements**: Photons delivered at the sensor's operating wavelength with a controlled delay after the outgoing pulse are registered by the SPAD or APD array as valid range returns. An attacker who characterizes the sensor's pulse timing can generate false point cloud objects at arbitrary positions within the sensor's field of view.
- **Statistical circumvention of pulse fingerprinting anti-spoofing controls**: Where the sensor attempts to verify that returning pulses match its own emission signature, projecting a high pulse repetition rate distributed uniformly across the scan cycle increases the probability that at least one injected pulse satisfies the expected timing window by statistical chance, partially defeating the fingerprinting check without requiring precise knowledge of the sensor's synchronization parameters.
- **High-intensity near-infrared illumination creating persistent blind sectors**: Sustained high-power near-infrared illumination directed at the sensor aperture saturates the receiver array in the affected angular segment, producing a blind sector in the point cloud output for as long as the illumination continues. As a result, objects located within the blind sector are not detected by the vehicle's perception system during the period in which the illumination is sustained.
- **Unauthenticated CAN control channel enabling mode manipulation**: LiDAR sensors that accept configuration and operational mode commands over an unauthenticated CAN interface are susceptible to crafted messages that can modify detection sensitivity thresholds, force low-power modes that reduce effective range, or trigger firmware update sequences that may be used to load modified firmware under certain update implementations.

Ultrasonic proximity sensor weaknesses due to lack of echo source validation

Ultrasonic sensors are susceptible to acoustic interference from any source capable of producing sound in the 40 to 58 kHz operating range. The weakness is due to lack of source validation in the echo processing pipeline. That is, the sensor measures the time of flight of the

first returning echo that exceeds the detection threshold, without any mechanism for determining whether that echo originated from the sensor's own outgoing pulse or from an externally injected acoustic signal. An attacker in close physical proximity can inject false echo returns, causing the parking assistance system to report an obstacle that does not exist or suppress legitimate returns by transmitting continuous interference that raises the noise floor above the detection threshold.

Because inexpensive ultrasonic transducers operating in the 40 to 58 kHz range are commercially available as components in consumer electronics and robotics applications, the hardware barrier to conducting this type of attack is low. The short effective range of the attack, which is constrained by the operating distance of the sensor itself, requires an attacker to achieve close physical proximity to the vehicle. In automated parking scenarios where the vehicle operates at low speed in confined spaces such as parking garages or driveways, an attacker can easily position themselves within effective injection range and corrupt the proximity sensor data to affect the vehicle's control decisions.

Specific weaknesses due to lack of echo source validation in ultrasonic sensors include the following:

- **False echo injection producing phantom obstacle reports**: An externally injected acoustic pulse at the sensor's operating frequency, timed to arrive at the sensor within the detection window, is processed as a legitimate reflection from an obstacle at the corresponding distance. The parking assistance ECU generates a proximity warning or triggers an automated stop response based on the falsified sensor reading.
- **Continuous acoustic interference suppressing legitimate return detection**: Sustained acoustic transmission at the sensor's operating frequency raises the noise floor seen by the echo detection threshold, causing the sensor to fail to detect genuine obstacles. An automated parking or maneuvering system that receives no proximity signal may interpret the absence of returns as clear space and continue movement into an actual obstruction, potentially resulting in a collision.

Asymmetric authentication between sensor data plane and control plane

Externally mounted sensors with network connectivity commonly separate high-bandwidth measurement outputs from a lower-bandwidth control and configuration channel. A camera module may deliver its video stream over a SerDes link while receiving exposure and synchronization commands over I2C. A LiDAR unit may deliver its point cloud over Automotive Ethernet while receiving operational mode commands and firmware update triggers over CAN.

This separation is a functional design feature, but it introduces a security consequence when the control plane is protected with less rigor than the data plane. This is because its bandwidth

and apparent functional criticality are lower. This perception can lead to authentication being optional or entirely omitted on control-plane interfaces. In practice, the control plane is an effective attack target precisely because it can influence sensor behavior persistently. A crafted control message that adjusts calibration parameters, forces a synchronization event, or triggers a firmware update mode can affect the sensor's outputs even if the data plane itself is protected by end-to-end authentication and integrity verification. Both planes must therefore be explicitly included in the attack surface analysis and held to equivalent authentication standards where technically feasible.

Trust boundary misconfiguration in domain and zonal architectures

In domain and zonal vehicle architectures, a cross-cutting weakness arises from the way externally mounted sensor data is aggregated at zone controllers or domain controllers before being forwarded to the central HPC, and it applies regardless of the specific sensor type involved. This aggregation reshapes the trust boundaries applicable to sensor data and affects how a sensor compromise propagates. A single compromised sensor can inject malicious data into the aggregated stream before it is authenticated or validated by the receiving domain controller, and a compromised zone controller can affect the data delivered by multiple sensors simultaneously.

A common design error in these architectures is to treat sensor data as trusted once it has been accepted by a zone controller, relying on the zone boundary as the sole trust enforcement point. If the zone controller does not validate individual sensor messages before aggregating them, or if the communication path between the sensor and the zone controller does not apply authentication, then the effective trust boundary is at the sensor's RF or optical interface rather than at the zone controller. An attacker targeting the physical signal path can therefore inject data that propagates through the zone controller to the HPC with the same trust level as legitimate sensor data.

DNN perception system weaknesses at training and model integrity layers

The weaknesses of DNN-based perception systems arise from fundamental properties of the learning process rather than from implementation errors in the conventional software sense. A neural network is a function approximator that learns a mapping from inputs to outputs based on the statistical structure of its training data. This process does not produce a model that reliably generalizes to all possible inputs, but only to inputs that are statistically similar to the **training distribution**. Inputs that exist outside the training distribution, or that are specifically crafted to exploit the model's learned decision boundaries, can cause the model to produce confident but incorrect outputs.

The perception pipeline is also subject to **distributional shift**, in which the statistical properties of the operating environment differ from those of the training data in ways that degrade model performance without any deliberate adversarial input. Unusual lighting conditions, weather effects, or objects with visual characteristics underrepresented in the training dataset can cause degraded detection performance with safety implications that are operationally similar to those of a deliberate attack.

Note

Terms:

- Training distribution is the set of inputs and their statistical properties on which a neural network is trained; a model's ability to produce correct outputs is bound by this distribution, and inputs that fall outside may produce unreliable or incorrect results.
- Distributional shift is a condition in which the statistical properties of inputs encountered during deployment differ meaningfully from those of the training distribution, causing degraded model performance without any deliberate adversarial input.

A further concern is the integrity of the model itself. Weaknesses in the **Over-the-Air (OTA)** update process and in the security of the model supply chain can allow an attacker to substitute or tamper with the model in ways that introduce malicious behavior that is difficult to detect through standard functional testing. The manipulated model could behave correctly under normal operating conditions but produce incorrect outputs in specific trigger scenarios, which could result in unsafe or safety-critical vehicle behavior.

Specific weaknesses related to DNN perception include the following:

- **Adversarial perturbation exploitation of DNN decision boundaries**: Inputs crafted to maximize a model's classification error while remaining imperceptible to a human observer cause the model to produce high-confidence misclassifications. For camera-based object detection, this is physically realized through adversarial patches placed in the camera's field of view. The attack's effectiveness transfers partially between models with similar architectural characteristics, limiting the protective value of model architecture confidentiality.
- **Performance degradation under distributional shift without deliberate attack**: Operational conditions not adequately represented in training data, including unusual sun angles, sensor contamination, or novel object appearances, produce degraded

detection accuracy with safety consequences equivalent to those of a deliberate attack. This is relevant to security assessment because an attacker can deliberately trigger such conditions without needing to compute an adversarial perturbation.

- **OTA model substitution via unsigned update channel**: An OTA update process that does not enforce cryptographic signature verification before activating a new model allows an attacker who compromises the update delivery channel or the distribution infrastructure to replace the active model with a manipulated version. The substituted model is activated directly on the vehicle without passing through any functional validation, meaning that incorrect or malicious behavior takes effect immediately upon the next inference cycle.
- **Supply chain model tampering producing dormant backdoor behavior**: An attacker who is able to tamper with or replace the model during the development, training, or integration phase can craft a manipulated model that behaves correctly under normal conditions but produces incorrect outputs for specific trigger inputs that would typically not be included in a standard test suite. The backdoored model then passes pre-deployment validation and is released as part of the normal software delivery process, with the malicious behavior remaining dormant until the trigger condition is encountered in the field.

In-vehicle AI assistants weaknesses

LLM-based in-vehicle AI assistants receive input not only through voice commands but also through external data sources such as navigation point of interest descriptions, incoming messages, and connected service responses, all of which the model processes as part of its context. This creates a **prompt injection** exposure (**Open Worldwide Application Security Project** (**OWASP**) LLM Top 10: LLM01 Prompt Injection [8]) where content in any of these data sources that contains instruction-like text may be treated by the model as an authoritative directive rather than as passive data, potentially causing the assistant to invoke functions without any interaction from the legitimate user. Unlike conventional input validation weaknesses, prompt injection does not arise from a parsing error but from the fundamental property of an LLM that cannot reliably distinguish between instructions and data once both are present in the same context window [9].

A related but distinct weakness is **acoustic command injection**, which exploits the difference between human auditory perception and the statistical signal processing performed by the speech recognition layer. By embedding carefully computed perturbations into an audio signal, an attacker can produce a sound that a human listener perceives as background noise or music but that the speech recognition model transcribes as a specific command, which the assistant then acts upon. This attack operates at the speech recognition layer before the LLM receives the

input and can be delivered through audio played by a nearby device, broadcast media, or content streamed through the vehicle's own audio system.

The practical impact of both above-mentioned weaknesses depends directly on the **permission scope** granted to the assistant (OWASP LLM Top 10: LLM06 Excessive Agency). An assistant authorized to invoke navigation changes, place calls, send messages, or control connected home devices is a substantially higher-value target than an assistant that is limited to informational responses.

A further systemic concern is voice data **privacy** (OWASP LLM Top 10: LLM02 Sensitive Information Disclosure). In-vehicle assistants process spoken language that could include home and work addresses, contact names, and calendar entries, and if this data is transmitted to cloud infrastructure without adequate encryption and access control, or retained beyond operational necessity, this could lead to a persistent privacy issue.

Now that we have surveyed common sensor security weaknesses, it is time to see how these can be put to use during a penetration testing engagement.

Penetration testing techniques against vehicle sensors and AI

Penetration testing of vehicle sensors and AI systems is organized around the distinct attack surfaces each component presents. To simplify, this section focuses on three core phases applicable to sensor and AI system testing: *discovery*, where the security tester characterizes the sensor's interfaces and signal properties passively before any active intervention; *vulnerability analysis*, where the findings from discovery are assessed to determine which attack paths are viable; and *exploitation*, where applicable techniques are applied to confirm and demonstrate the impact of identified weaknesses.

MEMS inertial sensor testing

The security tester begins the discovery phase by identifying the target sensor model and its documented resonant frequency from published datasheets or technical literature. The tester then locates the sensor's physical position within the vehicle and assesses the enclosure design and **Printed Circuit Board** (**PCB**) coupling paths that may influence acoustic coupling. In parallel, the tester identifies how the consuming ECU's output is accessible, whether through the vehicle's diagnostic interface, CAN traffic, or a dedicated sensor output signal, establishing the observation point against which subsequent stimulus results will be compared.

Next, vulnerability analysis focuses on assessing whether the consuming ECU applies plausibility checking to the inertial sensor output. The penetration tester reviews the ECU's response to baseline sensor readings and determines whether any range validation or cross-

sensor consistency check is applied to the accelerometer or gyroscope output. The sensor's resonant frequency and placement are also assessed to determine the sound pressure level and delivery angle required to achieve sufficient acoustic coupling, as these factors determine the practical feasibility of exploitation.

Finally, the exploitation phase involves generating acoustic stimuli at and around the identified resonant frequency using a precision audio source, varying sound pressure levels and delivery angles to characterize the attack envelope. A pentester can monitor the consuming ECU's output through the diagnostic interface or vehicle network during stimulus delivery, observing whether anomalous inertial readings are produced and whether those readings propagate into safety-relevant functions such as electronic stability control. Note that this attack requires no access to the vehicle network and can be conducted using commercially available audio generation equipment, which lowers the practical barrier for this class of attack compared to other sensor exploitation techniques covered in this chapter.

TPMS testing

The discovery phase begins with passive RF reconnaissance, capturing TPMS transmissions from the target vehicle using a **Software-Defined Radio** (**SDR**) receiver operating in the 315 MHz or 433.92 MHz band depending on the regional standard. Selecting the correct band is a prerequisite for successful capture, as equipment calibrated for one region will not receive transmissions from a vehicle manufactured for the other. Examples of appropriate tools include **RTL-SDR V3** [10] and **HackRF One** [11], which provides sufficient sensitivity and frequency coverage for both bands. Then, signal decoding can be performed using **rtl_433** [12], which includes built-in TPMS protocol support for a wide range of production sensor families, or using the **Universal Radio Hacker** (**URH**) [13] for manual demodulation and protocol analysis where the target sensor is not recognized by automated decoders. Moreover, **GNU Radio** [14] with the **gr-tpms** [15] module provides a flexible flow-graph environment for capturing, demodulating, and parsing packets in real time. A pentester can record sensor IDs, pressure values, and packet structure from multiple wheel sensors and confirm whether sensor IDs are static across successive transmission cycles, which allows for passive tracking of the target vehicle.

Next, vulnerability analysis focuses on assessing the authentication posture of the TPMS packet structure and the filtering behavior of the receiver. The penetration tester can examine captured packets to confirm whether any cryptographic authentication field is present beyond the CRC checksum, which serves only to detect transmission errors rather than authenticate the source. The receiver's plausibility filtering is assessed by reviewing the observed acceptance behavior for packets with matching sensor IDs, determining whether the regulatory requirement for rapid pressure loss alerts creates pressure to accept packets with minimal validation. The enrollment mechanism is also characterized during this phase,

establishing whether it requires diagnostic-level access to initiate and whether BLE enrollment support is present, as both determine which exploitation paths are viable.

Finally, the exploitation phase covers packet injection, enrollment mechanism abuse, and BLE enrollment path assessment, aligned to the weaknesses identified in the weaknesses section. A pentester performing packet injection uses an SDR transmitter to replay captured packets and transmit crafted packets with modified pressure values, assessing whether false warnings are triggered and whether the receiver applies any plausibility filtering to implausible readings. Test cases include below-threshold pressure injection, zero-pressure reports across all four wheels, and packets carrying sensor IDs not registered with the vehicle, each targeting the permissive receiver filtering weakness. The enrollment mechanism can be tested by transmitting a correctly formatted enrollment packet with an attacker-controlled sensor ID during or shortly after an enrollment trigger event, evaluating whether a rogue transmitter can be registered as a legitimate sensor. For TPMS implementations with BLE enrollment support, the security tester assesses the BLE advertisement parsing path using a BLE packet sniffer such as **Ubertooth One** [16] combined with **Wireshark** [17], with particular focus on the enrollment message parsing code path where prior research has documented memory safety weaknesses that may be exploitable.

Camera system testing

First, a pentester in the discovery phase can begin with physical inspection to identify the camera type, connector pinout, and SerDes link variant used on the target vehicle. The tester can passively observe I2C bus traffic using **i2cdump** to characterize the configuration register layout and establish a baseline of normal exposure, gain, and synchronization settings. In parallel, the tester can identify which ADAS functions consume the camera output, such as automatic emergency braking, lane keeping assistance, or object detection, as these determine the downstream impact of a successful optical or interface-layer attack.

In the next phase, vulnerability analysis, the tester assesses the authentication posture of both the SerDes video stream and the I2C control channel and evaluates the inference layer for susceptibility to adversarial inputs. The penetration tester inspects the SerDes stream to determine whether it carries any end-to-end cryptographic authentication tag and uses **i2cdetect** to enumerate addressable devices on the I2C bus and assess whether the control channel applies any access controls before accepting register writes. Furthermore, the spectral sensitivity of the image sensor is characterized to confirm whether the 850 nm to 940 nm near-infrared range falls within the sensor's sensitivity window, establishing the feasibility of optical saturation testing. Additionally, the deployed object detection model is assessed for its known susceptibility to adversarial patches based on its architecture and any published research on similar production systems.

In the exploitation phase, the pentester conducts various testing such as optical saturation, adversarial patch-induced detection failure, SerDes stream injection, and I2C control channel manipulation. For example, to perform near-infrared saturation testing, the pentester directs illumination at the camera aperture at varying intensities and distances and records the threshold at which saturation artifacts appear in the output frame and at which ADAS detection functions degrade, confirming that the effect is not visible to vehicle occupants. Adversarial patch testing can be performed using the **Adversarial Robustness Toolbox** (**ART**) [18] or **Foolbox** [19] frameworks to generate patches applied to objects in the camera's field of view, with the security tester observing downstream effects on ADAS behavior such as suppression of automatic emergency braking in response to a patched obstacle.

Moreover, to perform interface-level testing requires physical access to the target system. For example, SerDes link injection requires physical access to the wiring harness and a SerDes interposer configured to capture and re-inject the link signal, with the penetration tester assessing whether a crafted video stream injected at the link is accepted by the processing ECU without error. In addition, I2C control channel testing uses **i2cset** to perform targeted register write tests, assessing whether unauthenticated writes to configuration and mode registers are accepted by the camera module and whether such writes can modify exposure, gain, frame timing, or synchronization parameters in ways that degrade image quality or disrupt camera operation.

Radar testing

The penetration testing begins with the discovery phase with passive characterization of the radar's RF signal before any active transmission is performed. Using a software-defined radio receiver and a **GNU Radio** flow graph, a pentester can capture the radar's emissions and resolve the chirp parameters including slope, duration, and any frequency agility or pseudo-random scheduling applied by the radar. In parallel, the security tester can identify the vehicle network interface carrying the radar's object list output and record the message format, update rate, and any observable authentication or integrity fields.

Vulnerability analysis focuses on assessing the authentication posture of both the radar's network output and its configuration interface. The penetration tester assesses whether the object list carries any cryptographic authentication tag and whether the radar module accepts unauthenticated configuration and calibration commands over CAN. The recovered chirp parameters are analyzed to determine the feasibility of synchronized injection, that is, whether the waveform is sufficiently predictable to allow a crafted return to be timed correctly. Fixed or low-complexity chirp sequences represent a higher-risk finding than randomized or agile waveforms, as they significantly lower the bar for active exploitation.

The exploitation phase covers both RF-layer attacks and cross-modal fusion evasion. A pentester can perform active RF testing by transmitting crafted chirp returns synchronized

with the radar's emission cycle and monitoring the fusion layer for phantom objects propagating into the object list. Broadband noise jamming involves transmitting interference on the operating frequency and monitoring for suppression of real objects below the CFAR threshold. A key consideration is that all active transmission tests in the 76 to 81 GHz band require regulatory authorization in the relevant jurisdiction and must be conducted in a controlled environment. Moreover, sensor fusion evasion testing involves presenting a phantom radar object at a specific range while simultaneously applying an adversarial camera patch that suppresses detection of the corresponding object, observing whether the fusion layer resolves the discordant inputs conservatively or accepts the radar-only reading.

LiDAR testing

First, a pentester starts with the discovery phase with physical inspection of the sensor to identify the operating wavelength, either 905 nm or 1550 nm, and confirm whether the design is rotating or solid-state, as these properties determine which optical attack techniques are feasible. A security tester can also log CAN control traffic using **candump** [20] during sensor startup and normal operation to identify control message identifiers and establish a baseline of expected configuration and mode command traffic. In parallel, the Automotive Ethernet point cloud stream can be captured using **Wireshark** or the sensor manufacturer's diagnostic software and decoded to confirm format, update rate, and coordinate frame. As part of this phase, the tester also characterizes the sensor's pulse timing parameters, which are needed to assess the feasibility of timed photon injection and to determine whether pulse fingerprinting is implemented.

Vulnerability analysis assesses the authentication posture of both the CAN control interface and the Ethernet data path and evaluates the optical attack surface against the weakness patterns identified in the weaknesses section. The penetration tester can transmit targeted messages on the CAN control channel, testing whether mode commands, detection sensitivity modifications, calibration parameter writes, and firmware update triggers are accepted without authorization. The point cloud stream is inspected for the presence of any integrity protection field. Where pulse fingerprinting is present, the tester assesses the implementation to determine whether it can be statistically circumvented by projecting a high pulse repetition rate distributed across the scan cycle.

The exploitation phase tests each of the optical-layer weaknesses identified during vulnerability analysis. A pentester performing timed photon injection uses a pulsed laser source at the sensor's operating wavelength, synchronized to the scan cycle, to insert false point cloud objects at attacker-specified positions. If pulse fingerprinting is used to prevent spoofing, the security tester projects a high repetition rate of pulses distributed uniformly across the scan cycle to assess whether at least one injected pulse satisfies the expected timing window by statistical chance, partially defeating the fingerprinting check. Aperture saturation

testing applies sustained near-infrared illumination directed at the sensor aperture and measures the extent of the resulting blind sector within which real objects are suppressed. It is important that active optical tests using laser sources be conducted under appropriate safety controls given the eye-safety risks associated with near-infrared illumination at the power levels required for these tests. Moreover, sensor fusion evasion testing combines aperture saturation to suppress a real object from the LiDAR point cloud while a corresponding radar return for the same object remains present. The penetration tester then observes whether the fusion layer flags the cross-modal discordance or treats the object as not present based on the LiDAR output alone.

Ultrasonic testing

The discovery phase begins with the penetration tester physically inspecting the target to confirm sensor placement and identify the interface type, either LIN or PWM, from the connector pinout. The tester then monitors CAN or LIN bus traffic from the parking assistance ECU during slow-speed maneuvers to characterize the proximity data format, update rate, and detection threshold. This phase also establishes the acoustic pulse repetition interval and determines whether the sensor applies echo gating that would reject signals arriving outside the expected timing window, as this directly affects the feasibility of timed injection during exploitation.

Next phase, vulnerability analysis, assesses whether the sensor applies any form of echo source validation, that is, whether it can distinguish between a genuine reflection of its own outgoing pulse and an externally injected acoustic signal. Further, the penetration tester can review the pulse repetition interval and detection window recovered during discovery to determine the timing precision required for a successful false echo injection. It is worth mentioning that the hardware barrier to conducting acoustic injection against ultrasonic sensors is low, given that transducers operating in the 40 to 58 kHz range are commercially available in consumer electronics and robotics components, which increases the practical risk associated with this weakness. In addition, the tester should assess whether the sensor or parking assistance ECU exposes a configuration interface that allows detection sensitivity thresholds or echo timing parameters to be modified without authorization, as unauthorized changes to these settings can directly affect what the sensor reports as an obstacle.

Lastly, the exploitation phase tests the weaknesses identified. For example, a pentester can perform false echo injection testing by positioning an ultrasonic transducer adjacent to the target sensor and injecting acoustic pulses timed to produce proximity reports at attacker-specified distances. By monitoring the CAN or LIN bus output the tester can confirm whether the injected returns are processed as genuine obstacle detections by the parking assistance ECU. Moreover, continuous interference testing involves transmitting sustained acoustic energy at the sensor's operating frequency and confirming whether the noise floor rises

sufficiently to prevent the sensor from reporting genuine obstacles within its detection range, which in an automated parking scenario could cause the system to interpret the absence of returns as clear space and continue movement into an actual obstruction.

AI assistant and LLM testing

For AI assistant and LLM testing, the discovery phase focuses on interface enumeration and supply chain assessment. A security tester should identify all data sources the assistant is authorized to ingest, such as navigation points of interest, incoming messages, calendar entries, and connected service API responses, and map the complete set of vehicle functions the assistant can invoke, noting which require explicit user confirmation and which are accessible through conversational commands alone. The OTA update channel is assessed to determine whether model update packages are cryptographically signed and whether signature verification is enforced before activation.

Next, vulnerability analysis assesses the permission scope of the assistant and the trust posture applied to each ingested data source. The penetration tester reviews the mapped function set to identify which safety-relevant or privacy-sensitive functions are reachable without an explicit confirmation step, as an overly broad permission scope directly amplifies the impact of both prompt injection and acoustic command injection. Each identified data source is evaluated for whether its content is treated as trusted input or as untrusted data subject to validation before being incorporated into the model's prompt context. In addition, the security tester can assess whether voice recordings and conversational logs are transmitted to cloud infrastructure, and if so, whether the transmission is encrypted and whether a documented retention policy governs how long the data is stored (OWASP LLM Top 10: LLM02 Sensitive Information Disclosure).

The exploitation phase then covers, e.g., acoustic command injection, prompt injection, and tool call scope testing, aligned to the weaknesses described in the weaknesses section. A pentester could perform acoustic command injection by generating audio adversarial examples [21], embedding inaudible perturbations into audio played through the vehicle's audio system or a nearby source, and monitoring the speech recognition output to confirm whether the target transcription is produced. Prompt injection testing (OWASP LLM Top 10: LLM01 Prompt Injection) focuses on crafting instruction-like content within controlled instances of each identified data source, such as a navigation point loaded from a test dataset or a message delivered through a connected test device, and observes whether the model treats the injected content as a directive and attempts to execute it without user interaction.

