SIGMA: A UNIVERSAL DETECTION RULE LANGUAGE

Version	Date	Change Description
1.0	2025-01-02	Initial version of the document.
1.1	2025-01-12	Added license information. Document slightly rewritten. Added watermark.
1.2	2025-01-22	Added bibliography, Practical Use Cases, Security Advantages
1.3	2025-01-30	Document slightly rewritten.
1.4	2025-01-31	Added case studies
1.5	2025-06-07	Removed watermarks for better readability. Added cover. Content check performed. Updated usage licenses.
1.6	2025-06-10	The bibliography was checked and errors were removed. All rules have been reviewed and rewritten for clarity throughout the document. Improved text formatting of the rules code.
2.0	2025-06-22	The document has been slightly rewritten while maintaining the previous editorial style and preserving current knowledge. The number of pages has been reduced.
2.1	2025-06-23	Editorial changes have been made to the document.
2.2	2025-08-28	Author sections added and document checked.

License and Disclaimer

Permission is hereby granted to copy and distribute this e-book under the following terms and conditions:

1. **Attribution**
   You must retain the author's name (or pseudonym) and the original title of the e-book on all copies. Any distribution must clearly attribute the work to the original author.

2. **No Modification**
   You may not alter, transform, or build upon the content of this e-book in any way. All copies must be distributed in their original, unmodified form, including this license text.

3. **Disclaimer of Liability**
   The author is not liable for any misuse of the information contained in this e-book. All material is provided for educational and informational purposes only. Any actions taken based on the content are solely at the reader's own risk.

4. **No Warranty**
   The e-book is provided "as is," without warranty of any kind, either expressed or implied. The author does not guarantee the accuracy, completeness, or applicability of the information herein, and shall not be held responsible for any errors, omissions, or potential damages resulting from its use.

5. **Governing Law and Dispute Resolution**
   Any disputes arising from or related to this license or the e-book itself shall be governed by the laws of the author's jurisdiction, unless superseded by mandatory legal provisions.

6. **Final Provisions**

   - This license aims to protect both the author's rights and the freedom of access to knowledge.
   - By using, copying, or distributing this e-book, you acknowledge and agree to be bound by these terms.

About the Author

Wojciech Ciemski is a cybersecurity expert with over a decade of hands-on experience in SOC operations, detection engineering, and compliance. As a consultant and vCISO, he has helped organizations strengthen resilience, implement SIEM/XDR platforms, and meet regulatory requirements such as NIS2 and ISO 27001.

He is the author of several **bestselling cybersecurity books** and numerous technical publications, as well as the founder of **Security Bez Tabu®/Security Beyond Taboo®**, a blog followed by houndrets thounsends readers every year.

Recognized as one of the **Top IT Speakers in Poland since 2021**, Wojciech has delivered over houndret talks, including a **TEDx** presentation, where he shared his passion for practical cyber defense and education. In 2024, he was named to the **"40 under 40 in Cybersecurity"** list as the only representative from Poland.

Today, he develops educational and research initiatives under, combining technical expertise with the mission of training the next generation of cybersecurity professionals.

Learn more:

- Blog: https://securitybeztabu.pl | https://securitybeyondtaboo.com
- LinkedIn: https://www.linkedin.com/in/wojciech-ciemski

To those who learn not to impress, but to protect.
This guide is for every blue teamer, SOC analyst, and curious mind who believes that understanding threats begins with understanding the logs.
Created out of love for knowledge — and the belief that persistence always wins.
Never stop. Never settle. Never give up.

- Wojciech Ciemski

Table of Contents

1. Introduction

1.1 Why a Universal Detection-Rule Language Matters

In the realm of Security Information and Event Management (SIEM), each platform historically spoke its own "language." Splunk has its Search Processing Language (SPL), Microsoft's Azure Sentinel and Elastic use their own Kusto/Kibana Query Language (KQL), IBM QRadar relies on Ariel Query Language (AQL), and so on. For years, writing detection rules across different SIEMs felt like translating between ancient dialects – a time-consuming and error-prone exercise. In the early days, detection logic (queries, correlation rules, alerts) had to be rewritten for each SIEM's proprietary format, which hindered the sharing of threat detection techniques. SIEM rule-sets were siloed in vendor-specific databases, making collaboration difficult at a time when up-to-date detections were desperately needed across organizations. Clearly, the industry needed a common tongue for detections – a universal rule language that could work across platforms.

