Practical Detection Engineering with Sigma

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

Wojciech Ciemski

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

Table of Contents

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.

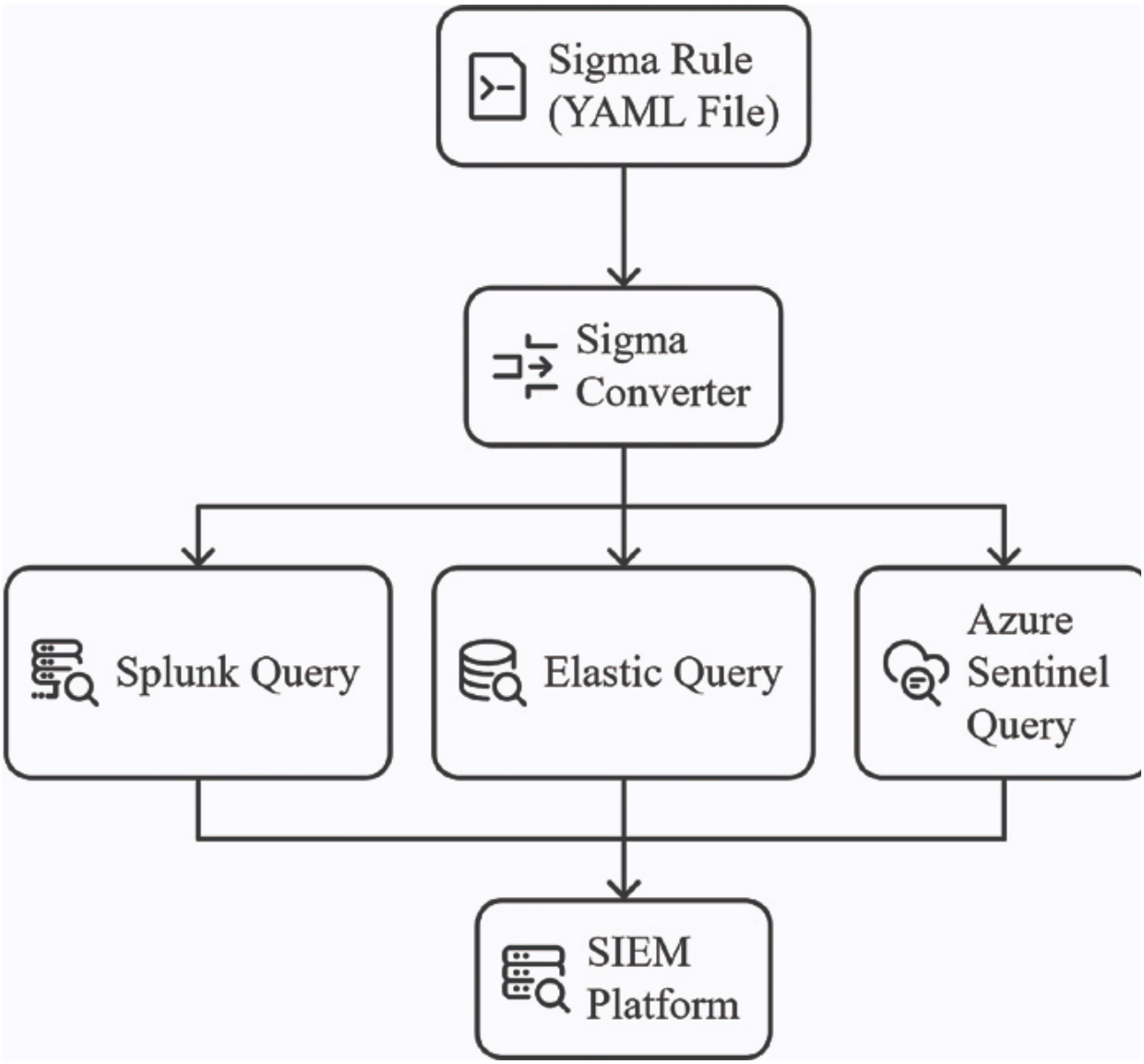

Figure 1.1: *Sigma's vendor-agnostic design—one generic Sigma rule (YAML file) can be converted by Sigma's tools into multiple SIEM-specific queries. In this example diagram from the Sigma project, a single detection rule in YAML passes through the Sigma converter, and yields queries for different SIEM 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.