Moreover, tool call scope testing targets the excessive agency weakness (OWASP LLM Top 10: LLM06 Excessive Agency) by attempting to invoke sensitive functions through indirect conversational paths without triggering a confirmation prompt, documenting which function categories are reachable without explicit user authorization and confirming the effective permission boundary of the assistant.

Now that we have built an understanding of the common penetration testing techniques, it's time to explore some real-life case studies in which sensors and AI/ML systems were compromised.

Case studies of real-life attacks on vehicle sensors and AI/ML systems

In this section, we examine three real-world attacks on vehicle sensors and AI/ML systems, analyzing how attackers identified and exploited weaknesses in the machine learning inference pipeline, the physical sensing layer, and the wireless sensor communication stack. The case studies illustrate how the attack surfaces described in the preceding sections can be discovered and exploited in practice. Together, they span a range of threat scenarios from adversarial manipulation of a production neural network perception system, to physical spoofing of a LiDAR sensor using low-cost equipment, to a zero-click remote code execution chain that entered the vehicle through a TPMS sensor.

Case study 1: Adversarial machine learning attacks on Tesla Autopilot

This case study examines three attack scenarios developed by security researchers against the Tesla Autopilot system on a Model S [22]. The researchers conducted reverse engineering of the Autopilot ECU (APE) and CAN bus communication stack, then designed and demonstrated attacks against three Autopilot functions: the *remote steering* capability, the neural network-based *autowipers* system, and the *lane detection* system. The most operationally significant finding was the fake lane attack, in which three small adhesive patches placed on the road surface at an intersection caused the Autosteer system to steer the vehicle into the oncoming lane without any network access to the vehicle. The research was publicly disclosed in 2019 and Tesla addressed the APE root access vulnerability with a firmware update.

An overview of the attack steps for this case study is given in *Figure 9.2*:

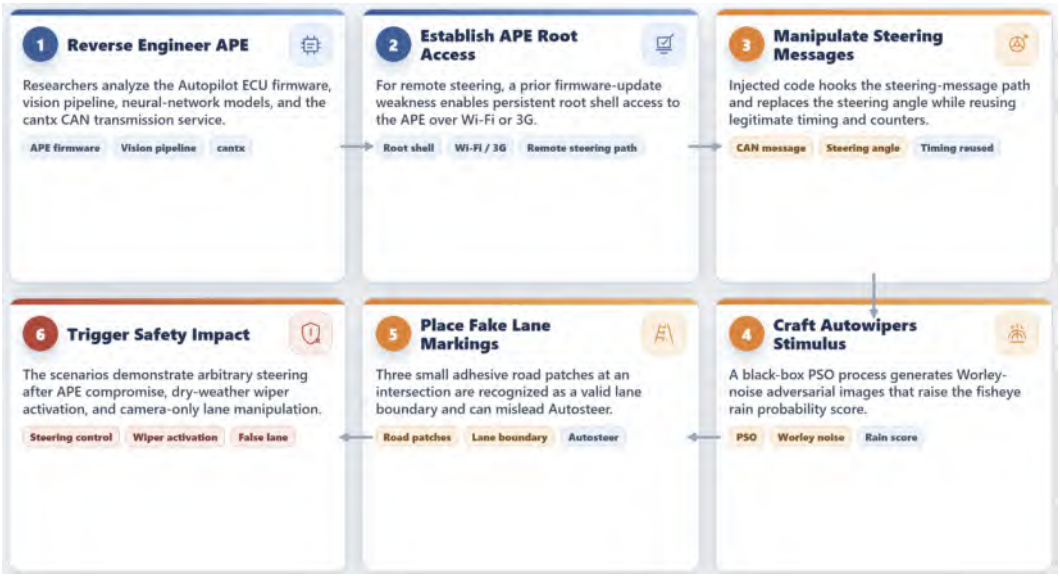


Figure 9.2: How software compromise and physical-world sensor manipulation

Note

Source: `https://keenlab.tencent.com/en/whitepapers/Experimental_Security_Research_of_Tesla_Autopilot.pdf` [22]

Discovery and reconnaissance

The researchers began with static reverse engineering and dynamic debugging of the APE firmware, which runs a customized Linux environment with Autopilot software binaries located under `/opt/autopilot`. The APE module is based on the PX2 AutoChauffeur platform and connects to all primary cameras, that is, main, narrow, and fisheye, exclusively via **Camera Serial Interface (CSI)** connections, with radar data relayed through an Infineon Aurix coprocessor.

The vision pipeline was identified as the central component, processing camera frames through a large shared neural network that handles multiple tasks simultaneously, including lane detection, object tracking, and environment mapping. The autowipers function uses a dedicated fisheye camera neural network defined in `fisheye.prototxt`, a rain classifier that outputs a probability value between 0 and 1 representing the likelihood of windshield moisture. Lane detection operates on 1280x960 **High Dynamic Range (HDR)** images from the

main and narrow cameras, producing a 416x640 lane probability map through a series of **Compute Unified Device Architecture** (**CUDA**) kernels.

For the CAN bus component, reverse engineering of the **cantx** service revealed the `APE2LB_CAN` protocol, a logical CAN bus transmitted over **User Datagram Protocol** (**UDP**) that routes messages from the APE to the Infineon Aurix coprocessor, which in turn forwards them to the vehicle's chassis, body, and powertrain CAN buses. The **DasSteeringControlMessage**, carrying the steering angle, control type, message counter, and a one-byte checksum, was identified as the key message used by the **Electric Power Assisted Steering** (**EPAS**) unit to execute steering commands in **Adaptive Cruise Control** (**ACC**) and **Automatic Parking Control** (**APC**) modes.

Initial access

It is important to note that the three attack scenarios require different levels of access. First, the *remote steering* attack relies on root access on the APE, obtained through a firmware update mechanism vulnerability. Initial access was based on an approach presented at Black Hat USA 2018 [23], using a remote attack chain over 3G or Wi-Fi to first compromise the vehicle's gateway before moving laterally to reach the APE. The APE's firmware update service accepted **Hypertext Transfer Protocol** (**HTTP**)-hosted SquashFS firmware images over the network without sufficient authentication. By serving a legitimate firmware image on the first request and switching to a malicious image in the second request, the researchers caused the APE to execute a backdoored updater script with root privileges. This script modified the **Secure Shell** (**SSH**) configuration and firewall rules to allow the researchers persistent root shell access to the APE.

In contrast, *autowipers* and *lane detection* attacks require no physical or remote access to the vehicle at all. These attacks operate entirely through the physical environment that the vehicle's cameras observe, meaning an attacker within the vehicle's field of view can execute them without any connection to the vehicle's network or control of any ECU software.

Exploitation

The exploitation for each of the three attack scenarios are described as follows. First, the *remote steering* attack used dynamic code injection into the **cantx** service running on the compromised APE, hooking the `DasSteeringControlMessageEmitter::finalize_message()` function to intercept and replace the steering angle with an attacker-controlled value while reusing the legitimate message timestamp and counter. This bypassed the CAN message integrity protections and allowed arbitrary steering commands to be issued to the EPAS unit. The attack was demonstrated while the vehicle was operating in ACC mode, in which the Autosteer system actively manages lateral vehicle movement, meaning the injected commands could directly alter the vehicle's intended trajectory. The researchers demonstrated this

scenario using a gamepad, which was used to generate steering angle inputs, connected via Bluetooth to a mobile device, which in turn translated the steering angles into **DasSteeringControlMessage** values and provided the values to the compromised APE **cantx** service over a 3G or Wi-Fi channel.

Next, the *autowipers* attack used a black-box adversarial example approach. The fisheye neural network was compiled with **8-bit integer quantization** and partial private CUDA kernel implementations, making its internal computations less accessible to external analysis and leading the researchers to treat it as a black box. They applied **Particle Swarm Optimization** (**PSO**) to generate **adversarial image** perturbations that maximized the **rain probability score** of the neural network without access to its internal weights or gradients. The resulting images raised the score above the activation threshold while remaining visually indistinguishable to a human observer. The researchers initially attempted salt-and-pepper noise but found that the camera resolution was insufficient to capture its detail. They then tested **Worley noise patterns**, which appeared better suited to the task, likely because the rain detection network responds primarily to image texture.

For the physical world implementation, adversarial images based on these patterns were generated and optimized end-to-end using a test vehicle with root access to the APE: each candidate image was displayed on a screen within the fisheye camera's field of view, captured by the camera, evaluated by the autowipers neural network, and the resulting rain probability score was used to iteratively refine the image until the activation threshold was exceeded. This approach ensured that real-world distortions from the lens, display characteristics, and lighting were accounted for during generation.

It is important to note that root access was only required during this image development phase. Once the adversarial image was generated, deploying it against a target vehicle required no network or physical access to the vehicle or any software exploitation. Placing a screen displaying the pre-computed image within range of the fisheye camera of the target vehicle was sufficient to cause the autowipers to activate in dry conditions.

Note

Terms:

- 8-bit integer quantization is a technique that reduces a neural network's numerical parameters from 32-bit floating-point values to 8-bit integers, reducing the memory and computation required to run the model on embedded hardware.
- **Particle Swarm Optimization** (**PSO**) is a population-based optimization algorithm that finds solutions by iteratively adjusting a set of candidate inputs

based on their individual and collective performance, without requiring access to the internal structure of the target function. Adversarial images are inputs to a neural network that have been deliberately crafted to cause an incorrect output, typically by introducing subtle pixel-level changes that are imperceptible to a human observer but significantly alter the model's prediction.

- Rain probability score is the numerical output of Tesla's autowipers neural network, representing the model's estimated likelihood that the windshield is wet. When this value exceeds a predefined threshold, the system activates the windshield wipers.
- **Worley noise patterns** is a type of procedural texture used in computer graphics to simulate natural surfaces such as stone or cellular structures.

The *lane detection* attack produced the most operationally significant result with potential safety damage. The researchers found that Tesla Autopilot's lane recognition in Autosteer mode depends entirely on a computer vision neural network, with no cross-validation against other inputs such as **Global Positioning System** (**GPS**) map data or known road geometry. In this experiment, three small adhesive stickers placed at an intersection simulated a continuation of the right lane curving across the centerline. When the vehicle in Autosteer mode reached the intersection, the neural network recognized the stickers as a valid lane boundary, ignored the actual road markings and lanes, and thus steered the vehicle into the oncoming lane, as illustrated in *Figure 9.3*:

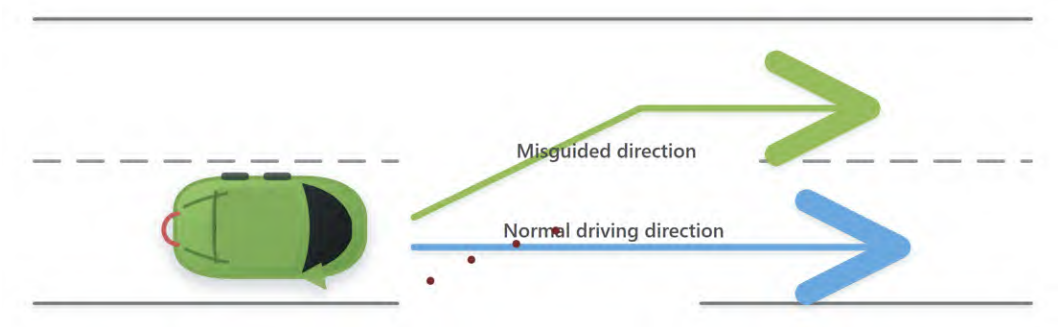


Figure 9.3: Fake lane attack using three stickers in the physical world

Persistence

In the *remote steering* attack, a persistent root shell was first established on the APE via a backdoored SSH configuration, which allowed the researchers to reconnect to the vehicle at any time over Wi-Fi or 3G without repeating the initial exploit chain. In contrast, the *autowipers* and *lane detection* attacks do not achieve persistence in the traditional sense by establishing a software foothold. Instead, both depend on physical materials remaining in place in the target vehicle's surroundings and leave no detectable trace in the vehicle's systems. For example, the road-surface stickers used in the lane attack, shown in *Figure 9.3*, can be placed and removed without requiring access to the vehicle itself.

Impact

The *remote steering* attack demonstrated that following APE compromise, spoofed CAN messages could be injected directly to the steering actuator to achieve arbitrary steering control at high speeds. With Autosteer actively managing the vehicle's lateral movement at the time, these commands had a direct effect on the vehicle's trajectory, with the potential to cause it to veer out of its lane or off the road entirely. The *autowipers* and *fake lane* attacks operated on a different principle entirely, requiring no network access or software vulnerabilities to be exploited. In both cases, a stimulus presented to a camera-based neural network was sufficient to trigger an unintended automated response: an adversarial image displayed on an electronic screen in view of the fisheye camera caused the autowipers neural network to classify the scene as rainy and activate the wipers, and three small adhesive stickers placed at an intersection were enough to cause the Autosteer system to steer the vehicle into the oncoming lane.

These three scenarios illustrate that the attack surface of an automotive AI system extends from the software stack through the physical environment the sensors observe. Adversarial manipulation of the perception layer can bypass software-level safety mechanisms when the vehicle has no means of distinguishing a genuine input from a crafted one, whether that input is a spoofed CAN message from a compromised ECU, a false optical stimulus directed at a sensor, or a lane marking crafted from adhesive stickers.

Case study 2: LiDAR spoofing via pulse recording and replay

In this attack, security researchers demonstrated that the LiDAR systems relied upon by most autonomous vehicles could be attacked with a setup costing approximately $60 [24]. The research, presented at Black Hat Europe 2015, showed that an attacker could inject phantom objects at chosen distances within the sensor's 110-degree field of view, force an autonomous vehicle to brake unnecessarily, or overwhelm the object tracking system with enough spurious signals that real objects in the environment would go undetected. The work demonstrated that the physical sensing layer of autonomous vehicles represents a largely unaddressed attack

surface, independent of any vulnerabilities in software or network communications. *Figure 9.4* shows an overview of the attack sequence steps:

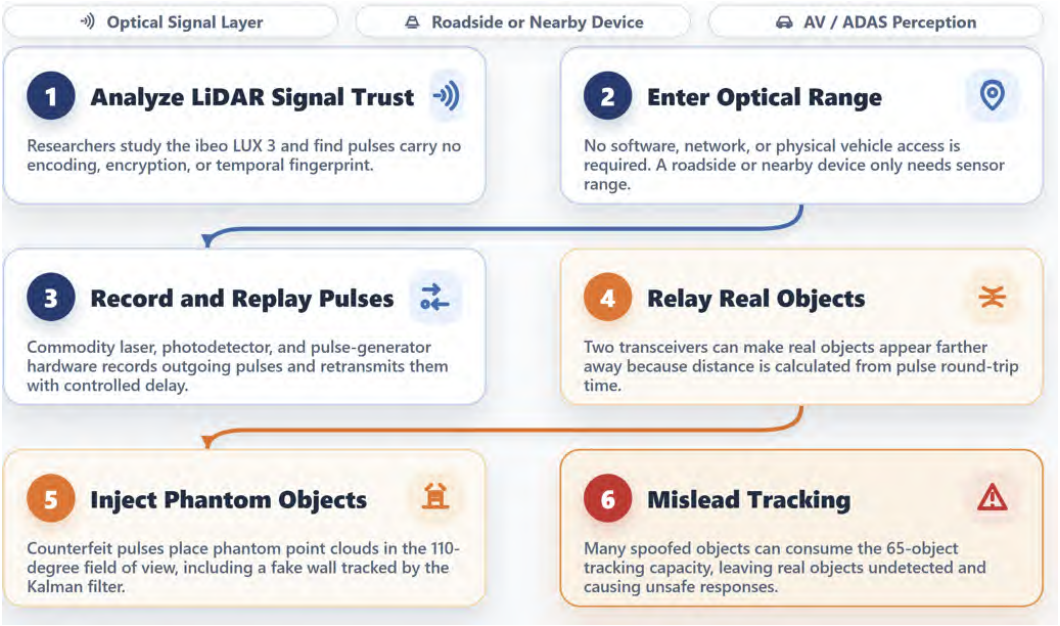


Figure 9.4: Replay of unauthenticated time of flight pulses to shift real objects and exhaust tracking capacity

Note

Sources:

- `https://blackhat.com/docs/eu-15/materials/eu-15-Petit-Self-Driving-And-Connected-Cars-Fooling-Sensors-And-Tracking-Drivers-wp1.pdf` [24]
- `https://blackhat.com/docs/eu-15/materials/eu-15-Petit-Self-Driving-And-Connected-Cars-Fooling-Sensors-And-Tracking-Drivers.pdf` [24]
- `https://www.youtube.com/watch?v=C29UGFsIWVI` [24]

Discovery and reconnaissance

The researchers' investigation started from the observation that the primary environmental sensors in autonomous vehicle platforms of that period, specifically LiDAR systems, operated

on principles that provided no built-in integrity protection. The underlying reasoning was straightforward: if a self-driving vehicle receives corrupted sensor inputs, it will correspondingly make poor driving decisions.

Their target was the *ibeo LUX 3 LiDAR unit*, which is a commercially available sensor used in urban traffic and highway driving scenarios. The researchers found that the ibeo LUX 3 system used pulses carrying no encoding, encryption, or temporal fingerprint. This meant that a device capable of receiving and retransmitting those pulses with a controlled delay could convincingly simulate an object at an arbitrary distance.

Initial access

The attack required no access to the vehicle's internal systems, network interfaces, or software. The attacker only needed to be within range of the target LiDAR sensor, with the transmitting laser hardware capable of operating at distances of up to 100 meters, from positions in front of, to the side of, or behind the sensor, and without requiring precise aim at the unit.

The attack device consisted of a laser diode, a photodetector receiver, and two pulse generators, assembled from commodity components at a total cost of under 60 US dollars. Because the pulses emitted by the ibeo LUX 3 carried no encoding or encryption, the researchers were able to record them and replay them with a controlled delay. The only technically demanding aspect of the setup was synchronization, specifically ensuring that the replayed signal was fired back at the sensor within the correct timing window.

Exploitation

Once the recording step was complete, the researchers demonstrated two classes of attack: a **relay attack** that manipulated where real objects appeared to be, and **spoofing attacks** that injected phantom objects that did not exist. Both exploit the same fundamental weakness, namely that the ibeo LUX 3 pulses carry no encoding or authentication, and any device capable of receiving and retransmitting them can influence what the sensor perceives.

In the **relay attack**, two transceivers were positioned relative to the ibeo LUX 3, one to receive the outgoing LiDAR pulse and one to retransmit it from a different position with a controlled delay. Because the sensor calculates distance from the round-trip time of the pulse, the retransmitted signal caused the real wall at 1 meter to appear as false echoes at approximately 20 and 50 meters from the sensor. Notably, the relay attack does not require direct line of sight to the sensor, as reflected LiDAR pulses traveling past the unit can be intercepted and retransmitted from an adjacent position.

The **spoofing attack** extended this capability by using the original pulse as a trigger to generate counterfeit pulses at precisely controlled delays, placing phantom point clouds at chosen distances within the sensor's 110-degree field of view. The researchers demonstrated two spoofing variants. In the first spoofing attack, a single phantom wall was injected at

distances between 40 and 70 meters and successfully classified and tentatively tracked by the ibeo LUX 3's embedded **Kalman filter tracking system**. At the application layer, an automated driving system receiving such input would interpret the phantom as an obstacle and apply braking, though this consequence was not directly demonstrated in the laboratory setup. In the second spoofing attack variant, the researchers demonstrated that introducing spoofed objects caused the ibeo LUX 3 to misallocate its tracking resources, and noted that a sufficient number of spoofed objects could exhaust the sensor's capacity to simultaneously track up to 65 objects, potentially causing real objects in the environment to go undetected.

Note

Terms:

A Kalman filter tracking system is a recursive algorithm that estimates and refines the position, velocity, and trajectory of detected objects over time by combining motion predictions with incoming sensor measurements. In LiDAR-based perception, it is used to associate point cloud returns across successive scans, enabling the sensor to track moving objects consistently and determine their direction of travel.

Persistence

Because both the relay and spoofing attacks operate entirely at the physical layer through optical signal injection, they do not interact directly with the vehicle's software stack or network interfaces. As such, any entries produced in software logs would reflect only the sensor output, such as unexpected object detections or tracking anomalies, rather than the optical manipulation that caused them. Moreover, an attacker operating from a roadside position or from a vehicle traveling nearby could sustain either attack for as long as the target sensor remains within operating range. It is also worth noting that the attack device emits pulses in the near-infrared range at 905 nm, which are not visible to the human eye, and neither attack variant requires physical contact with the target vehicle. Therefore, an attacker positioned on the roadside or in an adjacent vehicle would not be easily visually detectable, allowing the attack to be sustained.

Impact

The implications of this work extend beyond a single product or platform. Sensor spoofing attacks use the same physical channel as the targeted sensor to manipulate its readings at the point of acquisition, before any digital processing occurs. Because the attack introduces no anomaly into the software stack or communications interface, the corrupted sensor output is indistinguishable from a legitimate measurement by the time it reaches the processing pipeline.

At the time of publication, no LiDAR manufacturer had implemented defenses against this class of attack. The research work established that the entire category of time-of-flight optical sensors was structurally vulnerable to replay and spoofing when emitted pulses carry no cryptographic identity or timing randomization. Subsequent generations of LiDAR systems have begun to address this through techniques such as pulse coding and randomized probing intervals, though the effectiveness of these countermeasures under active attack conditions remains an area of ongoing research. From an automotive cybersecurity standpoint, the researchers established an early threat model for LiDAR integrity attacks, which remains a reference for understanding how the physical sensing layer of autonomous vehicles and ADAS can be manipulated without access to software or communications infrastructure.

Case study 3: Zero-click remote code execution via TPM sensors

At Pwn2Own Vancouver 2024, security researchers demonstrated a vulnerability in the Tesla Model 3 that allowed attackers to execute arbitrary code on an ECU. The attack chain began with entry through the TPMS, where a weakness in the sensor setup process led to an out-of-bounds write. This flaw allowed the researchers to take control of a security-critical ECU responsible for vehicle access and startup functions. The research was subsequently presented at Hexacon 2024 in October of that year [25]. The exploit required no user interaction to achieve remote code execution on this access-control ECU. An overview of the attack steps for this case study is presented in *Figure 9.5*.

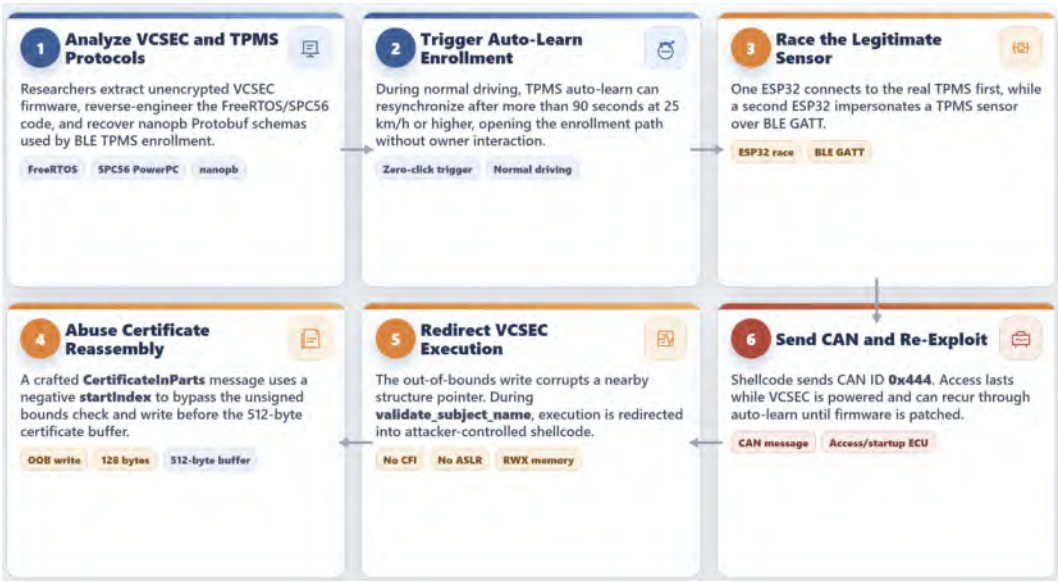


Figure 9.5: How TPMS autolearn and a certificate reassembly bug lead to malicious code execution

Note

Sources:

- `https://www.synacktiv.com/sites/default/files/2024-10/hexacon_0_click_rce_on_tesla_model_3_through_tpms_sensors_light.pdf` [25]
- `https://www.youtube.com/watch?v=R33cR3ZMTxM` [25]

Discovery and reconnaissance

The **Vehicle Controller Secondary** (**VCSEC**) ECU manages tire pressure data but is also responsible for a broader set of vehicle functions. It serves as the endpoint for the Tesla mobile application, handling vehicle access control and startup. To support these functions, the VCSEC communicates over multiple wireless interfaces, such as BLE for TPMS sensor data and Tesla app connectivity, and **Near-Field Communication** (**NFC**) for keyless vehicle access. The VCSEC also interfaces with the vehicle CAN bus, sending and receiving standard CAN messages and supporting **Unified Diagnostics Services** (**UDS**)-based provisioning and maintenance operations.