Sigma emerged to meet this need. Sigma is an open, text-based standard for describing log-based detections in a structured way. Think of Sigma as the **"SQL for security detection,"** allowing you to express a log query or alert logic once and then convert it to many platforms – Splunk, Elastic, QRadar, Microsoft Sentinel, and more. In essence, a Sigma rule is a generalized representation of a threat detection that can be translated into the specific query language of your SIEM tool. This portability means a security analyst can write a rule one time and avoid manually re-writing it for each different tool they use. For example, a Sigma rule to detect a particular malware behavior can be automatically converted to an SPL query for Splunk, a KQL query for Sentinel or Elastic, or an AQL query for QRadar, without the author needing to know the ins and outs of each language. Sigma decouples the detection logic from the SIEM implementation, breaking down the barriers to sharing detection intelligence.

Benefits of a Universal Detection Language (Sigma):

- **Standardization:** Sigma provides a consistent rule format across *all* SIEM and log management platforms. Every Sigma rule uses the same fields and structure (YAML fields like title, logsource, detection, etc.), which makes rules easy to read and write. A conversion tool (converter) then translates this standard format into the target SIEM's query syntax. This consistency streamlines detection engineering – once you learn Sigma's schema, you can understand any rule and deploy it anywhere.
- **Collaboration and Community Sharing:** Before Sigma, analysts could only share detection queries with others who used the same SIEM, limiting knowledge transfer. Sigma changed that paradigm. Now, defenders worldwide share Sigma rules on GitHub and other platforms, knowing anyone can apply them regardless of SIEM. This open sharing accelerates our collective response to new threats. Even less-experienced analysts can leverage rules contributed by seasoned experts, bridging skill gaps and leveling up security for everyone. The result is a crowdsourced knowledge base of detections, readily available to any organization.
- **Flexibility and Vendor Independence:** Sigma frees organizations from vendor lock-in. Because detections are written in a SIEM-agnostic way, you can swap out or integrate multiple SIEM tools without losing your catalog of detection rules. For instance, a company migrating from Splunk to Elastic can convert its Sigma rules to the new

platform's queries with minimal effort. This ability to "port" detections helps organizations adopt the tools that make sense for them, without the costly process of rewriting hundreds of rules for each new platform. In short, Sigma future-proofs your detection logic against changes in tooling.

- **Operational Efficiency:** By writing a rule once in Sigma, security teams save time and reduce errors. Analysts can respond faster to emerging threats – when a new attack technique is discovered, someone can publish a Sigma rule for it and *any* team can quickly implement it in their SIEM of choice. This speed and efficiency are a boon for strained SOC teams. Indeed, Sigma rules have been described as a *"boon to accelerate SOC operations"* by allowing one-and-done rule development for multiple systems.

It's no surprise that Sigma has been widely embraced by the community. Since its inception, thousands of Sigma rules have been created and shared publicly, dramatically expanding the detection coverage available to defenders. The official Sigma repository alone contains over 3,000 curated rules (as of 2025) covering a broad range of threats. Many more rules are shared via community projects and vendors. This collective rule base means that when a new threat arises, you can likely find a Sigma rule (or quickly write one) to detect it, and deploy it to your environment regardless of what logging platform you use. **In summary, a universal detection-rule language matters because it enables a "write once, use everywhere" approach** to threat detection, fostering collaboration and agility in an era where speed and shared knowledge are critical to defense.

1.2 Evolution of Sigma (From Early YAML Prototypes to the Current 2.0 Spec)

Sigma was first introduced in 2017 by security researchers Florian Roth and Thomas Patzke as a solution to the SIEM rule-sharing problem. From the very beginning, Sigma was designed as a simple YAML-based format that any analyst could read or write with basic text editing skills. This was deliberate – YAML is human-readable, which lowered the barrier to entry for writing detection rules, and it's flexible enough to express complex logic when needed. In spirit, Sigma took inspiration from other successful security rule languages: *what Snort rules are to network traffic, and YARA rules are to files, Sigma aims to be for logs*. In other words, Roth and Patzke envisioned Sigma as the open standard for writing SIEM detection signatures, analogous to how YARA became the standard for malware file signatures.

Early days (v0.x to v1.x): The initial Sigma specification was fairly minimal, focusing on the core elements needed for log search patterns. A Sigma rule has always consisted of a few key sections – metadata (title, description, author, etc.), a **logsource** definition (to specify which logs the rule applies to), and a **detection** section containing the conditions to match in the logs. This basic structure has remained constant. For example, a simple early Sigma rule might have a logsource of product: windows and service: security (to target Windows Security Event logs), and a detection with a few field checks (like an Event ID and keywords). Even in prototype form, Sigma's design philosophy was to cover the *majority* of common use cases with a straightforward, shareable format, rather than to account for every edge-case of every SIEM. This trade-off – favoring broad applicability over exhaustive feature sets – meant Sigma stuck to things like basic string matching, wildcard patterns, and Boolean logic ("and"/"or" conditions) that would translate well to many backends. Advanced correlation or statistical functions

(which some SIEM-specific languages support) were initially out of scope, ensuring that any Sigma rule could be converted to practically any SIEM query language.