The researchers began their investigation by extracting VCSEC firmware images from the Tesla infotainment system's root filesystem, where firmware for multiple hardware revisions was stored without encryption. The VCSEC ECU itself runs FreeRTOS on an ST Microelectronics SPC56 PowerPC system-on-chip, and reverse engineering was performed using **IDA Pro** [26], which provides decompiler support for the PowerPC **Variable Length Encoding** (**VLE**) instruction set used by the SPC56. Two standard libraries were identified in the firmware: **Mbed-TLS**, which handles **Transport Layer Security** (**TLS**) and X.509 certificate operations, and **nanopb** [27], a Protocol Buffer implementation designed for embedded systems with constrained memory. Both are relevant to the attack chain discussed in the following subsections.

Since the TPMS-to-VCSEC communication is based on **Protocol Buffer** (**Protobuf**), a binary serialization format that cannot be interpreted without the underlying message schema, recovering the Protobuf message definitions from the firmware was a prerequisite for understanding and reconstructing the messages exchanged during sensor enrollment. The researchers used the **anvilsecure/nanopb-decompiler** [28] tool for this purpose. Because the SPC56 is a big-endian processor and the tool was originally designed for little-endian targets, the researchers patched it to support big-endian encoding. The TPMS sensors communicate with the VCSEC over BLE using **Generic Attribute Profile** (**GATT**) characteristics, with

messages serialized using Protobuf. It is worth noting that the **.proto** schema governing both TPMS and Tesla application communication had already been partially extracted from the Tesla mobile application by the research community, which provided a useful reference for understanding the protocol structure. For example, the **TeslaProtobufs** project [29] on GitHub, which includes the `vcsec.proto` file, is one such example.

Initial access

Before the VCSEC can accept data from a TPMS sensor, the sensor must complete an enrollment process. For type 5 TPMS sensors, this enrollment additionally requires certificate-based authentication to verify the sensor's legitimacy, during which the VCSEC requests an X.509 certificate from the TPMS. This certificate handling capability was not yet deployed in production sensors at the time of the research, meaning the researchers had to emulate a type 5 TPMS sensor to trigger the vulnerable certificate reassembly code path.

The certificate reassembly handler within the VCSEC firmware accepted a `CertificateInParts` Protobuf message comprising three fields: `startIndex`, which specifies the byte offset within the global certificate buffer at which each incoming fragment is to be written; `certificateSize`, which specifies the total byte length of the certificate being assembled; and a **data buffer** with a maximum size of 128 bytes constrained by `nanopb`. The bounds check computed an unsigned 32-bit sum of startIndex and data_size and compared the result against the certificate buffer boundary of 512 bytes. The researchers identified that a carefully chosen negative value for `startIndex` caused the unsigned addition in the bounds check to overflow, producing a sum small enough to satisfy the bounds check. The subsequent `memcpy` would then use startIndex as a signed offset into the certificate buffer, writing up to 128 bytes of attacker-controlled data to memory locations preceding the buffer. This constituted an **out-of-bounds write** vulnerability with up to 128 bytes of attacker-controlled data.

Exploitation

A pointer to a structure containing a function pointer was located just before the global certificate buffer in memory. By overwriting this structure pointer to redirect it toward the attacker-controlled buffer, and then placing the shellcode payload in that buffer, the researchers were able to redirect execution during the subsequent certificate validation call to **validate_subject_name**, at which point the VCSEC jumps directly into the attacker's shellcode, as illustrated in *Figure 9.6*:

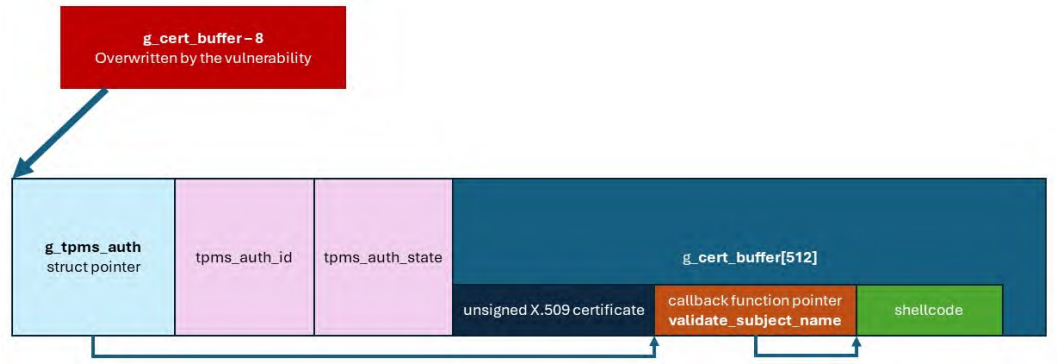


Figure 9.6: Exploitation flow - overwriting the structure pointer before g_cert_buffer redirects execution to attacker-controlled shellcode [adapted from 16]

The VCSEC firmware provided no **Control Flow Integrity** (**CFI**) and no **Address Space Layout Randomization** (**ASLR**) protection, and its **Memory Management Unit** (**MMU**) / **Memory Protection Unit** (**MPU**) were unconfigured, leaving all memory regions simultaneously readable, writable, and executable. The shellcode was compiled from C code using a **powerpc-eabivle** toolchain and called firmware functions at their known addresses.

TPMS sensor enrollment in VCSEC can be initiated through a UDS session. However, because the researchers aimed to demonstrate zero-click exploitation, that is, an attack requiring no interaction from the vehicle owner and no access to a diagnostic interface, they examined Tesla's TPMS **auto learn** mechanism as an alternative means of triggering the vulnerable enrollment code path. When a Tesla vehicle satisfies the criteria for auto learn, that is, traveling for more than 90 seconds at a speed of at least 25 km/h, the VCSEC and TPMS attempt to resynchronize, creating an opportunity to introduce a rogue TPMS sensor into the enrollment process.

The attack scenario used two BLE devices. The first, an ESP32 development board, was programmed to connect to a legitimate TPMS sensor before the VCSEC could do so, effectively denying the VCSEC access to that sensor during the auto learn phase. The second ESP32 was configured as a TPMS sensor simulator, responding to VCSEC enrollment messages and delivering the malicious **CertificateInParts** payload with the crafted **startIndex** value to trigger the out-of-bounds write. An earlier implementation using **BlueZ** was found to be too slow to win the connection race reliably, which led the researchers to implement the connection logic directly on the ESP32 hardware, achieving a consistently higher success rate.

Upon successful exploitation, the shellcode transmitted an arbitrary CAN message on CAN ID 0x444 across the vehicle CAN bus, demonstrating that an attacker impersonating a TPMS

sensor over BLE, without physical access to the vehicle or its diagnostic port, could inject messages onto the CAN bus controlling vehicle functions.

Persistence

Persistence in this attack operates at two levels. At runtime, shellcode executing on the VCSEC provides access to the vehicle CAN bus and BLE communications for as long as the ECU remains powered. Should the VCSEC restart, the shellcode is lost, but as long as the underlying vulnerability remains present in the firmware an attacker can exploit it again. The auto learn condition recurs naturally during normal vehicle operation, that is, whenever the vehicle travels for more than 90 seconds at a speed of at least 25 km/h, allowing the attacker to re-exploit the same vulnerability across repeated driving sessions without requiring physical access to the vehicle.

In more advanced scenarios, an attacker with sufficient shellcode capability could attempt to write persistent code to firmware flash storage, which would survive reboots and provide ongoing access to the CAN bus and the vehicle's BLE access control functions. The firmware's lack of CFI, ASLR and MMU/MPU protections would offer no resistance to such an attempt.

Impact

VCSEC is a security-critical ECU that manages vehicle access control and grants authorization for vehicle startup. With access to the vehicle CAN bus, an attacker achieving code execution on the VCSEC can send arbitrary CAN messages and directly affect vehicle functions.

It is noteworthy that the zero-click nature of the exploit substantially lowers the barrier to attack. No interaction from the vehicle owner is required, and the trigger conditions, that is, the vehicle traveling for more than 90 seconds at a speed of at least 25 km/h, occur naturally during normal operation. The attack hardware consists of two commercially available ESP32 development boards and standard BLE components, allowing the setup to be assembled at low cost and concealed in a roadside device or a vehicle traveling in proximity to the target.

Tesla addressed the vulnerability following coordinated disclosure by releasing a firmware update that remediated the integer overflow and introduced additional validation checks on related variables.

Addressing the common security weaknesses that impact vehicle sensors and AI/ML systems requires the same process rigor that we apply against any vehicle component. While it is important to address these weaknesses systematically, knowledge of security best practices that help eliminate such weaknesses is still useful, which is what we will do next.

Best practices for securing vehicle sensors and AI systems

The weaknesses and penetration testing techniques described in this chapter point toward a set of high-level principles that should guide the secure design, integration, and operation of vehicle sensors and AI systems. These principles are intended to be applicable across the development lifecycle and across the range of sensor and AI technologies described in this chapter, rather than prescribing specific technical implementations for each sensor type.

Applying defense-in-depth across the sensor and AI stack

It is imperative to consider that no individual sensor or AI component should be treated as the sole source of truth for safety-critical decisions. The vehicle's perception and planning architecture should be designed so that an attack against any single sensor interface, such as a radar spoofing attack that injects phantom obstacles, does not cause the vehicle to take an unsafe action without corroborating evidence from other sensing inputs. **Sensor fusion** logic should be designed to detect and flag anomalies that are inconsistent across sensor modalities, and safety-critical decisions should require agreement across multiple independent sources where feasible.

This principle applies equally to AI inference outputs. A perception model's classification of a traffic sign, an object, or a road user should be treated as a probabilistic input to the decision pipeline, not as ground truth. The vehicle's overall architecture should include monitoring logic that can identify situations where the AI system's outputs are implausible given the available physical context. As such, applying cross-sensor consistency checks as defenses, such as comparing radar object lists against LiDAR point clouds and camera object detections, is among the most practical approaches against sophisticated spoofing. These sensor fusion checks are appropriate because many spoofing techniques are most effective when applied against one sensing modality at a time.

Furthermore, within the HPC platform, sensor ingestion paths should be treated as high-risk software boundaries regardless of the underlying sensor type. The software component responsible for receiving and normalizing raw sensor data should operate with minimal privilege and strict isolation from the broader application stack, validating authenticity and freshness before publishing data to consuming applications, so that a compromised or injected sensor stream cannot propagate directly into the perception pipeline without passing through a controlled validation point.

Authenticating sensor data at the interface level

Where the protocol and interface type supports it, sensor data should carry a **cryptographic authentication tag** that allows the receiving ECU to verify both the origin and the authenticity and freshness of the data. This applies directly to TPMS systems, where the addition of message authentication codes to transmitted packets would eliminate the spoofing and replay attacks described in the weaknesses section. For in-vehicle interfaces such as SerDes camera links, end-to-end authentication of the video stream should be evaluated during system design, with the performance overhead of authentication assessed against the latency requirements of the perception application. In addition to data authentication, the receiving ECU should verify the identity of the sensor module itself. Where supported, each sensor module should be provisioned with a unique credential, such as an X.509 certificate or a symmetric key, that the receiving ECU can challenge during initialization to confirm that the connected device is a genuine and authorized sensor module. This **sensor identity authentication** helps mitigate the risk of counterfeit or cloned sensor replacement, where a device presenting a valid electrical interface produces fabricated or manipulated data that the receiving ECU would otherwise accept without question.

For network-attached sensors, the authenticated communication practices described in *Chapter 6* can be applied. **AUTomotive Open System ARchitecture** (**AUTOSAR**) **Secure Onboard Communication** (**SecOC**) [30] provides a well-defined mechanism for adding message authentication and freshness information to CAN and CAN-FD traffic. For higher-bandwidth sensors using Automotive Ethernet, **Media Access Control Security** (**MACsec**) [31] provides link-layer confidentiality, integrity, and data-origin authentication. It is important to note that MACsec is link-layer protection only; once frames traverse switches and are processed hop-by-hop, end-to-end integrity from the sensor's output to the perception consumer requires additional application-layer controls.

For sensor interfaces where cryptographic authentication is not feasible due to the unidirectional or constrained nature of the communication, the receiving system should apply **plausibility checking** as a compensating control. Readings that fall outside physically reasonable ranges or that deviate sharply from recent historical values without a corresponding change in vehicle dynamics should be flagged for further evaluation rather than consumed without validation.

Designing AI systems with adversarial robustness in mind

Neural network models deployed in the vehicle perception stack should be evaluated for adversarial robustness as part of the development and validation process, not only for accuracy on clean inputs, but testing could also cover a large set of representative misuse cases. Adversarial training techniques, in which the model is trained on a mixture of clean and

maliciously perturbed inputs, can improve robustness against known attack categories. Using multiple independently trained models to cross-check each other's outputs provides an architectural safeguard against adversarial inputs that exploit the decision boundaries of a single model. It is important to note that where the sensor data path feeds an ML perception model, the perception pipeline should be treated as part of the sensor's attack surface. This means that fuzzing the data path and evaluating robustness against **distributional shift** should be included as part of the security testing scope for that sensor.

Protecting in-vehicle AI assistants against injection and misuse

In-vehicle LLM-based assistants should operate under a clearly defined and enforced permission model that determines which vehicle functions the assistant may invoke and under what conditions. Permissions should be granted following the **principle of least privilege**, and the assistant should not have the authority to trigger safety-relevant vehicle functions through conversational commands alone without an explicit confirmation step.

The data sources that an in-vehicle assistant is permitted to ingest, such as navigation data, calendar entries, and connected service responses, should be treated as untrusted inputs and subject to content validation before being incorporated into the model's prompt context. It is important to note that where the assistant processes external content, technical controls should be in place to prevent instruction-like content in that data from overriding the assistant's system-level directives.

Personal data processed by in-vehicle AI assistants should be subject to a clearly documented data minimization and retention policy. Voice recordings, navigation history, and conversational logs should be retained only for the minimum period necessary to serve the function for which they were collected and should be protected by appropriate encryption and access controls both in transit and at rest.

Addressing sensor security throughout the development lifecycle

Sensor and AI security considerations should be integrated into the vehicle's cybersecurity management system from the earliest stages of development and follow best practices for secure software development lifecycle [32], consistent with the requirements of ISO/SAE 21434 [33]. **Threat Analysis and Risk Assessment** (**TARA**) for sensor and AI components should explicitly consider the attack categories described in this chapter, including physical signal manipulation, adversarial inputs, and AI assistant exploitation. The resulting risk assessment should drive security requirements that are applied through design, implementation, verification, and post-production monitoring.

Penetration testing against sensor and AI systems should be included in the vehicle's cybersecurity validation activities, with a scope that covers both individual sensor components and the integrated perception and AI stack.

In addition, post-production monitoring should operate at two levels. **Runtime anomaly detection** deployed on the vehicle monitors sensor and perception system outputs and raises alerts when behavior deviates from expected operational boundaries, such as sudden implausible object detections or objects appearing without a corresponding change in vehicle dynamics. These alerts should be collected and analyzed within the vehicle manufacturer's **Vehicle Security Operations Center** (**VSOC**), where **fleet-wide correlation** can identify patterns such as a recurring anomaly across multiple vehicles that may indicate a widespread attack or an actively exploited vulnerability in a deployed sensor technology.

Moreover, supplier engagement is essential, given that sensors and AI components are typically procured from specialized suppliers rather than developed in-house by the vehicle manufacturer. Cybersecurity requirements for sensors and AI components should be defined in the relevant supplier agreements as part of the dedicated cybersecurity interface agreements, and the vehicle manufacturer should verify through appropriate assessment activities that those requirements are met before the components are integrated into the vehicle.

Finally, the AI development supply chain should be subject to the same integrity controls applied to software supply chains. This includes ensuring that AI model files and training datasets are obtained from verified sources, stored with appropriate access controls, and validated against cryptographic signatures before deployment. It is worth mentioning that training datasets are a particular concern, as an attacker who is able to introduce malicious samples into the training data, that is, a **data poisoning attack**, can influence model behavior in ways that are difficult to detect through standard functional testing. Furthermore, the OTA update process for AI models should be treated with the same rigor as the OTA update process for vehicle software, including appropriate authenticity verification, rollback protection, and staged deployment, that is, rolling out updates to a limited subset of vehicles first before committing to the full fleet, where this is technically feasible.

Summary

Vehicle sensors and AI systems extend the vehicle's attack surface beyond the in-vehicle network into the physical sensing domain and the AI inference layer. As this chapter has shown, weaknesses in this space arise not only from implementation errors but from fundamental physical and architectural properties: sensors that cannot authenticate the signals they receive, interfaces that carry no cryptographic protection, perception models that are susceptible to adversarial inputs, and AI assistants that cannot reliably distinguish instructions from data.

This chapter examined these weaknesses across a range of sensor and AI technologies and mapped them to penetration testing techniques organized around discovery, vulnerability analysis, and exploitation phases. To better illustrate how these techniques apply in practice, the chapter also presented three case studies demonstrating how sensor and AI weaknesses can be exploited to affect vehicle behavior or compromise adjacent systems.

The best practices discussed reflect a defense-in-depth approach that treats every sensor input as untrusted until verified, applies cryptographic authentication at the interface level where feasible, and integrates sensor and AI security into the development lifecycle from the earliest design stages. Post-production monitoring through the vehicle manufacturer's VSOC provides a further layer of detection once the vehicle is deployed.

In the next chapter, we look ahead at the evolving threat landscape for connected and automated vehicles and discuss how the automotive industry can continue to advance its cybersecurity approach in response to emerging challenges.

References

[1] H. Khayyam, B. Javadi, M. Jalili, and R. N. Jazar, "*Artificial Intelligence and Internet of Things for Autonomous Vehicles*," in Nonlinear Approaches in Engineering Applications: Automotive Applications of Engineering Problems, Springer, 2020, pp. 39-68.

[2] M. Girdhar, J. Hong, and J. Moore, "*Cybersecurity of Autonomous Vehicles: A Systematic Literature Review of Adversarial Attacks and Defense Models*," IEEE Open Journal of Vehicular Technology, vol. 4, pp. 417-437, 2023.

[3] Z. Saeed, M. Masood, and M. U. Khan, "*A Review: Cybersecurity Challenges and Their Solutions in Connected and Autonomous Vehicles (CAVs)*," JAREE, vol. 7, no. 1, 2023.

[4] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar, "*Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study*," in Proc. 19th USENIX Security Symposium, Washington, D.C., Aug. 2010.

[5] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "*Robust Physical-World Attacks on Deep Learning Visual Classification*," in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR), Jun. 2018, pp. 1625-1634.

[6] J. Malik, R. Muthalagu, and P. M. Pawar, "*A Systematic Review of Adversarial Machine Learning Attacks, Defensive Controls, and Technologies*," IEEE Access, vol. 12, pp. 99382-99421, 2024. `https://arxiv.org/pdf/2409.20426`

[7] N. Wang, S. Xie, T. Sato, Y. Luo, K. Xu, and Q. A. Chen, "*On the Realism of LiDAR Spoofing Attacks against Autonomous Driving Vehicle at High Speed and Long Distance*," in Proc. Network

and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2025. DOI: 10.14722/ndss.2025.230628.

[8] OWASP, "*OWASP LLM AI Security and Governance Checklist -- LLM Top 10 for Large Language Model Applications*," OWASP Foundation, 2025. [Online]. Available: https://genai.owasp.org

[9] A. Esnaashari, J. Johansson, and M. Lauber, "*Security Risks of Agentic Vehicles: A Systematic Analysis of Cognitive and Cross-Layer Threats*," arXiv:2512.17041, Dec. 2025. https://arxiv.org/abs/2512.17041

[10] RTL-SDR Blog, "*RTL-SDR Blog V3 Dongles User Guide*," [Online]. Available: https://www.rtl-sdr.com/rtl-sdr-blog-v3-dongles-user-guide/

[11] Great Scott Gadgets, "*HackRF One*" [Online]. Available: https://greatscottgadgets.com/hackrf/one

[12] merbanan, "*rtl_433: Program to Decode Radio Transmissions from Devices on the ISM Bands (and Other Frequencies)*," GitHub. [Online]. Available: https://github.com/merbanan/rtl_433

[13] J. Pohl, "*Universal Radio Hacker (URH): A Complete Suite for Wireless Protocol Investigation with Hardware Support*," GitHub. [Online]. Available: https://github.com/jopohl/urh

[14] GNU Radio, "*GNU Radio: An Open-Source Software-Defined Radio Framework*," [Online]. Available: https://www.gnuradio.org

[15] B. Bloessl, "*gr-tpms: GNU Radio Module for Receiving and Decoding TPMS Signals*," GitHub. [Online]. Available: https://github.com/bastibl/gr-tpms

[16] Great Scott Gadgets, "*Ubertooth One: An Open-Source Bluetooth Experimentation Platform*," GitHub. [Online]. Available: https://github.com/greatscottgadgets/ubertooth

[17] Wireshark Foundation, "*Wireshark: Network Protocol Analyzer*," [Online]. Available: https://www.wireshark.org.

[18] M. Nicolae et al., "*Adversarial Robustness Toolbox v1.0.0*," arXiv:1807.01069, 2019. [Online]. Available: https://github.com/Trusted-AI/adversarial-robustness-toolbox

[19] J. Rauber, R. Zimmermann, M. Bethge, and W. Brendel, "*Foolbox Native: Fast Adversarial Attacks to Benchmark the Robustness of Machine Learning Models in PyTorch, TensorFlow, and JAX*," Journal of Open Source Software, vol. 5, no. 53, p. 2607, 2020. [Online]. Available: https://github.com/bethgelab/foolbox

[20] can-utils, "*can-utils: Linux CAN Utilities Including candump, cansend, and cangen*," GitHub. [Online]. Available: https://github.com/linux-can/can-utils

[21] N. Carlini and D. Wagner, "*Audio Adversarial Examples: Targeted Attacks on Speech-to-Text*," IEEE Security and Privacy Workshops, 2018.

[22] Tencent Keen Security Lab, "*Experimental Security Research of Tesla Autopilot*," Technical Whitepaper, Mar. 2019. [Online]. Available: https://keenlab.tencent.com/en/whitepapers/Experimental_Security_Research_of_Tesla_Autopilot.pdf

[23] S. Nie, L. Liu, Y. Du, and W. Zhang, "*Over the Air: How We Remotely Compromised the Gateway, BCM, and Autopilot ECUs of Tesla Cars*," Black Hat USA 2018, Las Vegas, NV, Aug. 2018. [Online]. Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Liu-Over-The-Air-How-We-Remotely-Compromised-The-Gateway-Bcm-And-Autopilot-Ecus-Of-Tesla-Cars-wp.pdf, https://www.youtube.com/watch?v=N2uD1PoHaUE

[24] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "*Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and LiDAR*," Black Hat Europe, Amsterdam, Netherlands, Nov. 2015. [Online]. Available: https://blackhat.com/docs/eu-15/materials/eu-15-Petit-Self-Driving-And-Connected-Cars-Fooling-Sensors-And-Tracking-Drivers-wp1.pdf,https://blackhat.com/docs/eu-15/materials/eu-15-Petit-Self-Driving-And-Connected-Cars-Fooling-Sensors-And-Tracking-Drivers.pdf,https://www.youtube.com/watch?v=C29UGFsIWVI

[25] D. Berard and T. Imbert, "*0-click RCE on Tesla Model 3 through TPMS Sensors*," Hexacon 2024, Paris, France, Oct. 2024. [Online]. Available: https://www.synacktiv.com/sites/default/files/2024-10/hexacon_0_click_rce_on_tesla_model_3_through_tpms_sensors_light.pdf

https://www.youtube.com/watch?v=R33cR3ZMTxM

[26] Hex-Rays, "*IDA Pro: Interactive Disassembler and Decompiler*," [Online]. Available: https://hex-rays.com/ida-pro

[27] jpa, "*nanopb: Protocol Buffers for Embedded Systems*," [Online]. Available: https://jpa.kapsi.fi/nanopb

[28] anvilsecure, "*nanopb-decompiler: Tool for Extracting Protocol Buffer Message Definitions from nanopb-Based Firmware*," GitHub. [Online]. Available: https://github.com/anvilsecure/nanopb-decompiler

[29] acvigue, "*TeslaProtobufs: Source .proto Files for Controlling Tesla Vehicles over the New API*," GitHub. [Online]. Available: https://github.com/acvigue/TeslaProtobufs

[30] AUTOSAR, "*Specification of Secure Onboard Communication (SecOC)*," AUTOSAR Classic Platform, Release 23-11, 2023. [Online]. Available: `https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_SWS_SecureOnboardCommunication.pdf`

[31] IEEE 802.1AE-2018, "*IEEE Standard for Local and Metropolitan Area Networks -- Media Access Control (MAC) Security*," IEEE, New York, NY, USA, 2018.

[32] D. K. Oka, "*Building Secure Cars: Assuring the Automotive Software Development Lifecycle*," Wiley, 2021. ISBN: 978-1-119-71074-5. DOI: 10.1002/9781119710783.

[33] ISO/SAE 21434:2021, "*Road Vehicles -- Cybersecurity Engineering*," International Organization for Standardization and SAE International, Geneva, Switzerland, 2021.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

10

A Path Forward

Throughout this book, we have explored a wide array of attack methods targeting the various components and interfaces of the modern vehicle, alongside the best practices necessary to mitigate them. However, as vehicle architectures evolve into **software-defined vehicles** (**SDVs**) centered around **high-performance computing** (**HPC**) platforms, an ad-hoc approach to security, often characterized as "patch and pray," is no longer sustainable. This reactionary mindset, where vulnerabilities are only addressed as they become imminent, offers no provable assurance that a system can remain resilient against increasingly sophisticated threats.

In this chapter, we pivot from the offensive domain to the defensive domain, emphasizing a systematic managed security approach. At the heart of this transition is the **cybersecurity management system (CSMS)**, while acknowledging the role of Generative AI in accelerating and widening the scope of compliance. We will move beyond the "checkbox" compliance exercise to explore a secure-by-design philosophy that integrates security across the entire product lifecycle, from initial planning and **threat analysis and risk assessment** (**TARA**) to secure production and continuous post-deployment monitoring.

Furthermore, we must acknowledge that we are in a race against an evolving adversary. The emergence of AI-augmented offensive agents has fundamentally altered the attacker's cost model, shifting the bottleneck from human expertise to computational limits. To win this race, defenders must also leverage Generative AI to automate defensive tasks, close the "automation gap," and strive for vulnerability elimination rather than mere mitigation. By establishing a synergy between offensive insights and defensive rigor, we can build a resilient future for the next generation of mobility.

In the rest of this chapter, we will cover the following main topics:

- Ad-hoc security approach
- Managed security approach
- Raising the security bar in automotive systems
- Winning the race

Ad-hoc security approach

The **ad-hoc approach** to cybersecurity is typical of organizations that view security as a burden to be addressed only when a crisis emerges. At its core, this is a reactionary mindset, often referred to as "patch and pray," where engineering teams constantly scramble to fix discovered weaknesses while hoping no further vulnerabilities are exploited. This method offers no provable security assurance and fails to remain resilient as vehicle architectures evolve. Organizations operating in this sphere typically fall into a spectrum of maturity.

Compliance-only mindset

On one side of the spectrum are the organizations that simply will do nothing for automotive cybersecurity unless they are pushed to do so by an explicit customer or regulatory requirement. Even then, they will only do the absolute minimum to fulfill the requirement without considering other aspects that may eventually violate the requirement. For example, an organization may decide to forego code signing and encryption of their software binaries for all customers by default. However, if a specific customer mandates code signing and encryption, then they will enable it through a configuration option. These organizations consider code signing and encryption to be features just like any other feature that could be turned on or off on demand.

But unlike normal features, shipping binaries with encryption turned off can have impacts on other members of the supply chain without their knowledge. Consider the case where a supplier ships common libraries, firmware components, or software modules across multiple ECUs or vehicle platforms. If attackers obtain an unencrypted binary from one OEM configuration, they may be able to reverse engineer shared code that also appears in protected builds delivered to other OEMs. In that case, the unprotected binary becomes a roadmap for understanding proprietary logic, identifying vulnerabilities, or developing exploits against systems where the binary itself is encrypted. The ad-hoc approach does not treat cybersecurity as a cross-cutting engineering concern with implications beyond a single product or customer, and this is one of its fundamental pitfalls.

PSIRT-only fallacy

At the other end of the spectrum are organizations that have a cybersecurity program that only addresses risks as they become imminent or proven through an active exploit. These organizations invest resources in a **Product Security and Incident Response (PSIRT)** program that actively looks for vulnerability reports and ensures such vulnerabilities are triaged and addressed. While having a mature PSIRT program to detect and patch vulnerabilities when they become known is certainly desirable, choosing to act only when cyber incidents occur reeks of reckless negligence.

Unlike IT systems where weekly or even daily patch events are deemed normal business, having to constantly patch automotive systems is neither practical nor ethical for that matter. For one, software updates require a rigorous testing process to ensure that the security fix does not introduce unforeseen quality and safety issues. This process makes patching lengthier than a typical IT system. Some OEMs have limiting factors that make it impossible to release software updates faster than two months or even more from the time the issue is reported.

Also, since the software delivery depends on the OTA process, there can be cases where end customers will not receive the patch in their vehicles if they keep delaying the update or if they are out of the range of service. Above all, when an attack occurs that results in either loss of life or injury, the PSIRT process will never restore the human beings that were affected. Therefore, it goes without saying that focusing attention on responding to incidents is only marginally better than the other end of the spectrum where the absolute minimum is done for cybersecurity.

Why ad-hoc approach fails in the modern era

As explored throughout this book, the ad-hoc approach is fundamentally broken due to several critical factors:

- **Exploding attack surfaces**: The transition from physically isolated ECUs to SDVs and HPC platforms has collapsed traditional hardware barriers. An ad-hoc approach often relies on security through obscurity, a myth dispelled by the ease of firmware extraction and reverse engineering seen in modern exploits.
- **Implicit trust weaknesses**: Ad-hoc security is unable to account for implicit trust claims within the system because it does not apply a rigorous threat modeling approach. This inevitably leads to untreated threats under the mantra of "ignorance is bliss". An example of this, which we saw in *Chapter 7*, is lack of consideration of peripheral attacks that abuse DMA engines through USB. Since the DMA engine is physically part of the system, there is a false assumption that it is inherently "benign" and requires less oversight which leads to the system exposure to attacks that abuse

permissive SMMU configurations to write into memory regions that are outside the intended range of that controller.

- **AI inflection point**: Perhaps the most significant threat to the ad-hoc model is the collapse of the attacker's cost model due to Generative AI. When exploit development is no longer expertise-bound but compute-bound, the window between a vulnerability being introduced and a weaponized exploit being available shrinks from months to days. A manual, reactive triage-and-patch cycle cannot converge fast enough to mitigate multi-agent offensive systems that reason about entire architectures to find exploit chains as we will see later in the chapter.
- **Supply chain exposure**: Organizations that use the ad-hoc method often lack a comprehensive **bill of materials** (**BOM**) for the software and hardware IP that goes into the final product. This includes third-party libraries, open-source components, commercial software modules, semiconductor IP, firmware blobs, and hardware components used in the vehicle. Without a reliable BOM, the organization may be blind to inherited vulnerabilities in the components it ships.

A separate but related risk comes from the development environment itself. Attackers may target compilers, build toolchains, SDKs, signing systems, CI/CD pipelines, package repositories, or developer workstations to compromise the software before it ever reaches the vehicle. In this case, the vulnerable element may not be part of the shipped product, but it can still corrupt the product during development or release. Both risks require visibility and control. If an organization cannot account for what it ships and cannot protect the tools used to build it, even a strong patching process becomes reactive and incomplete.

Signs of an ad-hoc culture

An organization is likely operating within the ad-hoc sphere if it subscribes to any of the following dangerous misconceptions:

- Security risks are only considered right before product launch or after a breach
- Cybersecurity is the sole responsibility of a "designated person" rather than the entire engineering team
- A third-party or in-house penetration test is sufficient proof of security
- The presence of a single hardware feature, such as an embedded HSM, automatically makes the ECU secure
- Applying a cybersecurity process is costly and time-consuming, plus the customer did not require it, so the organization's security responsibility should remain minimal

Ultimately, the ad-hoc approach assumes that attacker scaling is slower than defender scaling. In an era of autonomous vulnerability discovery pipelines and rapid exploitation design, this assumption is false, leaving the vehicle, and its occupants, unacceptably vulnerable.

Now, that we have studied the ad-hoc approach, we will switch our attention to the managed security approach which is meant to address the pitfalls mentioned above.

Managed security approach

At the core of this approach is the **cybersecurity management system** (**CSMS**), which governs cybersecurity across the product engineering lifecycle, as shown in *Figure 10.1*. A **managed security approach** is not just about adding secure software features such as encryption, secure boot, or access control. Regulations such as UNECE R155 and China's GB 44495-2024 require manufacturers to establish and demonstrate a lifecycle framework for managing cybersecurity risk. This means showing governance over risk assessment, requirements, design, implementation, verification, production, operations, incident response, updates, and supplier coordination:

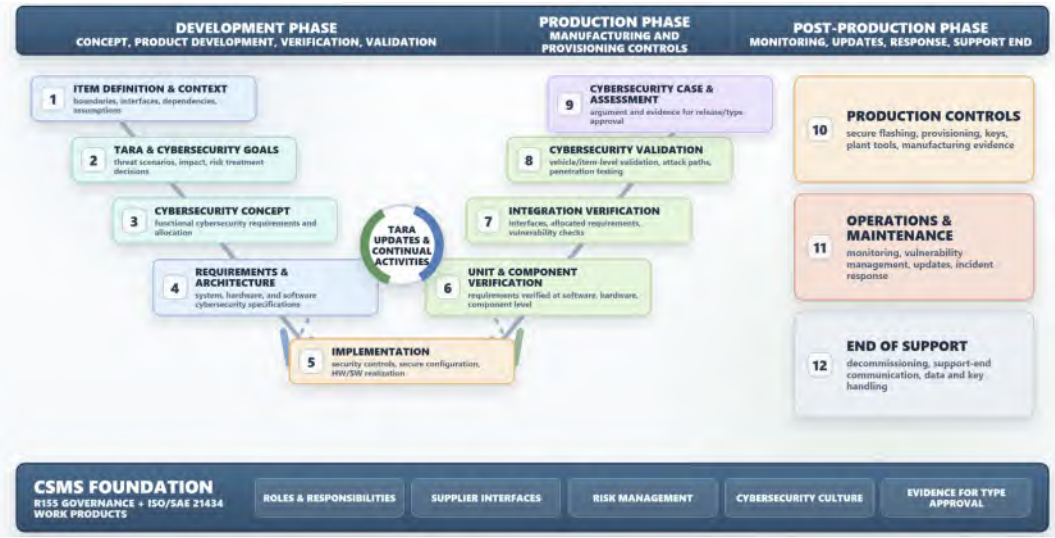


Figure 10.1: Applying the cybersecurity management system to the full development lifecycle

Since automotive engineers are already familiar with V-cycle, CSMS can be viewed as a cybersecurity governance overlay across each phase of development. ISO/SAE 21434 fits into this picture as the engineering standard that defines how cybersecurity activities should be performed throughout the vehicle lifecycle. In practice, organizations use ISO/SAE 21434 to structure the processes, work products, and evidence needed to support CSMS compliance under regulatory regimes such as UNECE R155 and GB 44495-2024.

Before we dive deeper into the CSMS, we must explore the role of existing engineering management systems that support the product lifecycle and preserve evidence of disciplined engineering work. These systems, such as requirements management, configuration management, change management, issue tracking, and verification management, are not cybersecurity-specific and should not be created as separate cybersecurity silos. Instead, they should be extended so that cybersecurity requirements, work products, decisions, and evidence are managed with the same rigor as safety, quality, and functional requirements. If these lifecycle systems are missing or weak, cybersecurity becomes difficult to scale, difficult to audit, and easy to bypass.

At a minimum, a managed approach typically presupposes the following:

- **Requirements management:** A requirements management system enables definition, baselining, and traceability, both to customer needs and to product/component requirements. In CSMS, traceability is not paperwork exercise, but a mechanism that allows an assessor to follow the thread from a risk treatment decision to an implemented control and its verification evidence.
- **Configuration management:** Cybersecurity relies on the integrity of both code and work products. Configuration management ensures that software, calibration/ configuration data, and supporting documentation are version controlled, reviewed, and traceable to engineering baselines. Without this discipline, the organization cannot reliably answer a basic assurance question: "*Which exact artifacts were assessed and released?*"
- **Documentation management:** The automotive cybersecurity standards are evidence driven. CSMS depends on controlled work products such as TARA records, cybersecurity requirements, interface agreements, verification results, and release readiness arguments. If documentation is not managed as an engineering artifact, the program cannot preserve its own security rationale over time.
- **Quality management**: When it comes to cybersecurity, a quality management system is a prerequisite, not an optional enhancement. It is especially important in modern automotive supply chains, where new entrants do not always arrive with the same level of process maturity expected from established automotive suppliers. Weak quality management does not merely result in lower quality products, it also causes requirements to be poorly captured, design intent to be incompletely reflected in implementation, and defects to escape into production systems under edge-case or adversarial conditions. The result is a wider space for attackers to find vulnerabilities. A buffer handling defect, a state machine inconsistency, or a feature that behaves differently than intended under malformed input may all begin as "quality issues"

before becoming security vulnerabilities. That is why a baseline level of quality discipline is necessary if the organization expects to build secure systems.

- **Tool management**: Tools are part of the product development lifecycle and therefore are part also of the attack space. If a toolchain is compromised, the attacker does not need to "hack the ECU" directly but rather influence what gets built, signed, and shipped. A managed security approach therefore treats tool selection, tool updates, tool access control, and tool provenance as security-relevant engineering decisions.

Among all the adjacent management systems, we will focus first on tool management due to its relevance to both defensive and offensive security.

Tool management and supply chain realities

Tool integrity matters because modern automotive development is an ecosystem of build systems, CI/CD pipelines, compilers, dependency managers, artifact repositories, code signing infrastructure, and test environments. A compromised tool or vendor can enable malicious code insertion or build-time manipulation that is difficult to detect downstream, especially when the resulting artifacts are legitimately signed and distributed. This is why supply chain attacks are not an edge case.

As secure development processes mature and products become more resilient to direct exploitation, adversaries have strong incentives to target vendors in the supply chain, build infrastructure, and tooling, where a single compromise can scale across many products and variants. For advanced persistent threat actors and nation-state operators in particular, this model is attractive because it bypasses hardened product interfaces and instead subverts the engineering path by which trusted software is produced.

The SolarWinds incident is commonly referenced because it demonstrated an uncomfortable truth: even sophisticated organizations can be compromised through trusted supply chain channels when toolchain safeguards and provenance controls are insufficient. The lesson is that vendor and tool trust must be governed by a strong process that is continuously validated. It is not enough to trust the output of a vendor; organizations must also assess the vendor's own security practices, software update hygiene, and ability to preserve build integrity over time.

Another aspect that is frequently overlooked is the number of target-side tools that come in contact with the system before it is released to production. In this context, target-side tools are utilities or software components that run on the target ECU, HPC, or embedded device during development, manufacturing, flashing, calibration, testing, or service, but are not intended to remain generally available in the final production system. Examples include flashing utilities loaded onto the target, factory test software, calibration helpers, diagnostic unlock tools, provisioning scripts, and temporary service applications.

Because these tools execute on the target system, they must be carefully controlled. If they are accidentally left behind, exposed to production images, or usable without proper authorization, they can become attack vectors against the final product. For example, consider a special factory test tool that is loaded on the target before the vehicle leaves the factory. If the tool is not signed, access-controlled, and cryptographically bound to an intended device or production stage, anyone who obtains a copy may be able to reuse it against a production system to aid reverse engineering, bypass normal controls, or alter device state.

Therefore, a managed approach expects tool management practices that address at least the following technical needs for both engineering-environment tools and target-resident tools. Note that engineering-environment tools run outside the target system, such as on developer workstations, build servers, CI/CD infrastructure, manufacturing stations, or service laptops:

- **Build integrity and traceability**: The organization must be able to prove what was built, from which sources, by which pipeline, using which tool versions, and under which approvals.
- **Access control and separation of duties**: Tooling and build systems must not become high-privilege shared infrastructure where compromise of a single account translates into compromise of production artifacts. Developer access, source repository administration, signing authority, and release approval should not collapse into the same trust boundary.
- **Dependency control**: Third-party dependencies and tool updates must be treated as controlled change, not as uncontrolled drift.
- **Code integrity checks and auditability**: Signed builds, integrity verification of produced artifacts, regular audits of build environments, and static analysis of changes and outputs all help detect abnormal modifications before they reach release readiness.
- **Vendor risk management**: Suppliers must be evaluated not only for functionality and cost but also for the maturity of their security practices. Trust in a tool vendor without continuous evaluation eventually becomes an unmanaged risk.
- **Tool binding to targets**: Code signing should not be limited to production software only. Any target-side tools that offer capabilities useful to an attacker during one of the device lifecycle states must be cryptographically signed while also incorporating a device-unique identifier to ensure that the tool can only be used on the specific target on which it was designed to run.
- **Evidence preservation**: When an incident occurs, the organization must be able to reconstruct the state of toolchains, dependencies, approvals, and produced artifacts.

Tip

Food for thought:

If an attacker can influence what gets built and signed, is there any value to defense-in-depth measures that the production software implements?

Acquisition and supplier management

Automotive supply chains are wide and deep. Whether you are building a single chip or a high-performance ECU, products routinely incorporate components and IP from many suppliers. In a managed approach, the system cannot be treated as secure if the security posture of supplied components is unknown or unmanaged.

This drives two requirements that are frequently underestimated:

- **Accounting for supplied components**: the program must establish visibility into what is supplied, how it is integrated, and which security assumptions are inherited
- **Defining responsibility distribution**: security responsibilities must be explicitly allocated across parties

A practical mechanism for this allocation is the **cybersecurity interface agreement**, which clarifies what each party provides and what each party assumes. During acquisition, suppliers must also be evaluated for cybersecurity capability, and security expectations, both from a process and technical point of view. This must be communicated early, including in RFQs where it is applicable, to avoid a case of picking a supplier solely on feature fit, cost, or delivery commitments. When supplied components are inventoried, responsibilities are assigned, and security expectations are verified early, adversaries have fewer unknown dependencies, inherited vulnerabilities, and integration gaps to exploit.

Planning

Planning cybersecurity activities is the first step in setting expectations for what the CSMS will do for a given product. The cybersecurity plan becomes the roadmap that drives execution as it defines activities, tailoring decisions, supplier and tool handling strategy, treatment of off-the-shelf components, and the security testing approach.

The plan also becomes an assessment anchor. When the product reaches readiness gates, the plan allows the organization to answer two decisive questions:

- What security activities were committed to?
- Were those activities executed with sufficient completeness and quality?

Here alignment matters because cybersecurity failures can usually be traced to process failures such as skipped cybersecurity activities, improper assignment of roles and responsibilities, lack of attention to peer reviews, missing or incomplete evidence about performing the work, or late discovery of high-impact risks without the proper planning for vulnerability and patch management.

Note

Best practice:

If the program intends to heavily tailor the cybersecurity process, have the tailoring rationale reviewed early before committing the organization to a plan that cannot survive an independent assessment at the end of the project. The worst outcome is to discover late in the lifecycle that aggressive tailoring created assessment findings severe enough to force rework just to avoid blocking release.

Integration with engineering lifecycle

Let us now take a closer look at how the CSMS integrates with the full engineering lifecycle as illustrated in *Figure 10.1*, starting with the TARA during the concept phase, through the maintenance phase, and all the way until end of life and decommissioning.

TARA

A hallmark of the managed approach is the role of threat modeling in controlling cybersecurity risk as features are introduced. Rather than waiting for incidents or test findings, TARA ensures that risk is identified, scored, treated, and documented before design decisions become expensive to change.

Engineers often raise legitimate concerns: threat modeling can be time-consuming, and outcomes can vary by team and method. These issues are real, but they do not invalidate the need. Threat modeling remains the most systematic mechanism for demonstrating that security risks were considered and addressed with traceable evidence. Certainly, many optimizations are possible to improve consistency and efficiency, but operating without TARA is effectively choosing to manage risk in a best effort manner. Without a systematic application of the threat analysis and risk management approach, it is impossible to prove completeness and correctness of the technical risk management activities.

In practice, TARA can be applied using different methods. The asset-driven approach as prescribed by ISO/SAE 21434 is widely used, while interaction-based approaches are selectively applied when system interactions and trust boundaries are the dominant risk drivers. The operational advantage of the asset-driven method is that new features can be analyzed by

mapping them to assets and their security properties, enabling reuse of existing threats and controls where appropriate.

Consider a feature added to record events in an **electronic data recorder** (**EDR**). If the recorded data includes camera frames that were previously identified as assets requiring confidentiality protection, then those confidentiality controls remain relevant and traceable. However, this feature may introduce additional properties that require protection, such as non-repudiation for legal or accountability reasons. This shifts risk treatment: the program must reduce the risk of forged camera frames by an attacker who gains access to the EDR data path or storage.

This is the practical value of TARA. It forces explicit analysis of impacted asset properties, records risk treatment decisions, and generates evidence that an independent assessor can review. In an ad-hoc approach, the risk of repudiation may be missed entirely or mitigated informally without durable evidence, leaving the program unable to justify its posture. Another benefit of TARA is that it provides a structured mechanism for ingesting new threats learned from incidents. This closes the loop between offensive discovery and defensive engineering. As attack techniques evolve and feasibility becomes better understood, TARA updates improve attack path modeling and risk scoring, while driving more targeted controls in subsequent product generations.

If TARA is not employed, the program is left with a binary choice: patch or do nothing, often without a coherent way to evaluate impact, justify a risk retention decision, or document why a vulnerability is not practically exploitable in a given system design. TARA therefore provides the missing evidence layer. It also allows the program to justify retained risk when appropriate. A cache-timing issue, for instance, may be assessed as not practically exploitable in a single-tenant design that does not retain secrets in shared caches because sensitive cryptographic operations are delegated to a hardware-protected security environment. Without TARA, that conclusion is merely opinion. With TARA, it becomes an assessable engineering argument.

Need for a cybersecurity concept

A thorough TARA should result in a high-quality cybersecurity concept that articulates how risks are reduced through technical security controls. The cybersecurity concept tells the security story that explains how security goals are achieved and why the chosen controls are sufficient for the identified risks. For a smart sensor handling camera frames, the cybersecurity concept may hinge on a small set of controls, such as supply chain management, identity provisioning and certificates, and sensor authentication protocol.

By articulating the assumed system context, the assumed operational environment and expected use cases that the product meets, the concept acts as the blueprint for which threats were considered during the analysis, what security goals the product must fulfill and most

importantly, the security controls and requirements that are needed to achieve those goals and reduce the overall technical risk to a reasonable level. The concept's value is that it ties these controls directly to the risks and assets identified in the TARA, demonstrating completeness of risk treatment.

The more feature-rich a product is, the more elaborate the security concept must be. For example, in an HPC platform, the cybersecurity concept typically spans several foundational controls such as the following:

- Secure boot and initialization
- Secure partitioning and isolation
- OS hardening with **identity and access management** (**IAM**)
- Secure communication
- Secure update management
- Cryptographic services and key management

Each control family traces up to a set of threats while tracing down to a set of technical security requirements that the product must fulfill to address those threats. In summary, the cybersecurity concept provides a coherent map from risk to control strategy, and it becomes the anchor for the cybersecurity specification work that follows.

Design and implementation

Products developed under a CSMS are expected to follow the principle of secure by design. Rather than bolting controls onto an insecure product after the fact, secure by design allocates controls deliberately during development as risks are uncovered and as architecture is defined. After the cybersecurity concept is established, cybersecurity specifications are derived to satisfy the concept. This requires allocating controls across architectural layers and deriving cybersecurity requirements for components. These requirements must then be reflected in both the software and hardware architecture artifacts so that implementation is constrained by design intent.

For example, a cybersecurity concept may specify that DNN models must be encrypted at rest using a device-unique key and an approved encryption algorithm. Realizing this requirement requires an architecture that separates key material from application logic and prevents plaintext model data from being exposed outside trusted boundaries.

The security requirement can be implemented through an architecture that integrates hardware-based key management, HSM-resident cryptographic services, secure storage, isolated memory, and controlled access by the inference hardware engine as shown in *Figure 10.2*. At the hardware layer, the embedded HSM securely stores and manages the device-unique key, ensuring that key material does not leave the HSM boundary in plaintext. The GuestOS

cryptographic library does not directly handle the key or perform the sensitive cryptographic operation itself; instead, it provides the API used by higher software layers to request cryptographic services from HSM.

In the application layer, a security manager coordinates security service requests, while the DNN manager governs the model lifecycle. When a DNN model must be loaded, the DNN manager invokes the security manager, which requests model decryption through the GuestOS cryptographic API. The request is forwarded to the HSM-resident cryptographic software, which uses the device-unique key to decrypt the model. The decrypted model is written directly into an isolated memory region that is accessible only to the inference hardware engine, not returned as plaintext to the general application layer.