Despite its simplicity, Sigma quickly gained traction. The community began contributing rules for all sorts of attacks – from malware signature detections to suspicious admin activity – using the Sigma YAML format. Over time, the project grew into a large repository of detection rules maintained on GitHub. By leveraging Sigma, defenders could **"harness the community's power to react promptly to critical threats and new adversary trade-craft"**, as one blog put it. In practical terms, when a new threat was discovered, analysts would write a Sigma rule for it and share it, allowing others to immediately benefit by converting that rule into their own SIEM's query. This open approach dramatically improved the industry's ability to disseminate detection techniques. Within just a few years, Sigma's rule set grew to thousands of entries and became a de-facto standard for sharing detections (akin to how ATT&CK became a standard for sharing threat behaviors). As of mid-2025, the Sigma community had contributed over **15,000** rule entries or modifications in the main repository alone – an incredible growth from the handful of YAML prototypes in 2017.

Tooling and ecosystem: Alongside the rules, the Sigma project provided tools to convert and use those rules. Initially, a simple Python converter called **sigmac** was used to turn Sigma YAML into target queries. As Sigma's popularity and complexity grew, the maintainers undertook a complete rewrite of the toolset in 2023, resulting in **pySigma** and the Sigma CLI. This modern toolchain improved conversion accuracy and added support for many more target platforms. It also set the stage for enhancements to the Sigma rule format itself, because the converter became powerful enough to handle more complex rule logic. In essence, by 2023 Sigma had matured from a rough YAML schema with a basic script, into a full-fledged framework with a specification, a rich library (pySigma), and integration into various SIEM products and platforms (some SIEM vendors even directly support Sigma or provide built-in converters).

Sigma 2.0 – the next generation: The evolution of Sigma's specification culminated in the release of **Sigma Spec v2.0** in August 2024. This was a milestone update reflecting the lessons learned over years of community use. The Sigma team introduced v2.0 to expand the language's expressiveness while keeping it user-friendly. Notably, Sigma 2.0 brought *new rule fields, improved detection primitives, and entirely new rule types* to cover scenarios the early format couldn't easily handle. According to the Sigma maintainers, users had been asking for certain capabilities, and the growth of the project's vision necessitated this evolution. Here are some key enhancements in Sigma 2.0:

- **New metadata fields:** The spec added fields like **taxonomy** and **scope** to enrich rule metadata. *Taxonomy* allows organizations to map Sigma's generic field names and values to their own naming conventions or a canonical schema (helpful for log normalization). *Scope* lets a rule specify that it only applies to certain environments – for example, only on domain controllers, or only on Windows hosts with a particular software – making rules more precise in large, diverse environments. These additions help in tailoring and categorizing rules without hard-coding platform specifics into the detection logic.
- **Expanded detection syntax:** Sigma 2.0 introduced new ways to express conditions. One example is *placeholders* – special marker values in rules that get replaced at query time or during conversion. Placeholders can stand for environment-specific constants or even reference external data (like a list of known admin accounts) when the rule is

deployed. Another improvement is the ability to explicitly check for **field existence** (useful if a log field may or may not appear in an event). Sigma also standardized some common placeholder values (for instance, a placeholder that represents "any Windows admin user account") to make rules more portable and easier to configure. On top of that, Sigma 2.0 included a **"keywords"** search feature – a way to specify a list of free-text keywords to search across an entire event log entry (this is akin to a broad text search, useful for finding any of several terms in an unstructured message field).

- **More powerful modifiers:** Sigma rules use *modifiers* to refine how a condition matches. For example, endswith, contains, or regular expression matches on strings are modifiers. Prior to v2.0, Sigma had a basic set of these. The 2.0 spec formalized a **Modifiers Appendix** and expanded the list of modifiers available. Now there are categories of modifiers by data type: string operations (like case-insensitive contains, regex matches, base64 decoding), numeric comparisons (greater than, less than, etc.), IP address operations (CIDR matching), time extraction (e.g. extract hour or month from a timestamp), and more. These give rule writers more flexibility to express complex conditions. For instance, one could write a condition like CommandLine|regex: '^PowerShell.*-Encod.*' to detect a PowerShell command with an encoded command parameter – something that would have been tricky to do cleanly in early Sigma versions.