In this way, the requirement is satisfied without exposing keys or decrypted model contents to untrusted software, preserving the secure-by-design intent across the hardware, operating system, and application layers as illustrated in *Figure 10.2*:

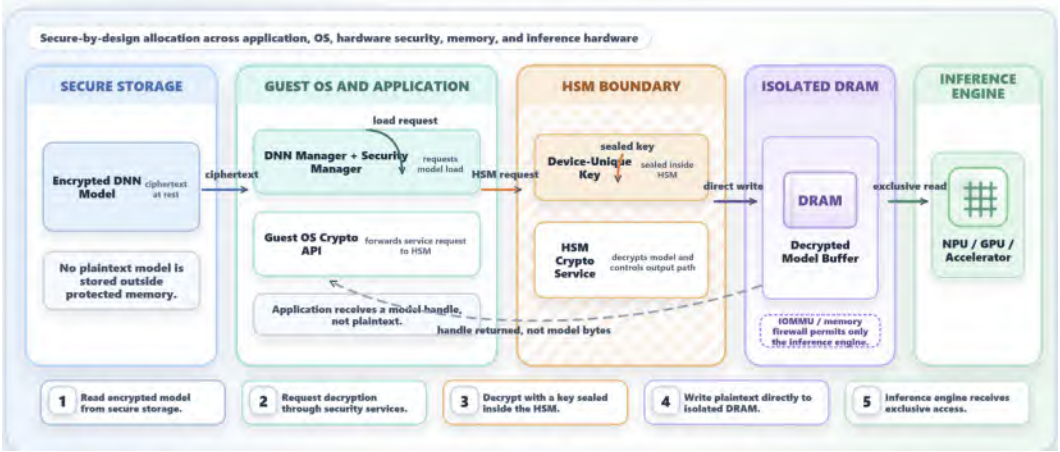


Figure 10.2: A secure architecture to ensure the DNN models are decrypted securely

Note

The architectural intent must be preserved as a traceable thread: **concept → requirement → design allocation → implementation evidence → verification results**. Without this thread, secure by design becomes an aspiration rather than an engineering property.

Security design principles and patterns

When attempting to satisfy security requirements for a specific architectural solution, engineers normally struggle with choosing the right architectural approach. A reliable way to strengthen security design is to start with security principles and secure design patterns and apply them consistently. NIST SP 800-160 is a common reference for design principles, but principles can feel abstract without being translated into engineering decisions.

For that reason, we will focus first on principles that most directly affect automotive attack surfaces and trust boundaries then switch gears to secure design patterns:

- **Domain separation**: Domain separation isolates subsystems with different security criticalities. For example, separating safety-critical systems like ADAS from non-critical systems like infotainment limits the risk of attacks spreading in a central computer design. Techniques such as hardware-based isolation, IP firewalls, and memory encryption ensure that a compromise in one domain does not affect others.
- **Least privilege**: The least privilege principle ensures that every system component, process, or user has only the permissions necessary to perform its specific function. For example, an infotainment application designed to play media should not have access to cryptographic keys used to generate authenticated in-vehicle messages, unlock diagnostic services, or authorize software update actions. If an attacker exploits that media app, those excessive privileges could allow them to send trusted messages, access protected diagnostic routines or interfere with update mechanisms. By restricting permissions, the compromise is more likely to remain localized, reducing the attacker's ability to pivot, escalate privileges, or affect safety-relevant functions.
- **Defense-in-depth**: Defense-in-depth involves deploying multiple layers of security measures to protect automotive systems from failures or attacks. When designing a control, it is important to layer the security measures to ensure that a single point of failure does not lead to a complete violation of the security goal.
- **Reduced complexity**: Complexity in automotive systems can lead to errors and security vulnerabilities. Simplifying system design reduces the likelihood of misconfiguration and enhances security. For example, minimizing the number of configuration options that apply to a security control reduces the likelihood that the wrong combination is applied which can degrade the security control or disable it completely.
- **Least functionality**: This principle emphasizes that system components should only include the features required for their specific roles. Excess functionality increases the risk of exploitation. For example, diagnostic services reserved for factory mode should

be disabled when the product transitions to production mode. Limiting functionality reduces the attack surface and simplifies security analysis.

- **Mediated access**: Mediated access ensures that all interactions with system components are controlled and authorized. This can be achieved using secure APIs, role-based access controls, and robust authentication mechanisms. For instance, in a multi-VM system, the hypervisor mediates access to shared memory between guest operating systems rather than relying on a lower privileged software to do that.
- **Secure by default**: Automotive systems should come preconfigured with secure defaults that prioritize protection. For example, if OTA package authentication is optional, then the default should be to enforce authentication rather than expect the user to enable it. Adopting protective defaults reduces the risk of vulnerabilities caused by improper configurations.

In addition to security design principles, security design patterns can be extremely useful to bring the concepts even closer to the problem space in which engineers are working. Having a set of design patterns that apply to a specific product can jump-start security design as engineers can easily identify a pattern that applies and check how well they applied it. Next, we'll survey some of the common **security design patterns** that apply to automotive systems.

Distrustful decomposition

If all functionality resides in a single software entity, a vulnerability in one part can compromise the entire system. To avoid this, partition the system into smaller components that mutually distrust each other. Each component enforces strict interface boundaries and operates with the minimal privileges required for its function.

For example, consider a multi-VM system in which each VM has access to a common storage device. Designing a dedicated storage server outside the boundary of all VMs ensures that no VM must trust that the other VM will not access its storage partitions. By decomposing the access management function of the storage partitions into a separate storage server with its own privileges and hardware assignment, we have successfully decomposed the function while maintaining the right level of storage access partitioning.

Chain of trust

A complex boot chain consists of multiple runtime execution environments with dedicated software images and separate signatures. If an untrusted component is allowed to boot or load software throughout this boot chain without verification, then malicious code can be executed before any higher-layer defenses become active. In such cases, start from a hardware-anchored root of trust such as the ROM code, an HSM, or a secure bootloader, and extend trust step by step to each subsequent software stage. Each stage verifies the authenticity and integrity of the next stage before transferring control for execution.

Let's consider an automotive HPC that boots a hypervisor, multiple guest operating systems, and AI perception applications as an example. If the boot ROM first verifies the primary bootloader, the bootloader verifies the hypervisor image, and the hypervisor verifies each guest VM image before launch, then no stage is executed unless the prior trusted stage explicitly approves it. In this way, trust is extended incrementally from immutable hardware-rooted code to the full HPC software stack, ensuring that a tampered perception VM cannot be launched simply because it resides on an otherwise valid storage device.

Trusted third party (TTP)

Separate components or services that do not directly trust each other still need a reliable way to authenticate identities and establish secure communication. Without a common trust anchor, each pairwise trust relationship becomes difficult to scale and manage. To tackle this, introduce a trusted third party that vouches for identities, issues credentials, or brokers trust relationships between the communicating entities.

For instance, consider a vehicle telematics ECU that must establish secure communication with the OEM backend, a fleet management platform, and a service tool used at dealerships. Rather than provisioning each endpoint with manually exchanged trust material for every other endpoint, the OEM can operate a certificate authority that signs the credentials for all approved parties. The telematics ECU then only needs to trust the OEM certificate authority, while each external service proves its identity through certificates issued by that authority. By introducing the trusted third party, the vehicle no longer depends on ad hoc bilateral trust relationships between every connected entity.

Identity provider (IdP)

When multiple services require user or device identity information, storing, and managing separate credentials in each service leads to duplication, inconsistent policy enforcement, and a larger attack surface. To avoid this, centralize identity management in an identity provider that authenticates the principal and issues tokens or assertions that other services can validate.

Consider an automotive HPC hosting infotainment applications, driver profile management, subscription-based feature control, and remote companion app integration. If each application stores its own username, password, or session state, then credential management becomes fragmented and error prone. By introducing a dedicated identity provider service within the vehicle-connected ecosystem, the driver authenticates once and receives a token that can be validated by the infotainment app, seat personalization service, and premium feature manager. In this way, identity proof is centralized while the relying services avoid maintaining separate credential stores.

Authenticator

If every subsystem implements its own authentication logic, then security behavior becomes inconsistent and mistakes in one subsystem can undermine the trustworthiness of the whole ECU or vehicle function. Centralizing authentication into a dedicated authenticator component that verifies credentials, certificates, tokens, or signed requests before other modules grant access will help avoid this.

Consider a zonal controller that supports local diagnostics, secure flashing, and maintenance actions from authorized service tools. If each diagnostic service implements its own certificate parsing and challenge-response logic, inconsistencies are likely to emerge. By introducing a dedicated authenticator service inside the ECU, all incoming diagnostic sessions are first routed through the same authentication component, which validates the tester certificate, checks authorization policy, and only then releases a session token to the requested function. This ensures that secure flashing, debug unlock, and configuration updates all depend on one hardened authentication path rather than several independently implemented ones.

Defer to kernel

Re-implementing security enforcement in application code often leads to inconsistent behavior, incomplete mediation, and avoidable implementation bugs. Application-layer security checks are easy to bypass if the underlying operating platform does not enforce them. To prevent this, delegate security-critical enforcement to the operating system kernel, hypervisor, or other low-level trusted platform mechanism that can enforce policy with stronger isolation and fewer opportunities for bypass.

Consider an automotive HPC running several Linux-based applications, including navigation, media playback, and vehicle settings. If each application independently decides which files, sockets, or shared memory regions it may access, a single coding error can expose sensitive assets. Instead, the platform can rely on kernel-enforced mechanisms such as process credentials, namespaces, mandatory access control, and hypervisor-managed memory partitioning. In this design, the navigation application does not "choose" to stay away from vehicle control data. The kernel and hypervisor ensure that it cannot access those resources even if the application is compromised. By deferring enforcement to the kernel and virtualization layer, security policy becomes harder to bypass and easier to reason about.

Privilege separation

When multiple functions run with the same high privilege level, compromise of one function gives an attacker unnecessary access to unrelated resources and services. To avoid this, split the system into separate processes or components with distinct privilege levels and abilities so that each function only receives the permissions necessary for its role.

Consider an MCU-based gateway ECU that supports packet routing, secure diagnostics, logging, and firmware update handling. If all these functions execute inside one privileged process, then a flaw in the logging parser may allow an attacker to gain the same authority as the firmware update logic. By separating these functions into distinct tasks with different privileges, the logging task can be restricted to write-only access to its buffers, while the update manager alone retains authority to modify flash memory, and the routing task can only access network forwarding interfaces. This way, compromise of one service does not automatically grant control over all security-sensitive ECU operations.

Secure strategy (policy factor)

Hardcoding one security policy into the system makes it difficult to adapt to changing regulatory requirements, cryptographic transitions, or market-specific deployment constraints. Encapsulating security policy choices into interchangeable strategies so that the system can switch between approved security policies without changing the surrounding business logic will help mitigate this issue.

Consider an OEM backend and in-vehicle telematics stack that must support one cryptographic profile for legacy deployments and another for future deployments using stronger algorithms and different certificate chains. If the communication stack embeds one fixed policy throughout the codebase, migrating to a new policy becomes risky and invasive. By isolating the certificate validation logic, key exchange method, and signature policy into separate strategy modules, the telematics application can select the approved policy for a given vehicle program or region while leaving the rest of the software unchanged. In this way, the security policy becomes configurable and governable rather than buried inside application logic.

Security builder factory

Complex subsystems often require many security-relevant parameters during construction. When these parameters are assembled manually, insecure combinations, and omitted protections become more likely. To avoid this from happening, use a builder pattern that constructs the subsystem step by step while enforcing required security choices and rejecting incomplete or unsafe configurations.

Consider an ADAS software component on an automotive HPC that must be configured with secure sensor channels, authenticated message sources, encrypted model storage, and approved inter-process communication paths. If developers instantiate this subsystem by manually setting dozens of configuration fields, it becomes easy to forget a required authentication mode or accidentally allow plaintext transport for a sensitive channel. By using a security builder factory, each construction step validates that secure options have been selected before the subsystem can be finalized. For example, the builder may refuse to complete the ADAS runtime unless sensor inputs are bound to authenticated sources and

model loading is bound to encrypted storage. This ensures that the resulting subsystem is assembled in a secure state rather than relying on developers to remember every required setting.

Secure chain of responsibility

A single security checkpoint is often insufficient when data or commands pass through multiple layers, networks, or ECUs. If intermediate stages do not apply their own checks, a message that was valid in one context may become dangerous in another. In such cases, apply a chain of responsibility in which each handler in the message path performs its own defined security action before passing the message to the next stage.

Consider a remote door unlock command originating from the OEM cloud and passing through the telematics ECU, central gateway, and body controller before actuating the lock module. If the system relies on only one security check at the entry point, then downstream assumptions may be violated by replay, routing abuse, or policy mismatch. By implementing a secure chain of responsibility, the telematics ECU first verifies the cloud message signature, the central gateway verifies whether the command is permitted in the current vehicle state and whether it originated from an authorized source domain, and the body controller finally checks local execution constraints before actuating the lock. Each stage adds its own security decision, ensuring that the final action depends on a chain of validated responsibilities rather than a single trust decision made upstream.

Secure implementation

Up to this point, the managed approach has defined security intent, translated that intent into requirements and architecture, and allocated controls across the system design. However, the implementation phase is where many vulnerabilities are actually introduced. This is true whether the product is primarily software, hardware IP, firmware, or a mixture of all three. If the managed approach stops at good architecture but does not constrain how implementation is performed, then the organization is simply producing well-documented insecure products.

This is why **secure implementation** must be treated as a governed engineering activity rather than as a matter of developer preference. At this stage, the goal is to reduce the probability that weaknesses are introduced while translating design into executable code, synthesizable hardware, configuration artifacts, and integration logic. A CSMS therefore expects implementation to be performed under defined coding rules, review discipline, and automated analysis.

In software, this typically means applying secure coding guidelines, performing peer review with security-relevant checks, and using static analysis tools to identify dangerous constructs and violations before they propagate downstream. In memory-unsafe languages, these practices are especially important because common implementation defects such as out-of-

bounds writes, use-after-free conditions, integer overflow chains, and race conditions can become direct exploit primitives. In higher-assurance contexts, organizations may also adopt memory-safe languages selectively for security-critical components to reduce entire classes of vulnerabilities by construction.

The same logic extends to hardware and low-level embedded development. A weak implementation process in RTL, firmware initialization code, secure boot logic, debug access control, or lifecycle state enforcement can undermine an otherwise sound cybersecurity concept. In fact, secure architecture does not survive careless implementation.

The broader point is that secure by design must be followed by secure by implementation. One defines what the product should do to resist attacks. The other governs how that design is realized without introducing unnecessary weaknesses.

Note

Best practice:

Treat secure coding rules, review checklists, and automated static analysis as mandatory engineering gates for security-relevant components, not optional quality improvements.

Another emerging factor is the growing use of AI-assisted development tooling. AI code generators can improve productivity, but they also change the implementation risk model. They can reproduce insecure patterns, misapply APIs, or generate superficially plausible but unsafe code paths when prompts are ambiguous or when the surrounding security assumptions are poorly stated. For that reason, AI-generated code should not be treated as trusted merely because it accelerates implementation. It must still be subjected to the same secure coding rules, review standards, and analysis expectations as manually authored code. In practice, the presence of AI in the toolchain strengthens rather than weakens the case for managed implementation controls.

At the same time, AI-assisted review and vulnerability discovery tools are increasingly becoming part of the expected implementation environment. These tools can reason across large codebases, identify suspicious code paths, correlate weaknesses with known exploit patterns, and assist analysts in proving exploitability. When used correctly, they can help organizations find implementation flaws earlier and with greater coverage. If used carelessly, they can also create a false sense of assurance. The managed approach therefore treats them as useful tools within a governed process, not replacements for engineering judgment or traceable evidence.

Verification testing

Continuing with the V-model, **verification** is the phase where the organization evaluates whether the product has been built correctly with respect to its specifications, design artifacts, and engineering rules. In cybersecurity, this means confirming that the security controls defined earlier in the lifecycle were implemented as intended and that the intermediate work products faithfully reflect the required design. The guiding question for verification is straightforward: "*Are we building the product right?*"

Verification is primarily about conformance. It asks whether requirements are correct, whether design artifacts properly realize those requirements, whether code and hardware implementations follow the design, and whether engineering rules and standards were followed during development. In other words, verification checks that the product being built matches the stated intent of the managed process. Typical verification activities include requirements reviews, architecture reviews, interface reviews, secure code reviews, static analysis, test-case reviews, integration testing, system testing, and consistency checks across work products. Verification begins early, where requirements are checked for correctness and completeness, architecture is checked against requirements, and implementation is checked against design and coding rules. It continues where test execution confirms that the integrated components and system satisfy the verified requirements and design intent.

For some high-assurance components, organizations may also adopt stronger forms of verification such as formal methods. These techniques do not replace conventional engineering review, but they can strengthen assurance for security-critical functions where ordinary testing cannot easily cover the full state space. The value of such methods is greatest where the cost of failure is high and where the security property can be stated precisely enough to be proven or exhaustively checked.

Validation testing

The final phase before release is **validation**. Whereas verification is concerned with whether the system was built according to specification, validation asks whether the integrated system achieves the intended cybersecurity goals in its operational context. The guiding question here is: "*Are we building the right product?*"

This distinction matters as a product may verify successfully against its specifications and still fail to provide meaningful real-world security if the specifications were incomplete, if the assumptions were unrealistic, or if integration effects introduce new attack paths. This is why ISO/SAE 21434 expects cybersecurity validation activities to be performed at the vehicle level, where components are integrated into the full product along with relevant backend and supporting environments.

Cybersecurity validation focuses on whether the system performs securely in practice against the threats it is expected to face. This includes dynamic testing of running systems, evaluation of attack surfaces in realistic configurations, and examination of whether the final system resists unauthorized access, tampering, data compromise, privilege escalation, and other relevant attacks.

Typical validation activities include penetration testing, fuzz testing, dynamic analysis, system-level abuse case evaluation, and operational testing against expected use conditions. Unlike verification, which often examines whether engineering artifacts conform to intent, validation stresses the actual product to determine whether the intended protection remains resilient to real-world attacks.

Naturally, penetration testing is an ideal method for security validation because it attempts to approximate the behavior of realistic adversaries within a bounded assessment scope. It helps determine not merely whether controls exist, but whether they meaningfully resist exploitation in the integrated environment. This is one of the reasons offensive security techniques are so valuable in a managed program: they supply the realism needed to challenge assumptions made earlier in the lifecycle.

Fuzz testing also plays an important role because many automotive attack surfaces are parser-driven, stateful, or exposed to malformed inputs across complex trust boundaries. Validation that ignores malformed-input behavior is incomplete. Dynamic analysis is similarly valuable because some weaknesses only become visible when the system is exercised under realistic runtime conditions.

In practice, verification and validation complement one another. Verification catches deviations from requirements, architecture, and coding rules before they spread. Validation ensures that the fully integrated product can withstand realistic threats and that the original cybersecurity goals are actually achieved.

Product release

Before a product is released, its cybersecurity readiness must be assessed to determine whether the initially planned risk reduction has been achieved with sufficient evidence. In a managed approach, this readiness decision should not be based on optimism or schedule pressure but rather should be supported by a cybersecurity case. The **cybersecurity case** is the structured argument for why the product is ready to be released to road users or upstream integrators. It ties together both the process-driven and the technical evidence demonstrating that cybersecurity risk has been managed to an acceptable level.

On the process side, the case shows that planned cybersecurity activities were performed and that the resulting work products are complete, consistent, and controlled. This includes references to artifacts such as the TARA, cybersecurity requirements, design records, interface

agreements, verification reports, validation reports, supplier evidence, and configuration-controlled release records. Any deviations are captured and subjected to a cybersecurity risk argument for why such deviations can be accepted for the given product context and assumptions of use.

On the technical side, the case demonstrates that identified deviations, weaknesses, and residual risks were not simply ignored. They were identified, analyzed, dispositioned, and either corrected or explicitly retained with rationale. This distinction is essential as every nonconformance or discovered weakness must lead to an engineering decision, not merely a comment in a test report.

One work product that is particularly important at release is the cybersecurity manual and its supporting integration documentation. Its purpose is to communicate the assumptions, constraints, configuration requirements, and operational steps that must be followed for the product to remain secure when integrated by the customer or upstream supplier. A product may be secure as developed and insecure as integrated if these assumptions are not communicated and preserved.

This is not an academic concern but rather one that has been demonstrated to cause real-world security failures. Indeed, improper provisioning, incorrect lifecycle state transitions, or premature locking of hardware settings has left systems in insecure or unrecoverable states. A common example is incorrect fuse programming before the system has been fully provisioned with required secrets and configuration data. Once the wrong lifecycle transition occurs, the product may no longer be recoverable without expensive rework or replacement. Whether this information is packaged in a formal cybersecurity manual or in equivalent controlled documentation, the essential requirement is the same: the secure setup and secure integration assumptions and requirements must be explicitly documented.

Production planning

Manufacturing, deployment, and release processes must be considered to ensure that the system is produced and delivered securely. While not every product requires the same degree of production planning, any product that undergoes a transition between development and field release must account for the security implications introduced during that transition.

The fundamental issue is that security does not stop at design completion. A system may be engineered with strong security properties, but if the processes that move the product from the development environment into the field are poorly controlled, those security properties can be undermined before the system is ever used by a customer.

For some products, the production planning challenge is relatively limited. Consider a software library that is integrated by downstream developers. Such a product may not involve physical manufacturing, but it still passes through a transformation between development artifacts and

distributed software components. Even in the absence of a factory floor, the organization must address questions such as how the software is packaged, how releases are approved, how version integrity is maintained, and how distribution is controlled. If the software artifact can be modified between development and consumption, then the absence of physical manufacturing does not eliminate the production security problem. It simply changes its form. Secure packaging, release integrity validation, and authenticated distribution channels become the mechanisms that preserve trust in the delivered component.

For hardware products, the production planning problem becomes broader and often significantly more sensitive. Organizations producing chips, secure elements, or even complete ECUs must consider how security properties are preserved throughout the manufacturing process itself.

Several issues such as the following are critical in this environment:

- How manufacturing test modes are controlled
- How secrets are provisioned into devices
- How lifecycle states are transitioned during production
- How security-relevant configuration settings are locked prior to shipment

Manufacturing environments frequently require access mechanisms that are not acceptable in deployed systems. For example, debug interfaces, test access ports, and engineering backdoors may be necessary to validate the hardware during production. However, once the product leaves the factory, those same interfaces may become high-risk attack vectors if they remain enabled or are improperly restricted.

Similarly, the handling of cryptographic assets during manufacturing must be carefully controlled. Device identity keys, root certificates, and provisioning material must be injected into the device in a manner that prevents unauthorized duplication or extraction. If these assets are mishandled during production, the resulting devices may appear authentic while silently violating the trust assumptions embedded in the system architecture. A typical example is the handling of fuse secrets in plaintext form while in the factory environment. Too much trust in the factory environment security is a perennial problem for the automotive industry. The fact of the matter is that automotive manufacturing facilities have been repeatedly breached in recent years mainly through ransomware attacks which should force us to re-evaluate how much trust can truly be delegated to that "secure" factory environment.

Even products that are purely software-based may require production-specific security controls. Examples include the following:

- Code signing infrastructure used to authenticate released binaries
- Release approval workflows that ensure only authorized artifacts are published

- Artifact integrity controls that detect tampering in build pipelines
- Secure distribution mechanisms that prevent unauthorized modification during delivery

A managed security approach therefore does not treat development security and production security as separate concerns. Instead, it views them as connected phases within a single chain of trust. The controls applied during development must continue through manufacturing, packaging, release, and distribution so that the security posture established during engineering is preserved all the way to the deployed product.

The central question for any organization is straightforward: *what procedures are required to move from engineered design to fielded product without introducing new attack surfaces?*

The specific answer will vary depending on the nature of the product and the production environment. Nevertheless, the underlying principle remains the same. If production introduces unmanaged opportunities for tampering, unauthorized configuration changes, key exposure, or residual debug access, then the security assurances established during development can be lost before the end-user ever interacts with the system.

Operations and maintenance

Is our job done yet? If you guessed no, then you are indeed correct.

A managed security approach does not end at release. Connected vehicles, software-defined architectures, and subscription-based feature delivery have expanded the operational attack surface well beyond the point of sale. Once vehicles receive updates, unlock features after purchase, and maintain long-term relationships with backend services, cybersecurity becomes a continuing operational discipline rather than a one-time engineering milestone.

Software update management is therefore a critical part of the full cybersecurity story. The organization must be able to develop, validate, approve, distribute, and monitor updates in a way that preserves both security and system integrity. This includes not only patching vulnerabilities, but also managing feature activation, software authenticity, update rollback handling, and the interaction between cybersecurity and functional safety.

This challenge becomes more visible as OEMs monetize post-sale software features. In such environments, software piracy, and unauthorized feature enablement become practical threats. The attack stops being limited to stealing data or crashing services but extends to bypassing subscription checks, forging update packages, replaying backend responses, or tampering with local policy enforcement to unlock premium functionality. As seen in other digital industries, once a method for bypassing feature controls is discovered and leaked, it may spread rapidly beyond sophisticated attackers.

That is why update security cannot be treated merely as a convenience feature. It is part of the trust model of the vehicle. The organization must understand what changed, why it changed, what dependencies were affected, what validation evidence exists, and whether the update can be deployed safely to the intended fleet population. Secure OTA therefore depends on authenticated sources, integrity verification, version control, anti-rollback considerations where applicable, deployment governance, and post-deployment monitoring.

Operations also include **cybersecurity monitoring** and **vulnerability management**. Once products are in the field, new vulnerabilities will continue to emerge through internal testing, external research, supplier disclosures, incident response, and the public security ecosystem. A managed program therefore needs a process for ingesting vulnerability information, assessing applicability, prioritizing remediation, documenting impact, and deciding whether patches, mitigations, or retained-risk positions are appropriate. The explosion of open-source software within the automotive ecosystem particularly makes this a challenging task. Being able to quickly react to emerging vulnerabilities in open-source software will be critical for the automotive industry to withstand the speed at which offensive AI-driven security is able to uncover exploitable weaknesses.

This is another area where the managed approach clearly differs from the ad-hoc approach. In ad-hoc security, vulnerabilities often trigger reactive patching without a durable analysis trail. In a managed approach, each discovered vulnerability becomes an input into a governed process that can produce technical action, evidence, and where justified, a defensible risk decision.

End of life and decommissioning

The final chain in our managed security approach is the handling of the product's end of life where the product or some of its components are no longer supported. In fact, end of life is not the absence of a security problem but rather a security phase with its own set of risks.

When products, ECUs, backend services, or supporting tools are decommissioned, the organization must account for what happens to secrets, credentials, signing authority, diagnostic capability, maintenance access, stored personal data, and unsupported software components. A product that is no longer maintained but remains operational in the field may still expose customers and the brand to risk if its support boundaries are not clearly defined and securely enforced.

Decommissioning therefore requires a deliberate strategy. This may include revoking credentials, disabling backend dependencies, ending update trust relationships, removing unused accounts, sanitizing sensitive data, and documenting the security assumptions that no longer hold once support ends. For hardware, the organization may also need to consider

secure disposal, secure recycling, or retirement of components that contain keys, certificates, or proprietary algorithms.

The larger lesson is that a managed approach treats cybersecurity as a lifecycle discipline from concept to retirement. The CSMS is valuable not because it creates paperwork, but because it creates control over how risk is introduced, analyzed, reduced, verified, validated, monitored, and ultimately retired. That is the real difference between hoping a product is secure and being able to demonstrate, with evidence, why it is secure enough to deploy.

Now that we have demonstrated the need for the managed security approach, we will discuss the challenges that stand in the way of systematic risk reduction and raising the security bar.

Raising the security bar in automotive systems

In many cases, meeting the minimum-security bar may seem daunting by itself. However, as we have seen throughout this book, attackers are constantly improving their discovery and exploitation techniques. Therefore, if defensive security engineers fail to keep up with advances in the attack space, then losing the battle is imminent. But before we even consider how to raise the security bar, it helps to understand some of the limiting factors against doing so.

Process pressures in rapid development cycles

We discussed a lot throughout this chapter about the need for a cybersecurity management system that can enable the reduction of cybersecurity risks in a systematic fashion. This is easier said than done, especially in the world of SDVs. Modern SDVs thrive on innovation, adopting a "move fast, fail fast" culture that accelerates development. However, this approach often conflicts with the stringent security and safety protocols required by standards such as ISO/SAE 21434.

When compliance is perceived as overly cumbersome, teams may feel compelled to bypass process, undermining effective risk management. In a worst-case scenario, what starts as an honest attempt at process-driven risk reduction can be overcome by the tendency to reduce process to a "checkbox" exercise. The objective turns from truly addressing risk to producing heavy paperwork merely to satisfy an external assessor. Moreover, when the immediate benefits of thorough risk management are unclear, teams might prioritize meeting deadlines over addressing deeper, latent threats.

This superficial approach can mask vulnerabilities that become prohibitively expensive to address later in the engineering lifecycle. Compounding this issue is the reliance on open-source software and complex supply chains, which create additional challenges in maintaining a comprehensive **Software Bill of Materials** (**SBOM**). Ensuring visibility across such

ecosystems and addressing vulnerabilities in off-the-shelf modules not designed for automotive security further strains resources.

Human resource gap

The rapid evolution of SDVs demands expertise in areas such as advanced networking, virtualization, and cryptographic controls, domains traditionally associated with enterprise IT. These technologies require the automotive engineering taskforce to be retrained from the classic ECU development model of RTOS-based execution to service-oriented architectures. Consequently, transitioning to SDV architectures requires extensive workforce training, a daunting task under the pressure of relentless project deadlines.

One common challenge when discussing security risks with engineering teams is the disconnect between what is perceived as feasible versus what is truly feasible. The desire to ship the product on time biases engineering teams to underestimate risk. Naturally, delaying a project due to hypothetical threats can seem unwarranted when compared to the tangible repercussions of missing deadlines. This mindset, coupled with the inability to anticipate complex attack scenarios, leaves critical vulnerabilities unaddressed. Furthermore, the fast-paced nature of SDV development fosters burnout among developers, who may view security tasks as extraneous friction, leading to mental fatigue and compromised implementation.

Technical challenges of centralized architectures

The shift from domain-driven to centralized architectures collapses the physical separation that previously segmented automotive systems. Where attackers once faced multiple hardware barriers, they can now potentially exploit a single misconfiguration to access critical functions. Virtualization aims to replicate physical isolation, but its effectiveness is contingent on perfect configuration and flawless hardware implementations which have repeatedly been shown to fail.

Another unique technical challenge is the need for generating extensive logs and debug data, which can provide adversaries with valuable insights if left unsecured. At the same time, OEMs face logistical hurdles in distributing secure patches to millions of vehicles. SDVs also rely heavily on cloud services for AI augmentation and data processing, creating additional attack vectors. A breach in cloud infrastructure can cascade into vehicle-level compromises, emphasizing the interconnectedness of vehicle and enterprise security.

Attacker's landscape

SDVs' IT-like architectures lower the barrier of entry for attackers. Exploit toolkits developed for traditional IT environments can often be adapted to SDVs with minimal effort. Additionally, the increasing standardization across OEMs amplifies the scalability of successful attacks. A lack of high-profile exploits may create a false sense of security, delaying critical investments in defensive measures.

Elevating automotive security

Given the above-mentioned challenges, a holistic approach is needed to elevate the security bar. One that unites process improvement, workforce development, and technological innovation. First, we need to address the streamlining of secure engineering lifecycles. Security tasks must be seamlessly integrated into fast-paced development cycles. This involves creating adaptable frameworks that detect vulnerabilities early and manage residual risks systematically. Automation can assist in maintaining development velocity while ensuring compliance with rigorous security standards.

Second, there is a need to empower human talent. Upskilling engineers through continuous cybersecurity training is critical. A culture that values comprehensive risk assessment and scenario-based threat modeling can help overcome the tendency toward superficial "paper security." Third, we must transition to Zero Trust architectures. An example of that is hardening HPC platforms with robust virtualization, cryptographic safeguards, and secure boot mechanisms. Another example is enforcing strict resource isolation to ensure that vulnerabilities in one domain do not cascade into others.

Finally, we must embrace agentic AI workflows throughout our secure engineering lifecycle which is the focus of our next section.

Winning the race

When we began this discussion, we warned about the dangers of adopting the same security posture that has become common in adjacent industries, where the operational model has devolved into what is often described as "patch and pray." In those environments, software is released with the expectation that vulnerabilities will inevitably be discovered and corrected later through patches. While this model may be tolerated in consumer IT systems, it is fundamentally incompatible with safety-relevant systems such as those found in modern vehicles. If you have doubts about the last statement, then ask yourself: how quickly can an enterprise vendor issue a software patch as opposed to a vehicle OEM? Even if we solve the patch speed problem, the mere fact that vulnerabilities in automotive systems have an impact on human life and safety severely reduces our appetite for security failures.

To avoid drifting into that model, we advocated earlier in this book for the managed approach to cybersecurity, where security is treated as a systematic engineering discipline integrated throughout the product lifecycle. However, defensive security practices alone are not sufficient. Winning the cat-and-mouse race with attackers requires more than building defenses. It requires anticipating how those defenses might be bypassed.

In practical terms, this means investing not only in defensive controls but also in offensive security methods and tooling. The goal is not to emulate attackers for curiosity's sake, but to

understand their capabilities before those capabilities are used against the system in the real world. This calls for synergy between defensive and offensive security domains. Defensive security focuses on architecture, isolation, and risk reduction. Offensive security focuses on discovering weaknesses and constructing attack chains. When these two disciplines operate independently, each develops blind spots. When they operate together, each strengthens the other.

Moreover, the emergence of AI-driven offensive security dramatically increases the urgency of this collaboration. As we will explore in this section, AI is significantly reducing the cost of conducting offensive security campaigns. Tasks that previously required weeks of expert effort can now be attempted in hours. The barrier to entry for exploit development is dropping, and the possibility of large-scale attacks against automotive systems is becoming far more plausible. Paradoxically, this same technological shift creates an opportunity for defenders. If AI can be used to accelerate offensive discovery, it can also be used to systematically reduce the number of exploitable weaknesses before systems reach production. For the first time in history, the idea of approaching near-zero exploitable weaknesses has become a realistic engineering objective.

Achieving this outcome requires integrating AI assistance throughout the secure engineering lifecycle. However, AI-assisted workflows are not a substitute for sound security architecture. Architectures that prioritize isolation, containment, and well-defined trust boundaries remain the foundation of system security. Indeed, applying agentic workflows to an inherently insecure architecture is nothing more than an expensive way to produce AI slop as we fail to compensate for poor design decisions. In such cases, organizations end up spending compute resources and model tokens attempting to close vulnerabilities in systems that should never have been deployed in their current form.

For this reason, AI-driven workflows must be applied systematically and early, beginning with feature analysis and continuing through architecture review, implementation analysis, and penetration testing. Conceptually, the relationship between defensive and offensive domains can be visualized as a continuous feedback loop:

```
System Definition → TARA → Security Architecture → Implementation
        ↓                                    ↑
  Offensive Testing (Pentest) ← Vulnerability Discovery
```

In this model, TARA outputs identify high-value assets and security goals, penetration testing focuses on features and components associated with those critical assets, technical concept validation identifies architectural weaknesses and exploitable vulnerabilities, and findings feed back into design improvements and mitigation strategies. The result is a closed-loop

security engineering process where offensive insights continuously strengthen defensive architectures.

To understand why this integrated model is necessary, we must first examine how the economics of exploit development are changing.

From human-limited to autonomous attack scaling

For many years, exploit development in automotive systems was heavily constrained by human bandwidth. Offensive security required patience, intuition, and deep technical specialization. As we saw in previous chapters, a typical campaign often involved the following:

- Reverse engineering firmware images that were stripped of symbols and documentation
- Manually analyzing CAN, LIN, Ethernet, or proprietary transport protocols
- Building handcrafted fuzzing harnesses based on partial documentation and inference
- Constructing multi-stage exploit chains across ECUs, gateways, and backend services
- Applying domain knowledge spanning embedded constraints, automotive E/E architectures, and hardware security boundaries

Each of these activities imposed a nonlinear cognitive burden. Progressing from vulnerability discovery to a weaponized exploit required significant time and expertise. The primary bottleneck was the discovery phase, where researchers had to expose exploitable weaknesses before any attack chain could be constructed.

AI offensive agents fundamentally alter this equation, as shown in *Table 10.1*:

Activity	Human effort model	AI-augmented model
Binary analysis	Weeks	Hours
Protocol inference	Manual reverse engineering	Automated clustering and symbolic reasoning
Fuzz test generation	Heuristic-driven	Coverage-guided generation via LLM planning
Exploit synthesis	Manual payload crafting	Assisted payload construction

Activity	Human effort model	AI-augmented model
Attack chain assembly	Human cognitive modeling	Multi-agent planning

Table 10.1: Offensive security effort differences between the human and AI-augmented models

The shift from human to AI-augmented model is subtle but profound. The limiting factor is no longer purely expert labor; instead, it becomes computational scale and model capability. Historically, attacker scaling was linear with skill, where a small team of experts could only investigate a limited number of targets at a time. With AI-augmentation, attacker scaling becomes computational. As GPU cycles increase, model capability improves, and orchestration pipelines mature, exploit throughput scales with automation capacity rather than with the size of the offensive team. Rather than being an incremental improvement, this represents a structural acceleration in offensive capability.

To understand how this transformation unfolds in practice, we must classify the types of AI systems driving this shift.

Taxonomy of AI offensive capabilities

To reason rigorously about AI-enabled threats, we classify offensive systems into three categories. The distinction is operationally important because each category introduces a different type of risk to automotive security programs.

LLM-assisted human attackers

The earliest use of **large language models** (**LLMs**) in security research falls into this category. In this model, a human operator remains in control of the penetration testing campaign but uses LLMs as force multipliers. Typical tasks include the following:

- Explaining complex code and identifying potential vulnerabilities within large codebases
- Assisting with CVE-to-exploit transformation
- Interpreting decompiled binaries to accelerate reverse engineering efforts
- Generating fuzzing harnesses from partial interface descriptions

In this model, humans remain in the loop. The objective is to increase throughput and reduce the expertise barrier. For example, instead of requiring a penetration tester to manually interpret assembly language, the tester can ask an LLM to explain the logic of a disassembled function. Similarly, instead of assigning a team to manually review a large codebase such as an entire boot chain, the tester can prompt the model to search for patterns that correspond to specific vulnerability classes.

The primary impact is the reduction of the barrier to entry. Tasks that previously required senior engineers can now be attempted by mid-level practitioners augmented by AI. The skill ceiling remains important, but the skill floor collapses.

Autonomous vulnerability discovery pipelines

As experimentation with AI-assisted collaboration matured, researchers quickly realized that the human in the loop often remained the primary bottleneck. Early systems generated large volumes of findings, many of which were false positives that required manual triage. Reducing this bottleneck required moving from assistance toward automation.

Autonomous vulnerability discovery pipelines operate continuously and are typically triggered by events such as new code commits or nightly testing cycles. Typical components include the following:

- Static analysis engines that reason about secure coding violations and propose patches
- Automated harness generation for fuzz testing
- Coverage-guided fuzzing systems that explore code paths automatically
- Crash triage automation that clusters and prioritizes faults
- Root cause classification using CWE taxonomy mapping

When used offensively, these pipelines transform vulnerability discovery into a computational problem. Instead of deciding which weaknesses to investigate, attackers can investigate all of them provided sufficient compute resources are available.

The practical implication is that vulnerability discovery becomes ubiquitous rather than selective. Systems are scanned continuously rather than during occasional penetration testing engagements. This fundamentally alters the defender's timing assumptions.

Multi-agent offensive systems

The third category represents the most consequential shift and is considered the state of the art as of the writing of this book. These systems orchestrate multiple specialized agents that mirror what a full penetration testing team does. Combined with tool calling and skills, the multi-agent approach can automate all aspects from discovery, to attack chain construction all the way to testing the exploit to produce a fully operational exploit.

Figure 10.3 shows an example agentic workflow that is based on the discovered AI espionage campaign by Anthropic: `https://www.anthropic.com/news/disrupting-AI-espionage`

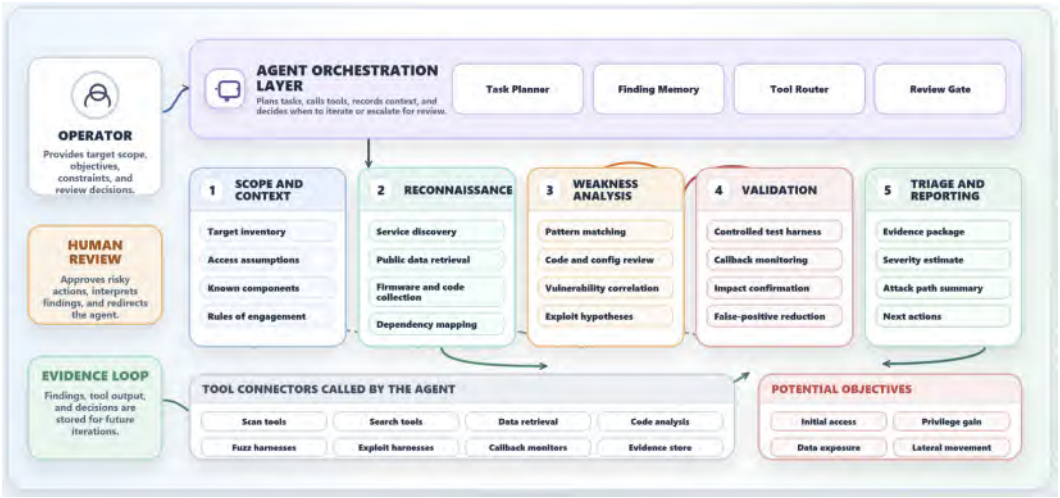


Figure 10.3: AI-orchestrated offensive security pipeline

This multi-agent offensive system operates as an orchestrated campaign rather than a single exploit attempt. The process begins with a human operator defining the target and objectives. From there, an AI coordinator decomposes the mission into iterative phases. In the reconnaissance phase, specialized agents invoke scanning, search, data retrieval, and code analysis tools to collect structured intelligence. Findings are aggregated, summarized, and fed back into the planning loop. In subsequent phases, exploitation agents attempt controlled exploit execution and validate callbacks against target services, while additional agents refine attack paths based on observed system behavior.

As access is gained, internal reconnaissance agents pivot laterally, harvest credentials, and enumerate new trust boundaries. The system continuously re-evaluates attack graphs, directs further tool invocation, and escalates privilege where feasible. Ultimately, the workflow culminates in objective-driven actions such as controlled data exfiltration. The key architectural insight is that reasoning, tool invocation, exploitation, and feedback are distributed across cooperating agents, enabling end-to-end attack chain construction that adapts dynamically to system responses.

This is the inflection point. AI systems are no longer limited to identifying coding vulnerabilities like memory corruption defects. They are reasoning about trust boundaries, update pipelines, shared services, privilege hierarchies, and cross-domain communication paths. This transforms the problem of required time and human resources to required computational resources and token generation costs. But unlike humans, scaling AI workflows can happen efficiently and on demand.

Why vulnerability management fails to scale

The traditional defensive model follows a reactive sequence:

1. Scan
2. Triage
3. Patch
4. Monitor CVEs

In legacy distributed ECU architectures with limited software complexity, this model was marginally sustainable. In centralized HPC platforms, it begins to fail. Several factors including the following contribute to this breakdown:

- Massive service counts within centralized compute nodes
- Large numbers of third-party libraries and opaque dependencies
- Supplier-delivered binaries that cannot be easily modified
- Mixed safety and security domains sharing common infrastructure
- OTA deployment pipelines that complicate coordinated updates

As AI reduces the cost of offensive discovery, the defender's manual triage and patching workflows cannot converge fast enough. The result is not merely a backlog of vulnerabilities. It is a systemic velocity mismatch between attacker automation and defender response. This leads to the next question: "*If vulnerability management does not scale, what does?*"

Vulnerability elimination: A reduction-oriented definition

Vulnerability elimination does not imply that systems will contain zero defects. Instead, we define elimination as a systematic reduction in exploitable weakness density through lifecycle-integrated AI agents that prevent classes of vulnerabilities before deployment.

The objective is not perfection, but rather a measurable reduction of entire vulnerability classes. Examples include eliminating memory corruption classes through safer language subsets or memory-safe runtimes, eliminating deserialization flaws through strict schema validation patterns, and eliminating update-chain tampering through provable integrity guarantees. The strategic shift is from reactive patching to structural prevention as it scales differently than remediation.

AI-augmented V-model

Now that we understand the implications of AI-driven offensive security, we must examine the defensive counterpart. Just as AI agents automate offensive discovery, they can also augment defensive engineering. The objective is straightforward: reduce the number of exploitable

weaknesses as close as possible to zero, thereby eliminating the advantage gained by automated offensive systems.

To achieve this objective, AI agents must be integrated across the automotive V-model, rather than being applied only at the end of development. Following are five use cases that illustrate how AI agents enhance our defensive security capabilities.

Feature-level threat modeling

At the feature level, AI agents can assist security engineers by first ingesting the feature specification and building context from existing requirements, architecture diagrams, and product documentation, as shown in *Figure 10.4*:



Figure 10.4: Feature-level threat generation agent flow

Using this context, the agent identifies assumed trust boundaries and relevant assets, then generates candidate threats such as unauthorized access, spoofing, denial of service, or data exposure. It can then derive mitigations from approved security catalogs, such as authentication, authorization, secure communication, input validation, and access control, and map those mitigations to existing security requirements and controls. Where gaps are found, the agent proposes updates to the requirements or architecture so that the feature enters downstream engineering with a clear set of security requirements and controls already attached.

Architecture-level security review

At the architecture level, AI agents can help security engineers evaluate whether a proposed design preserves the intended trust boundaries and security properties before implementation begins. As shown in *Figure 10.5*, the agent first builds context from existing design documents, security requirements, interfaces, and data flows, then analyzes the proposed architecture against known security principles. This analysis can identify weaknesses such as excessive privilege, weak isolation, untrusted interfaces, shared memory misuse, privilege-boundary violations, and gaps in secure boot or update chains:

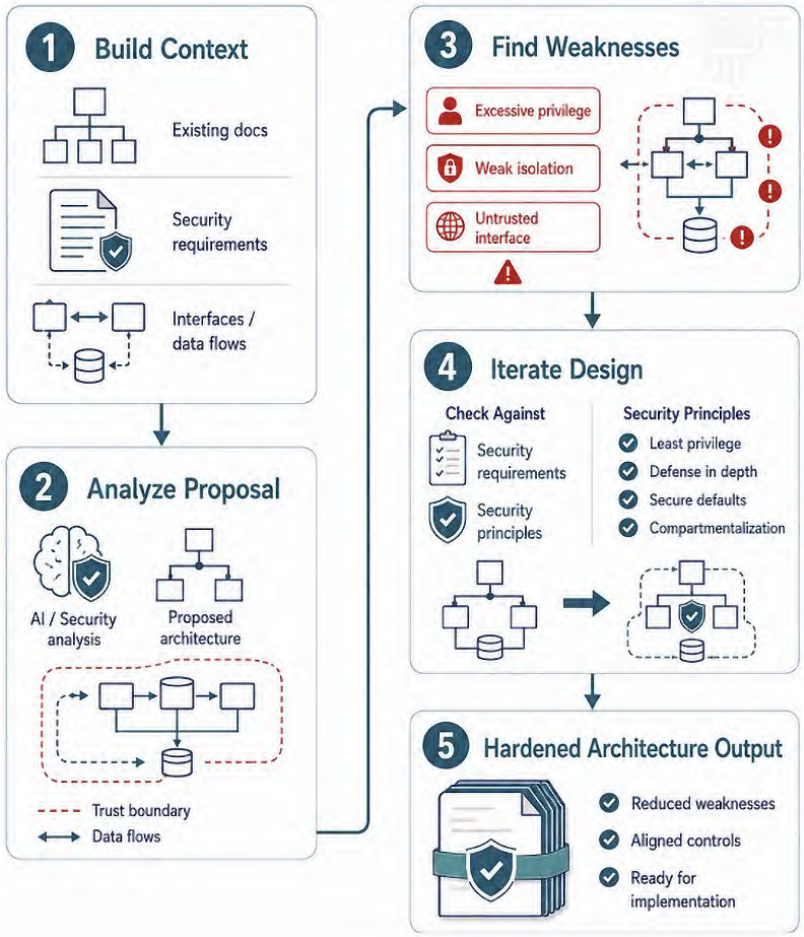


Figure 10.5: Architecture weakness finding agent

The agent then checks the design against requirements and principles such as least privilege, defense in depth, secure defaults, and compartmentalization, proposing architectural changes wherever needed. The result is a hardened architecture with reduced weaknesses, better-aligned controls, and a stronger basis for downstream implementation.

Implementation-level analysis

At the implementation level, AI agents can operate continuously inside the CI/CD pipeline so that new code changes are analyzed as soon as they enter the build flow. As shown in *Figure 10.6*, the agent checks each update against secure coding rules, such as MISRA and CERT C/C++, as well as CWE patterns, prior bug patterns, and organization-specific findings from earlier reviews. This allows the agent to detect weaknesses such as memory misuse, race conditions,

unsafe deserialization, recurring vulnerable patterns, and other known exploit primitives before they reach production builds:

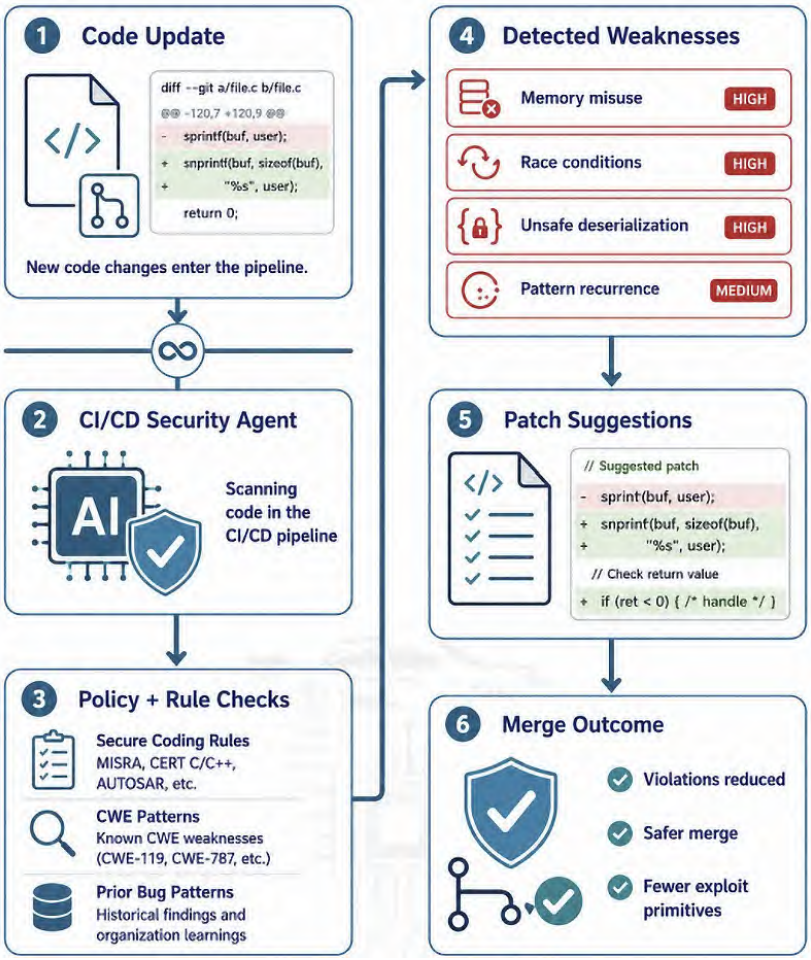


Figure 10.6: Code implementation weakness finder flow

Instead of only reporting violations, the agent can also provide immediate patch suggestions, helping developers correct insecure code while the context is still fresh. The outcome is lower detection latency, fewer repeated vulnerability classes, safer merge decisions, and a reduced chance that familiar exploit primitives propagate into released ECU software.

Automated fuzzing and white-box testing

AI agents can reduce one of the biggest barriers to effective fuzzing: the manual effort required to build harnesses, grammar, and useful test cases. As shown in *Figure 10.7*, agents can ingest

protocol specifications, and seed inputs then generate fuzzing assets including harnesses, protocol grammar, and targeted test cases. They can also infer protocol state machines, explore state space, prioritize coverage gaps, and use symbolic or boundary exploration to drive execution toward security-relevant paths:

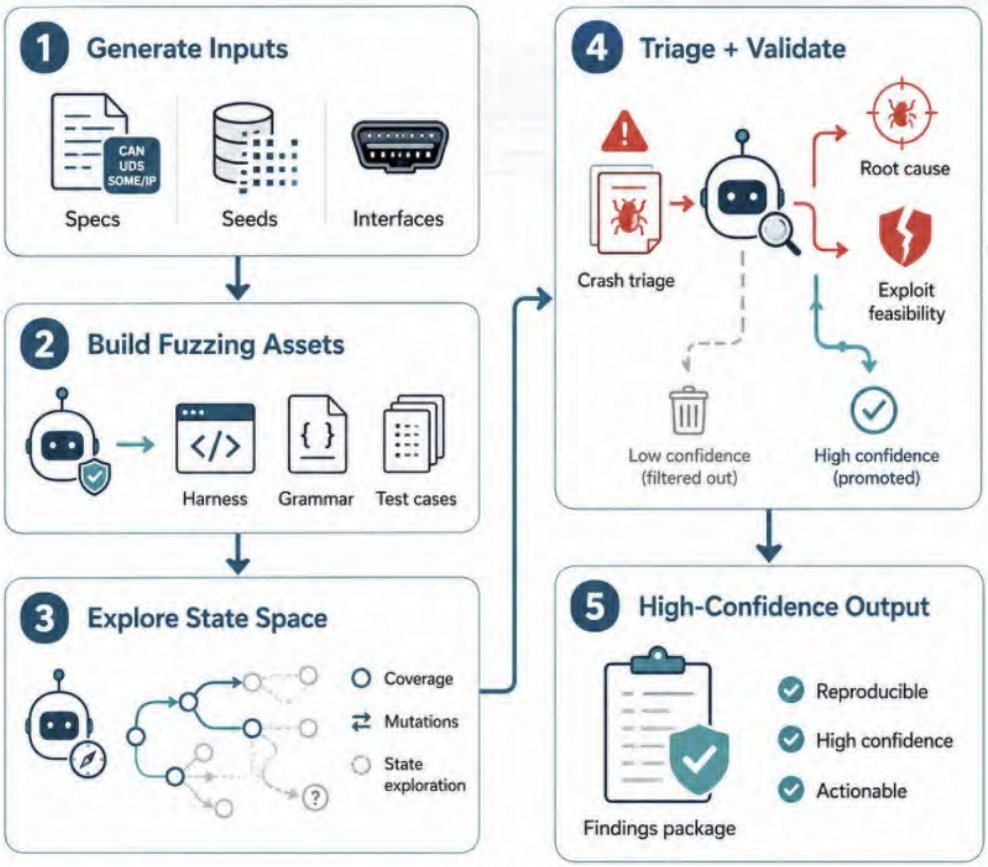


Figure 10.7: Automated fuzzing agent flow

After crashes or abnormal behavior are observed, the agent can assist with crash triage, root cause analysis, exploit feasibility assessment, and reproducibility checks. Low-confidence findings can be filtered out, while high-confidence issues are promoted with evidence, reducing noise and helping engineering teams focus on actionable vulnerabilities. In this model, coverage becomes a continuously optimized security metric rather than a byproduct of handcrafted tests.

Offensive simulation agents

Internal red-team AI agents can simulate adversarial behavior by constructing attack graphs, modeling privilege-escalation paths, evaluating attack chains, and testing target components with adversarial inputs. As shown in *Figure 10.8*, not every aspect of penetration testing can be fully automated, especially activities that require physical setup, ECU wiring, lab instrumentation, or human safety judgment:

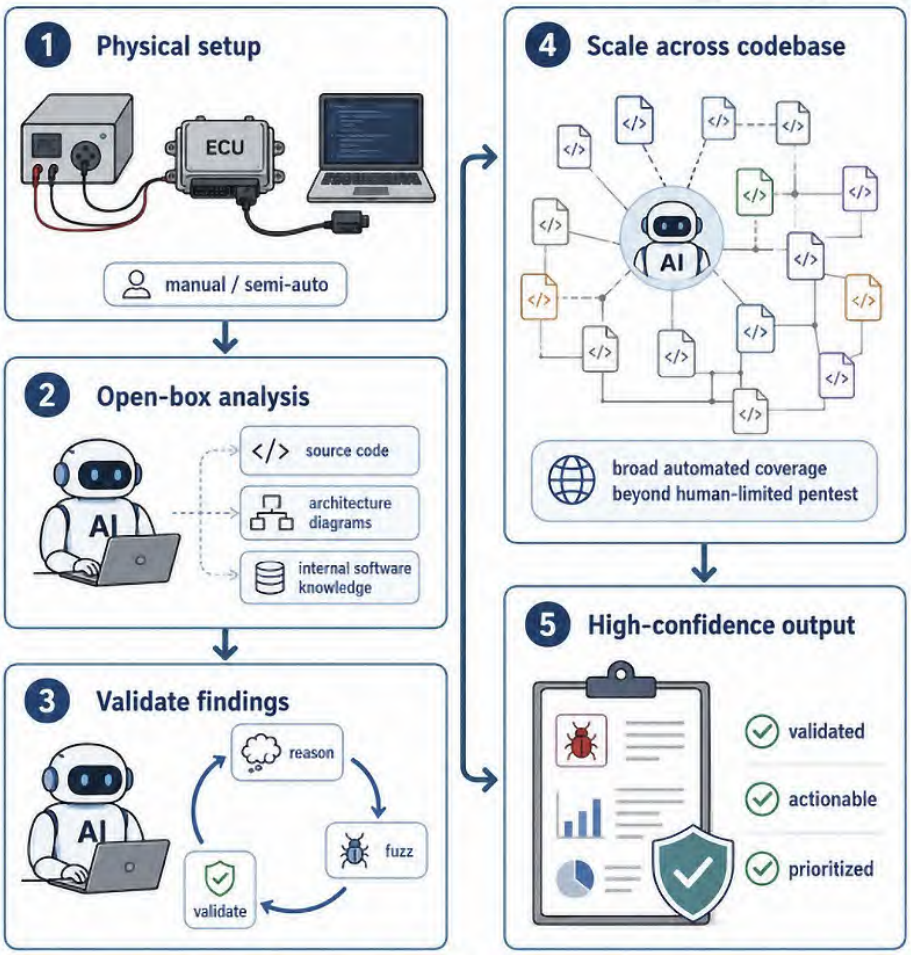


Figure 10.8: Pentesting agent automated flow

However, open-box penetration testing is well suited for automation because agents can analyze source code, architecture diagrams, and internal software knowledge at scale. They can reason through discovered weaknesses, generate fuzz tests or validation checks, and promote only findings that are reproducible, actionable, and prioritized. Similarly, pentesting

agents can emulate network attacks when given access to networking tools. This expands security testing beyond the portions of the codebase that a human pentest team can manually cover.

Even so, AI-augmented pentesting should be treated as an assurance layer rather than the primary defensive strategy. Agents will not find every vulnerability and relying too heavily on late-stage pentesting can create disruptive patch cycles. The objective is to apply strong security practices upstream, then use offensive simulation as an insurance policy to close the automation gap before attackers exploit it.

Summary

In this final chapter, we charted a path forward that addresses the new realities of automotive cybersecurity where we can no longer rely on reactive defenses. The historical model of discovering vulnerabilities late and correcting them through patches was always fragile, but in the context of SDVs and centralized HPC architectures, it becomes fundamentally unsustainable. Vehicles now resemble distributed computing platforms connected to cloud ecosystems, executing millions of lines of code across highly interconnected components. In such an environment, the assumption that security failures can simply be corrected after deployment is incompatible with both safety expectations and operational realities.

We discussed that the ad-hoc approach fails not because organizations lack effort, but because it lacks structure. Without a systematic method to identify risks, allocate controls, and verify their effectiveness, security decisions become fragmented and reactive. The examples discussed throughout this chapter demonstrated how such fragmentation leads to implicit trust assumptions, incomplete threat analysis, unmanaged supply chain exposure, and delayed vulnerability discovery. When these weaknesses accumulate, the result is a system that appears secure in documentation yet remains fragile in practice.

The insight into managed security approach provided you with the necessary corrective structure. By integrating cybersecurity into the full engineering lifecycle through a CSMS aligned with ISO/SAE 21434, organizations move from reactive incident handling to controlled risk management. TARA, cybersecurity concepts, secure design principles, disciplined implementation practices, and structured verification and validation create a chain of evidence that connects risk identification to concrete engineering decisions. This lifecycle perspective extends beyond development into production, deployment, operation, and eventual decommissioning, ensuring that security assumptions remain valid throughout the product's operational life.

The chapter also focused on the recognition that cybersecurity is not a single discipline but an ecosystem of cooperating activities. Defensive architecture, secure development practices, vulnerability monitoring, penetration testing, and operational monitoring must reinforce each

other rather than operate in isolation. Offensive security methods, when applied constructively within a managed framework, serve as a powerful validation mechanism for the assumptions made during design. By exposing weaknesses early, they allow organizations to strengthen their systems before adversaries can exploit them.

The path forward not only addresses the security improvements required to raise the security bar in modern vehicle architectures, but also confronts the emerging risks associated with AI-driven offensive capabilities. Paradoxically, we learned that the same technological shift provides defenders with powerful new capabilities. Generative AI can support secure engineering by scaling security processes, assisting engineers in reasoning about threats, and helping organizations maintain consistency across increasingly complex systems. AI agents can assist with threat modeling, architecture analysis, vulnerability detection, and secure implementation review, reducing the likelihood that exploitable weaknesses propagate into production systems. While AI cannot replace human ingenuity in anticipating novel attacks, it significantly enhances efficiency and enables a more integrated and proactive approach to cybersecurity.

However, automation alone cannot solve fundamental engineering weaknesses. If the automotive sector intends to build resilient SDVs, it must pursue structural improvements in how systems are engineered. We also discussed several technical directions that offer promising paths forward.

One of the most impactful changes is the adoption of memory-safe programming languages such as Rust or Ada/SPARK for security-critical components. Eliminating entire classes of memory corruption vulnerabilities reduces the attack surface before exploitation becomes possible. The chapter also explored how AI-assisted workflows can be integrated throughout the development lifecycle to reduce the density of exploitable weaknesses. When applied to requirements analysis, architecture review, code generation, testing, and vulnerability discovery, AI systems can help engineers identify weaknesses much earlier in the engineering process. The result is a shift from reactive vulnerability management to proactive vulnerability elimination.

The path forward is therefore clear. Secure mobility will not be achieved by reacting faster to vulnerabilities. It will be achieved by systematically reducing the number of exploitable weaknesses that ever reach the road. Security must be treated as a lifecycle discipline that begins with architecture, continues through implementation and validation, and persists throughout the operational life of the vehicle. By combining rigorous engineering processes, advanced security technologies, generative AI assistance, and a well-trained workforce, the automotive industry can build systems that are resilient not only against today's threats but also against the evolving adversaries of tomorrow.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

11

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:







Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/en-us/unlock?step=1`.

> **Note**
>
> **Note**: If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 11.1* on desktop), search for this book by name and select the correct edition.

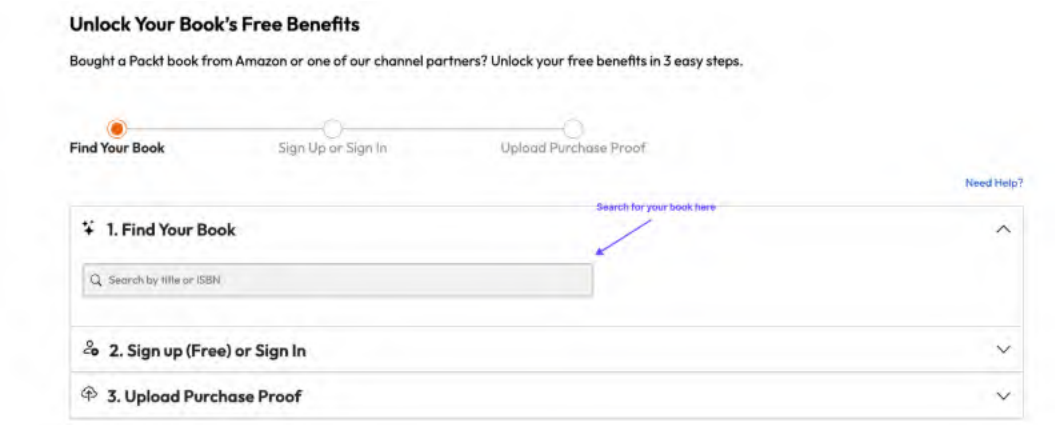


Figure 11.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.

Note

Note: If you are still facing issues, reach out to `customercare@packt.com`.



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Automotive Cybersecurity Engineering Handbook

Dr. Ahmad MK Nasser

ISBN: 978-1-80107-653-1

- Understand automotive cybersecurity standards like ISO 21434 and UNECE REG 155-156.
- Apply threat modeling techniques to identify vulnerabilities in vehicle systems.
- Integrate cybersecurity practices into existing automotive development processes.
- Design secure firmware and software architectures for automotive ECUs.
- Perform risk analysis and prioritize cybersecurity controls for vehicle systems
- Implement cybersecurity measures at various vehicle architecture layers.



Agentic AI for Offensive Cybersecurity

Orhan Yildirim

ISBN: 978-1-80611-447-4

- Understand agentic AI principles for offensive security
- Evaluate and select AI frameworks for security automation
- Build your first AI-driven recon workflow using n8n
- Automate attack surface discovery and port scanning with AI
- Design PCI segmentation testing workflows with n8n
- Build web app testing tools using MCP
- Automate exploitation, payload generation, and reporting
- Create threat intelligence agents that monitor CVE feeds

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Offensive Automotive Cybersecurity*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



https://packt.link/r/1836648634

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

8-bit integer quantization 439

802.1Q trunking 238

A

AFL 208

AFL++ 114, 196

AI and ML systems, vehicle sensors 409, 416
- ADAS perception and planning 416
- in-vehicle AI assistants 416

AI offensive agents 489

AI offensive capabilities 490
- autonomous vulnerability discovery pipelines 491
- LLM-assisted human attackers 490
- multi-agent offensive systems 491, 492

AI-augmented V-model 493
- architecture-level security review 495, 496
- automated fuzzing and white-box testing 497, 498
- feature-level threat modeling 494, 495
- implementation-level analysis 496, 497
- offensive simulation agents 499

AI-enhanced fuzzing 68

APKTool 147

ARP spoofing 244

AT set mode command 160

AUTOSAR SecOC 451

AUTOSAR XML (ARXML) 245, 248

AUTomotive Open System ARchitecture (AUTOSAR) 230

AWS AppSync 138

AWS IoT Device Defender 93

AWS Lambda function 138

Access Point Names (APNs) 144

Adaptive Cruise Control (ACC) mode 438

Address Resolution Protocol (ARP) 228

Address Space Layout Randomization (ASLR) 209, 374, 448

Advanced Audio Distribution Profile (A2DP) 181

Advanced Driver Assistance Systems (ADAS) 14, 104, 135, 407

Advanced Message Queuing Protocol (AMQP) 139

Adversarial Robustness Toolbox (ART) 119, 432

Amass 88, 146

Amazon Web Services (AWS) 134

Android Debug Bridge (ADB) 190, 374

Anti-lock Brake System (ABS) 267

Application Programming Interfaces (APIs) 416

Audio Video Bridging (AVB) 224

Automatic Parking Control (APC) mode 438

Automotive Attack Database 78

Automotive Ethernet 224

Automotive Ethernet penetration testing **253**
DoIP enumeration and exploitation 254
network discovery and traffic capture 253
SOME/IP testing 254
Automotive Ethernet penetration testingg
TLS configuration analysis 255
XCP over Ethernet testing 255
Avalanche Photodiode (APD) **413, 414**
Azure Security Center for IoT **93**
access control policies **272**
acoustic command injection **428**
active reconnaissance **59**
back-end and application API analysis 62
ECU enumeration and diagnostic discovery 62
network scanning and port enumeration 62
port scanning 63
techniques 60
test bench, building 60
wireless networks, mapping in automotive ecosystems 61
ad-hoc security approach **460**
compliance-only mindset 460
limitations 461, 462
misconceptions 462
PSIRT 461
advanced persistent threats (APTs) **18**
aircrack-ng **95**
airodump-ng **192**
anvilsecure/nanopb-decompiler **446**
application layer, in-vehicle network protocols **229**
secure onboard communication (SecOC) 230
SOME/IP 230
unified diagnostic services (UDS) 230
universal measurement and calibration protocol (XCP) 230
artifact analysis **63**
AI-enhanced fuzzing 68
AI-enhanced SAST 64
dynamic analysis 67
firmware reverse engineering 65, 66
fuzzing 67
information flow analysis 63
static code analysis 63
version-based vulnerability scanning 66
attack chains **71, 72**
at vehicle vs. ECU level 77
common attack pattern enumeration 73
frameworks for building 73
attack methods **78**
attack phase **71**
attacker levels **19**
criminal groups 20
ethical hackers 20
motivated individuals 20
nation-state actors 21
organized crime 20
researchers 20
script kiddies 19
attacker profile **18**
developer's psychology 21, 22
psychological tactics 22
attention (AT) commands **188**
authorization testing **149**
automotive context **75, 76**
autonomy CAN bus **263**

B

BUSMASTER **105, 248**
Ball Grid Array (BGA) rework station **160**
Basic Safety Messages (BSMs) **183**
Bettercap **97**
BinaryAI **114**

Binwalk **113, 147, 194**
BladeRF **203**
BlueZ **97, 193, 448**
Bluetooth **181**
Bluetooth Low Energy (BLE) **181, 419**
Bluetooth testing **193**
Branch Target Identification (BTI) **399**
BtleJack **97, 193**
Burp Intruder **148**
Burp Suite **90, 145 – 154**
Bus Pirate **111**
backend connectivity protocols **138**
- AMQP 139
- CoAP 139
- diagnostics protocols 140
- HTTP/REST APIs 139
- MQTT 140
- SMS and legacy telephony 139
- WebSockets 139
- XMPP 139

black box penetration test **36**
bluetoothctl **193**
boofuzz **91, 148**
btmgmt **193**
buffer overflow **189**
bus-off attack testing **249**

C

CAN and CAN FD networks **104**
CAN bus **50**
CAN interface tools **147**
CAN-C bus **267**
CAN-IHS bus **267**
CANToolz **105, 106**
CANalyzer **105, 108**
CANtact **104**
CANvy **248**
- bus-off attack 249
- CAN bus fuzzing 249
- ISO-TP testing 250
- message injection and spoofing 248, 249
- passive bus monitoring and traffic analysis 248
- replay attacks 249
- SAE J1939 transport protocol testing 250
- SecOC bypass testing 252
- UDS diagnostic exploitation 251, 252
- XCP testing 252

CAPEC **73, 74**
CI Fuzz Spark **114**
CI/CD pipelines **144**
CMU software stack
- vulnerabilities 380, 381

CONNECT command **255**
CSMA/CR **227**
CSMS integration, with engineering lifecycle
- cybersecurity concept 469
- design and implementation 470
- end of life and decommissioning 484, 485
- operations and maintenance 483, 484
- product release 480, 481
- production planning 481 – 483
- secure implementation 477, 478
- security design principles and patterns 472, 473
- TARA 468
- validation testing 479, 480
- verification testing 479

Camera Serial Interface (CSI) **437**
Car Connectivity Consortium (CCC) Digital Key protocol **181**
Caring Caribou **106**
Censys **88, 146**
CertificateInParts **448**
Chameleon Mini **100, 193**
ChipShouter **112**
ChipWhisperer **112**
Common Attack Pattern Enumeration and Classification (CAPEC) **27**

Common Vulnerabilities and Exposures (CVEs) **26**
Common Vulnerability Scoring System (CVSS) **26**
Common Weakness Enumeration (CWE) **27, 74**
Compute Unified Device Architecture (CUDA) kernels **438**
ConnMan **206**
Connectivity Master Unit (CMU) **377**
Constant False Alarm Rate (CFAR) **413**
Control Flow Integrity (CFI) **448**
ControlDTCSetting **235**
Controller Area Network (CAN) **13, 137, 181, 411**
Convolutional Neural Networks (CNNs) **416**
Counterfit **119**
Crackle **97**
Cybersecurity and Infrastructure Security Agency (CISA) **27**
Cyclic Redundancy Check (CRC) **227, 418**
camera systems **412**
can-utils **104**
candump **248, 433**
cantools **105, 248**
cantx service **438, 439**
caringcaribou **248**
cellular connectivity **182**
cellular network manipulation **203**
central gateway **247**
command-and-control (C2) **376**
commercial off-the-shelf (COTS) **25**
common weaknesses, in-vehicle communications
Automotive Ethernet 237 – 244
CAN 231 – 236
LIN 245
network topology considerations 246, 247
control CAN bus **263**
cryptographic authentication tag **451**
curl **148, 197**
cybersecurity assurance **33**
cybersecurity interface agreement **467**
cybersecurity management system (CSMS) **28, 459**

D

DasSteeringControlMessage **438, 439**
Data Execution Prevention (DEP) **374**
Data Identifiers (DIDs) **251**
Data Length Codes (DLC) **248**
Database CAN (DBC) **248**
Deep Neural Network (DNN) **416**
Denial-of-Service (DoS) attacks **167, 183**
Desktop Bus (D-Bus) service **266**
Device Identifier Composition Engine (DICE) **397**
Diagnostic Trouble Codes (DTCs) **262, 359**
DiagnosticSessionControl **251**
Diagnostics over Internet Protocol (DoIP) **140, 229**
Distributed Denial of Service (DDoS) attacks **74**
DoIP routing activation authentication **273**
Domain Name System (DNS) **208**
Dynamic Host Configuration Protocol (DHCP) **208**
Dynamic Random-Access Memory (DRAM) **65**
DynamoDB **138**
data buffer **447**

data link layer, in-vehicle network protocols **227**
- CAN data link 227
- Ethernet data link 228
- FlexRay data link 228
- LIN data link 228

data poisoning attack **453**
defense-in-depth principle **168, 472**
defensive security **4**
device shadow **138**
diagnostic scan tools **106**
discovery phase, penetration test **53**
- active reconnaissance 59
- artifact analysis 63
- dead ends, navigating 70
- hardware reconnaissance 69
- passive reconnaissance 56
- practice 55
- trust mapping, in embedded systems 53, 54

distributional shift **427, 452**
domain separation **472**
double-tagging **239**

E

ECU-attached sensors **409, 410**
Eclipse Mosquitto **93, 146**
Electric Power Assisted Steering (EPAS) **438**
Electric Power Steering (EPS) **267**
Electrical/Electronic (E/E) architecture **3**
Electronic Control Units (ECUs) **6, 49, 134, 408**
Engine Control ECU **264**
Entity Identifier (EID) **239, 240**
Error Correction Code (ECC) **161**
Ethernet **14**
Ettercap **108, 253**
Executable and Linkable Format (ELF) **65**
Extended Diagnostic Session **251**
Extensible Messaging and Presence Protocol (XMPP) **139**
embedded Subscriber Identity Module (eSIM) **161, 182**
externally mounted sensors **409 – 411**
- architecture 415
- camera systems 412
- LiDAR systems 413, 414
- radar systems 413
- tire pressure monitoring systems 411
- ultrasonic systems 414

F

Fiddler **90, 149 – 151**
Field Bus Exchange (FIBEX) **245**
File Transfer Protocol (FTP) **190**
FindService message **240, 254**
Firmware Over-the-Air (FOTA) servers **205**
Flat Panel Display (FPD) **346**
FlexRay penetration testing **256**
- frame injection and timing manipulation 257
- passive traffic analysis and schedule reconstruction 256

Flipper Zero **96**
Foolbox **119, 432**
Frequency-Modulated Continuous Wave (FMCW) signal **413**
Frida **113, 145 – 147**
femtocell **195**
firmware reverse engineering **65**
firmware-mod-kit **194**
flashrom **112**
fleet-wide correlation **453**
fuzzing **67, 196**

G

GET_DAQ_LIST_INFO command **252**
GET_SEED requests **252**
GNSS testing **194**
GNSS-SDR **100**
GNU Radio **118, 193, 197, 430 – 432**
GPS Exchange Format (GPX) **259**
GPS-SDR-SIM **100**
Garak **120**
Gateway ECU **264**
General Data Protection Regulation (GDPR) **168, 261**
Generic Attribute Profile (GATT) **181, 446**
Ghidra **113, 147, 196**
Gigabit Multimedia Serial Link (GMSL) **346**
GitGuardian **89, 167**
GitLeaks **89, 147**
Global Positioning System (GPS) **440**
GraphQL **138**
gatttool **193**
generation-based fuzzers **67**
generation-based fuzzing **249**
global navigation satellite system (GNSS) **183**
gr-tpms **118, 430**
gray box penetration test **37**

H

HPC compromise, case studies **377**
- Mazda IVI 377 – 382
- Tesla IVI 382 – 386
- Tesla Pwn2Own 2023 386 – 394

HTTP/REST APIs **139**
HackRF **193, 199**
HackRF One **98, 119, 430**
Hands-Free Profile (HFP) **181**
Hardware Platform Security Engine (HPSE) **355**
Hardware Security Module (HSM) **49, 167, 271**
Hardware-in-the-Loop (HIL) system **36**
Heating, Ventilation, and Air Conditioning (HVAC) **267**
High-Performance Computers (HPCs) **65, 343**
- best practices, for securing 395 – 401
- hardware components 345 – 347
- hardware weaknesses 351 – 358
- penetration testing techniques, exploring 370 – 376
- software components 347 – 350
- software weaknesses 359 – 369
- supply chain compromise 376

hacktivists **20**
hardware reconnaissance techniques **69**
- fully invasive and destructive attacks 69
- non-invasive attacks 69
- semi-invasive or invasive fault-injection attacks 69

hashcat **96, 204**
hcitool **193**
head-of-line blocking condition **191, 192**
high-performance ECUs **110, 344, 345**
high-performance and MCU-based ECU tools
- AI-augmented tools 114 – 117
- fault injection and side-channel research tooling 112
- firmware acquisition and reverse engineering 112, 113
- fuzzing and kernel/system stress testing 114
- hardware interface discovery and debug access 111

hobbyists **19**

I

IAR Workbench **268**

IDA Pro **113, 147, 196, 268, 446**
IDOR with read access **156**
IDOR with write capability **156**
IEEE 802.1X **239**
ISO 21434
pentesting 30 – 32
ISO-TP **228**
ISO-TP testing **250**
Image Signal Processor (ISP) **412**
Initialization Vector (IV) value **162**
Input Output Controller (IOC) **267**
Insecure Direct Object Reference (IDOR) **155**
Inter-Integrated Circuit (I2C) **412**
InternalBlue **97**
International Mobile Equipment Identity (IMEI) **161**
Internet Protocol (IP) address **139**
Internet Protocol Security (IPsec) **229**
Internet of Things (IoT) **133**
Intrusion Detection System (IDS) **274**
i2c-tools **111**
i2cdetect **431**
i2cdump **431**
i2cset **432**
in-vehicle communications
common weaknesses 230, 231
overview 224
security best practices 271 – 273
in-vehicle network protocols **225**
application layer 229
data link layer 227
network and transport layers 228
penetration testing 248
physical layer 226
in-vehicle network tools **104**
AI-augmented tools 108 – 110
Automotive Ethernet access and mixed-protocol tools 107
CAN and CAN FD networks 104
SocketCAN-based workflows 104
information security assessment **29**
intellectual property (IP) **227**
iocupdate binary **268**
iw **192**

J

JBL CAN injection theft device **263**
JSON Web Tokens (JWT) **139**
JTAGulator **111**
Jadx **89, 147**
Japan Computer Emergency Response Team (JPCERT) **27**
Joint Bi-level Image Group 2 (JBIG2) **26**
Joint Test Action Group (JTAG) **65, 147, 160**
jwt_tool **91, 149**

K

Kalman filter tracking system **444**
Kayak **105, 248**
Kismet **96, 192**
KunnaEmu **208**
known risks **32**

L

LIN Bus tools **106, 107**
LIN penetration testing **256**
master impersonation and frame injection 256
passive monitoring and frame analysis 256
LiDAR systems **413**
Light Detection and Ranging (LiDAR) **225, 413**

Local Interconnect Network (LIN) **411**
Lua **268, 269**
lateral movement **76**
least functionality **472**
least privilege principle **472**
libFuzzer **196**
logical address **240**

M

MAC spoofing **244**
MAC table overflow **244**
MCU-based ECU components
communication interfaces and buses 288
debug and programming ports 289
direct I/O and sensor interfaces 289
internal peripherals 289
memory 288
microcontroller 288
MCU-based ECUs **110, 284 – 286**
attack chain construction 318 – 324
AUTOSAR basic components 293 – 299
AUTOSAR DLT 304
AUTOSAR runtime environment (RTE) 301, 302
best practices 334 – 337
bootloaders and boot process 291, 292
calibration and debugging code 303
case studies 324 – 327, 330
hardware and software anatomy 287, 288
hardware and software weaknesses 304 – 317
HPSE firmware 292, 293
penetration testing techniques 318 – 324
software architecture and execution model 290
MITRE ATT&CK framework **74, 75**
MQTT Explorer **91, 146**
Magisk **90, 147**
Master Service Agreement (MSA) **51**
Mazda IVI **377 – 382**
Mbed-TLS **446**
Media Access Control Security (MACsec) **228, 451**
Memory Management Unit (MMU) **448**
Memory Protection Unit (MPU) **448**
Message Authentication Code (MAC) **230**
Message Queuing Telemetry Transport (MQTT) **135, 140**
Metasploit **107, 197**
Microelectromechanical Systems (MEMS) inertial sensors **410**
Microsoft Azure **134**
Microsoft Defender for IoT **93**
MobSF (Mobile Security Framework) **89, 147**
Mobile Industry Processor Interface (MIPI) **346**
MultiMediaCard (MMC) partition **268**
machine learning (ML) model **134**
man-in-the-middle (MitM) attacks **96**
managed security approach **463, 464**
acquisition and supplier management 467
configuration management 464
documentation management 464
integration, with engineering lifecycle 468
planning 467, 468
quality management 464, 465
requirements management 464
supply chain 465, 466
tool management 465, 466
masscan **89, 193, 203**

media access control (MAC) addresses 228
mediated access 473
microcontroller unit (MCU) 227, 345
minicom 111, 195
mitmproxy 90, 146, 253
multi-factor authentication 165
mutation 138
mutation-based fuzzers 67
mutation-based fuzzing 249
mutual TLS (mTLS) 140, 242

N

NIST framework
leveraging, for penetration testing 29
National Vulnerability Database (NVD) 27, 196
NavTrailService D-Bus service 268
Near-Field Communication (NFC) 99, 175, 183, 446
interfaces 193
Negative Response Code (NRC) 251
Nmap 107, 146, 193 – 197, 253
Notification message 241, 254
nanopb 446
native VLAN 238
netcat 193, 197
network and transport layers 228
CAN network and transport 228
Ethernet network and transport 229
FlexRay network and transport 229
LIN network and transport 229
network daemon 189
network segmentation 272
network-attached sensors 409 – 411
networking protocols
controller area network (CAN) 13
Ethernet 14
local interconnect network (LIN) 13
nondisclosure agreements (NDAs) 51

O

OBD-II connector 239
OCB mode 191
OWASP Dependency-Check 92, 149
OWASP ZAP 90, 146 – 149
OfferService message 240, 254
On Path Diagnostic - Intrusion and Inference (OPD-II) 260
On-Board Diagnostics (OBD) interface 55
On-Board Diagnostics (OBD) port 181
Open Authorization (OAuth) 139
Open Source Intelligence (OSINT) 146
Open-Source Software (OSS) 189
OpenBTS 99, 195
OpenLTE/Open5GS 99
OpenOCD 111, 160
OpenSSL 162
OpenSSL s_client 90
OpenStreetMap 260
Osmocom suite 99
Osmocom tool 195
offensive security 3
motivation 4
one-time programmable (OTP) memory 351
one-time-programmable (OTP) memory 74
opendbc 249
original equipment manufacturers (OEMs) 133
oss-fuzz-gen 114

out-of-bounds write **189, 447**
over-the-air (OTA) **133, 376, 427**

P

Packet Capture (PCAP) file **91**
Pacu **92**
Park Assist Module (PAM) **267**
Particle Swarm Optimization (PSO) **439**
Passive Keyless Entry and Start (PKES) **182**
Peach Fuzzer **92, 148**
Phone Book Access Profile (PBAP) **181**
Pineapple (Hak5) **96**
PlateToVin.com **155**
Pointer Authentication Codes (PAC) **374, 399**
Postman **90, 145, 148 – 150**
Printed Circuit Board (PCB) **429**
adapters 160
Product Security Incident Response Team (PSIRT) **81**
Proman NAND reader **160**
Protected Identifiers (PIDs) **256**
Protocol Buffer (Protobuf) **446**
Protocol Data Units (PDUs) **230**
Prowler **92**
Proxmark3 **99, 193**
Public-Key Cryptography Standards #12 (PKCS#12) container **162**
Python scripts **148, 268**
passive reconnaissance **56**
baseline traffic capture 59
passive eavesdropping 59
passive wireless observation 58, 59
public and commercial information sources 57, 58
penetration test planning phase **48**
components, prioritizing 52
deliverables, defining 52, 53
engagement, defining 48
information sharing and test types 50
legal considerations and documentation 51
reporting 52, 53
scope, defining 49
testing conditions, defining 49, 50
penetration tester
black hat hackers 42
blue team 42, 43
red team 42
selecting 40, 41
white hat hackers 41
penetration testing **28, 145**
AI impact 121, 122
API exploitation and input manipulation 148
authentication 149
authorization 149
cloud reconnaissance and misconfiguration exploitation 146
communication testing and protocol analysis 146
cybersecurity assurance 32 – 35
exploring 370 – 376
privilege escalation 149
secret discovery and credential abuse 147
supply chain and integration security testing 149
penetration testing phases **37**
attack phase 39
discovery phase 38
enumeration 39
information gathering 38
planning phase 38
reporting phase 40
scanning 39
penetration testing, in-vehicle network **248**
Automotive Ethernet 253
CANvy 248

FlexRay 256
LIN 256
penetration testing, vehicle sensors and AI systems **429**
AI assistant and LLM testing 435
camera system testing 431
LiDAR testing 433
MEMS inertial sensor testing 429, 430
radar testing 432
TPMS testing 430, 431
ultrasonic testing 434
penetration testing, wireless vehicle systems **192**
AT command injection and control channel abuse 195
authentication weaknesses and secret extraction 196
cellular module compromise and privilege analysis 194
network services and debug functionalities, exploiting 197
private APN lateral movement and network abuse 195
RF and rolling code attacks 197
vulnerabilities, finding 196
wireless attack surface discovery and validation 192 – 194
wireless protocol-level attacks 198
penetration tests **35**
black box penetration test 36
gray box penetration test 37
white box penetration test 35, 36
pentester **250**
perfect forward secrecy **166**
persistent backdoors **205**
physical layer, in-vehicle network protocols **226**
Automotive Ethernet PHY 227
controller area network physical layer (CAN PHY) 226
FlexRay PHY 227
local interconnect network physical layer (LIN PHY) 226
pivoting **76**
plausibility checking **451**
point cloud **413, 414**
port-based access control **272**
powerpc-eabivle toolchain **448**
powertrain CAN bus **263**
prevent VLAN hopping **272**
principle of least privilege **165 – 167, 452**
privacy-by-design principle **168**
prompt injection exposure **428**
pulse fingerprinting **423**
purple teaming **43**
pySerial **195**
python-can **106, 248**
python-isoTp **106**

Q

QEMU **113, 208**
Quality of Service (QoS) levels **140**

R

RF-based RKE systems **193**
RTL-SDR **199**
RTL-SDR V3 **118, 430**
Radare2 **113, 196**
Radio Frequency (RF) channels **175**
Radio-Frequency Identification (RFID) **99**
Random Access Memory (RAM) **252**
Range-Doppler map **413**
ReadDataByIdentifier **251**
ReadMemoryByAddress **251**
Receive Error Counter (REC) **232**
Remote Keyless Entry (RKE) **182**

Representational State Transfer (REST) **139**
Request message **241, 254**
Return-Oriented Programming (ROP) **209**
Riscure Inspector **112**
Role-Based Access Control (RBAC) **165**
RoutineControl StartRoutine **252**
Routing Activation Request **240**
Rules of Engagement (ROE) **48**
radar systems **413**
rain probability score **439, 440**
random fuzzing **249**
rate limiting **167**
real-life attacks on in-vehicle communications, case studies **257**
- CAN injection attack 261 – 265
- remote exploitation, of passenger vehicle 265 – 271
- vehicle routes, reconstructing 257 – 261

real-life attacks on vehicle backends, case studies **150**
- insecure vehicle backend API design, exploiting 150 – 153
- remote vehicle control, through backend compromise 157 – 164
- remote vehicle takeover, via vehicle backend vulnerability 153 – 157

real-life attacks on vehicle sensors and AI/ML systems, case studies **436**
- adversarial machine learning attacks on Tesla Autopilot 436 – 441
- LiDAR spoofing, via pulse recording and replay 441 – 444
- zero-click remote code execution, via TPM sensors 445 – 449

real-life attacks on wireless vehicle systems, case studies **198**
- 4G module hacking 201 – 205
- automotive remote keyless entry systems, cloning via RF rolling code exploitation 198 – 201
- Zero-click remote code execution, on infotainment system 206 – 211

reduced complexity **472**
relay attack **443**
remote code execution **204, 205**
remote vehicle command control endpoint **163**
residual risks **32**
revalidation testing **82**
revocation checking **243**
root cause analysis **82**
rotating mechanical sensors **414**
rtl_433 **118, 430**
runtime anomaly detection **453**

S

SAE J1939 **229**
- testing 250

SEGGER's J-Link **111**
SHORT_DOWNLOAD **252**
SHORT_UPLOAD **252**
SMS-based AT command handling **203**
START_STOP_DAQ_LIST **252**
SavvyCAN **105, 248**
Scout Suite **92**
SecOC bypass testing **252**
Secure Shell (SSH) **190**
Security Access **251**
Security Operations Center (SOC) **167**
SecurityAccess RequestSeed **251**
SecurityTrails **88, 146**
Separation Time Minimum (STmin) **233**

Serial Peripheral Interface (SPI) **268**
Serial Wire Debug (SWD) **147**
Serializer-Deserializer (SerDes) **412**
Server Side Request Forgery (SSRF) **158, 163**
Service Set Identifier (SSID) **192**
Service-Oriented Vehicle Diagnostics (SOVD) **140**
Shodan **88, 145, 146**
Short Message Service (SMS) **139, 188**
Single Sign-On (SSO) system **144**
Single-Photon Avalanche Diode (SPAD) **413, 414**
Smart Key Receiver ECU **263**
SocketCAN-based workflows **104**
Software Composition Analysis (SCA) tools **196**
Software-Defined Radio (SDR) **98, 430**
Software-Defined Vehicle (SDV) **6, 409**
Statement of Work (SOW) **51**
Stingray simulators **99**
Structured Query Language (SQL) injection **142**
SubscribeEventgroup message **254**
System-on-Chip (SoC) **54, 227, 345, 415**
Syzkaller **114**
scalable service-oriented middleware over IP (SOME/IP) **230**
scanning
active scanning 39
passive scanning 39
scapy **106, 107, 210, 248**
scapy-radio **101**
script kiddies **19**
secure by default **473**
secure onboard communication (SecOC) **230**
security bar, in automotive systems
attacker's landscape 486
automotive security, elevating 487
human resource gap 486
process pressures, in rapid development cycles 485
raising 485
technical challenges, of centralized architectures 486
security best practices, vehicle backend infrastructure
communication, securing 166
default credentials, eliminating 167
defense-in-depth principle 168
detection 167
incident response 167
input validation and secure API development 166
monitoring 167
secrets, protecting 167
segmentation and network security 167
sensitive data and privacy, protecting 168
strong authentication and access control 165
supply chain and third-party management 168
security best practices, vehicle sensors and AI **450**
AI systems, designing with adversarial robustness 451
Defense-in-Depth, applying 450
in-vehicle AI assistants, protecting 452
sensor data, authenticating at interface level 451
sensor security, addressing 452, 453
security best practices, wireless vehicle systems **212**
authentication, designing 213, 214
authorization, designing 213, 214

credentials and cryptographic keys, protecting 214
designing, for containment 215
input validation 215
interfaces, treating as untrusted entry points 212
legacy control paths, eliminating 214
partial compromise, assuming 215
security, building into development lifecycle 216
strong isolation, enforcing 213
third-party components 216
unnecessary services, eliminating 214
unsafe fallback mechanisms 214
wireless-facing software, hardening 216
workflows, provisioning 214

security design patterns
authenticator 475
chain of trust 473
defer to kernel 475
distrustful decomposition 473
identity provider (IdP) 474
privilege separation 475
secure chain of responsibility 477
secure strategy (policy factor) 476
security builder factory 476
trusted third party (TTP) 474

security groups 167
self-signed certificates 242, 243
sensor fusion logic 450
sensor identity authentication 451
signature verification 190
solid-state sensors 414
spoofing attacks 443
srsRAN 98
sslyze 91, 146, 253
stack canaries 209
startIndex 448
static analysis security testing (SAST) 373
strategic decision making 71
strings 194
supply chain attacks 376

T

TP.CM_BAM 234
TP.CM_RTS 234
TPMS auto learn mechanism 448
Telematics Control Unit (TCU) 139
Telematics Service Provider (TSP) 163
Tesla IVI 382 – 386
Tesla Pwn2Own 2023 386 – 394
TeslaProtobufs project 447
TesterPresent 235, 240
Threat Analysis and Risk Assessment (TARA) 31, 452, 468, 469
Time Division Multiple Access (TDMA) 228
Time-Sensitive Networking (TSN) 224
Tire Pressure Monitoring Systems (TPMS) 411
Toyota RAV4 stolen 261
Transmission Control Protocol (TCP) 91, 139, 229
Transmit Error Counter (TEC) 232
Transport Layer Security (TLS) 140, 229, 446
Trivy 92, 149
Type-Length-Value (TLV) 241
tcpdump 91, 146, 253
test bench 60
threats 23, 27
time-of-check-to-time-of-use (TOCTOU) 78, 373
training distribution 426
truffleHog 89, 147
trunk port 238
trust boundaries 53

type confusion **241**

U

UART interface **159, 160**
UBI reader **203**
UDS Security Access **273**
UDS service IDs **251**
UDSim **106**
USRP (Universal Software Radio Peripheral) **98, 199**
UWB testing **193**
Ubertooth **193**
Ubertooth One **96, 431**
Unified Diagnostics Services (UDS) **446**
Unique Identifier (UID) **200**
Universal Radio Hacker (URH) **118, 193, 197, 430**
Universal Serial Bus (USB) interface **159, 160**
User Datagram Protocol (UDP) **91, 229, 438**
ultra-wideband (UWB) **182**
ultrasonic sensor systems **414**
unified diagnostic services (UDS) **230**
universal measurement and calibration protocol (XCP) **230**
unknown risks **32**
unmitigated risks **33**
untagged traffic **239**

V

V2X attack surface discovery **194**
V2X communication **183**
- Vehicle-to-Infrastructure (V2I) 183
- Vehicle-to-Networks (V2N) 183
- Vehicle-to-Pedestrians (V2P) 183
- Vehicle-to-Vehicle (V2V) 183

VLAN hopping **239**
Valhalla routing engine **260**
Vanetza **100**
Variable Length Encoding (VLE) **446**
Vehicle Control Security (VCSEC) module **207**
Vehicle Controller Secondary (VCSEC) ECU **446**
Vehicle Identification Number (VIN) **239**
Vehicle Security Operations Center (VSOC) **167, 274, 453**
Virtual Local Area Network (VLAN) **228**
Virtual Private Cloud (VPC) **167**
VulHunt **114**
vehicle E/E architecture
- anatomy 5
- centralized zonal-based architecture 8 – 10
- domain-based architecture 6 – 8
- electronic control units (ECUs) 11
- external vehicle interfaces 11
- networking protocols 13
- sensors 12

vehicle backend architecture
- broken authentication and authorization controls 143
- cloud infrastructure flaws 144
- input validation weaknesses 142
- insecure API design 142
- insecure communication 144
- overview 134 – 136
- security best practices 165 – 168
- security weaknesses 141
- supply chain and third-party integration weaknesses 145
- weak secrets or credential management 143

vehicle backend tools **88**
- AI-augmented tools 93 – 95
- API workflow reconstruction 90

cloud posture assessment and cloud exploitation simulation 92
external surface discovery and exposure validation 88, 89
inspection 90
interception 90
reference implementations and vendor tools 93
vehicle companion apps and backend APIs 89, 90

vehicle electrical/electronic (E/E) architecture 224

vehicle sensor tools 117
AI-augmented tools 119 – 121
RF signal capture and protocol analysis 117

vehicle sensors and AI
AI and ML systems 416
common weaknesses 417 – 429
ECU-attached sensors 410
interfaces 409
network-attached sensors 411
overview 408
penetration testing 429
security best practices 450
xxternally mounted sensors 411

vehicle unlock use case example 136 – 138

vehicle's operational environment 15 – 17

virtual machine (VM) 347

vulnerabilities 23 – 25
scores 26

vulnerabilities reporting 79
communication triggers 79
communication, during te penetration test 79
findings, presenting 80
findings, structuring 80
post-remediation activities 81, 82
reports, securing 81
reports, storing 81

vulnerabilities, wireless vehicle systems 184
AT command interfaces 188
cellular module architecture and trust boundaries 187, 188
network isolation 188
network services and debug interfaces 190
private APN failures 188
RF systems and rolling code cryptography 191
software vulnerabilities 189
unsafe control paths 188
weak credentials and hard-coded secrets 190
wireless attack surface, expanding 185 – 187
wireless protocol design limitations 191, 192

vulnerability elimination 493

W

Web Application Firewall (WAF) 167

WebSockets 139

Wi-Fi 180

Wi-Fi dongles 180

Wi-Fi testing 192

Wired Equivalent Privacy (WEP) 191

Wireshark 99, 107, 146, 194, 253, 431 – 433

Worley noise patterns 439, 440

WriteMemoryByAddress 252

weaknesses 23

white box penetration test 35, 36

wireless interface tools 95
AI-augmented tools 101
Bluetooth/BLE reconnaissance and experimentation 96, 97
cellular lab networks and rogue network testing 98, 99
GNSS/GPS simulation and test signal generation 100
RFID/NFC analysis tools 99
V2X prototyping and message stack tooling 100

Wi-Fi discovery and client-side manipulation 95, 96

wireless vehicle systems
Bluetooth 181
cellular connectivity 182
common weaknesses 184
global navigation satellite system (GNSS) 183
near field communication (NFC) 183
overview 176, 177
penetration testing 192
protocols 177 – 180
radio frequency (RF) 182
security best practices 212 – 216
ultra-wideband (UWB) 182
vehicle-to-everything communication (V2X) 183, 184
vulnerabilities 184
Wi-Fi 180, 181

X

X.509 certificates 141
XCP Data Acquisition Mode (DAQ) 236
XCP GET_SLAVE_ID broadcast 255
XCP on CAN (XCPoC) 235
XCP on Ethernet (XCPoE) 235

Y

Yard Stick One 119, 193
YateBTS 203
Yersinia 108

Z

Zero-Trust Architecture (ZTA) 165
zero-click attacks 189, 190
zero-click remote code execution exploit 206
zone ECU 246, 247