- **Correlation and sequence rules:** Perhaps the most significant leap in Sigma 2.0 is the introduction of **correlation rules**. Earlier versions of Sigma were mostly limited to single-event patterns (one log at a time). The new correlation specification enables rules that tie multiple events together. Sigma now supports expressing sequences or aggregations such as: "Detect five failed login events followed by a successful login for the same account within 10 minutes" or "Alert if more than 50 distinct hosts report the same error within an hour." These are accomplished with new correlation *types* like **event counting**, **value counting**, and **temporal sequencing**. For example, an *event_count* correlation can specify a threshold of X events in Y time (e.g. >50 failed logins in 3 minutes). A *temporal_ordered* correlation can define an ordered sequence of sub-rules that must occur in a given timeframe (e.g. a privilege escalation event followed by a persistence event within 5 minutes). By linking multiple Sigma rules or conditions, analysts can craft detections for attack **chains** and complex scenarios that single log events alone wouldn't reveal. This moves Sigma closer to the capabilities of some SIEM query languages that support join/sequence operations, all while remaining SIEM-neutral in the rule format.

- **Global filter rules:** Along with correlation, Sigma 2.0 introduced a concept called **Sigma Filters**. These are special rules dedicated to defining **exclusions** (false positive filters) that can be applied across multiple detection rules. The idea is to avoid duplicating the same "noise filters" in many rules. For instance, if an organization knows that a certain service account routinely triggers what would otherwise look like an alert, they can create one Sigma filter rule to exclude that account's activity and then apply that filter to all relevant Sigma rules globally. This centralized approach to false-positive suppression means detection rules can remain broadly applicable (sharing them is easier since they aren't hard-coded with organization-specific exceptions), and the local environment tuning is done in the separate filter definitions. The 2.0 spec defines how to write these filter rules in YAML and how they link to normal rules. This feature was driven

by the recognition that applying not conditions within every rule for local peculiarities was clunky and hard to manage at scale.

- **Formal schemata and consistency:** Sigma 2.0 also delivered a formal JSON schema for the rule format and its extensions. Having a schema means tools can automatically validate Sigma rules to catch format errors or spec deviations. It also helps in programmatically generating or modifying rules. Additionally, the Sigma project cleaned up and restructured its repositories to separate the core specification, the rule sets, and various extensions (correlation, filters, etc.) for easier maintenance and faster releases. All of this indicates a maturing project – Sigma is no longer "just a format" but a standardized language with a well-defined spec and lifecycle.

In summary, Sigma has evolved remarkably from a simple YAML prototype to a feature-rich universal detection language. It started by addressing the basic need of a common format for log queries, and over the years it incorporated community feedback to tackle more advanced detection scenarios while still retaining its core principle of SIEM-neutrality. The current Sigma 2.0 spec represents the state-of-the-art: it balances **power** (through correlations, modifiers, etc.) with **portability** (ensuring rules remain broadly convertible). This evolution demonstrates Sigma's commitment to being *the* go-to language for detection engineering – flexible enough for real-world use, but generic enough to unite a fragmented tool ecosystem. And the journey isn't over; as threats evolve, Sigma's maintainers and community continue to discuss new features and improvements, ensuring the language keeps pace with detection needs.

1.3 Sigma vs. Platform-Native DSLs (SPL, KQL, AQL, …)

While Sigma provides a unified way to **describe** detection logic, it ultimately must be converted into each platform's native query language to actually run on the data. To appreciate Sigma's value, it's helpful to compare how a detection is expressed in Sigma versus how it looks in the native domain-specific languages (DSLs) of popular platforms like Splunk, Elastic/Microsoft, and QRadar. Each of these platforms has its own syntax and nuances:

- **Splunk SPL (Search Processing Language):** Splunk's query language is characterized by piped commands. A typical SPL search starts with selecting data from indexes and then pipes (|) into filters or transforms. For example, a simple Splunk query might be: index=security EventCode=4625 AccountName="admin" to find Windows failed logon events for the admin user. SPL can be very powerful (with sorting, stats aggregation, etc.), but the key point is that it uses its own operators and field names. Splunk expects queries to reference its data model (indexes, source types, fields as named in events). This means a detection rule written for Splunk is usually tied to Splunk-specific concepts.

- **KQL (Kusto Query Language / Kibana Query Language):** Here we actually have two flavors under one acronym. In Microsoft's Sentinel (which uses Azure Log Analytics), **Kusto Query Language** is used – a SQL-like language where you typically specify a table (like SecurityEvent or Syslog) then a where clause for conditions, and possibly pipe into summarization commands. An example Azure Sentinel KQL query might be: SecurityEvent | where EventID == 4625 and Account == "admin". On the Elastic side, **Kibana Query Language** is a simpler Lucene-based syntax for filtering Elasticsearch logs (for instance, event.code:4625 AND user.name: admin). Elastic's newer detection engine also supports its own EQL (Event Query Language) for sequence rules. Both

Microsoft and Elastic queries use text filtering expressions, but again with platform-specific context (e.g. Sentinel's use of specific column names, or Elastic's JSON field names). KQL generally uses C-style operators (==, contains, etc.) for conditions, differing from Splunk's more free-form search syntax.

- **AQL (Ariel Query Language for QRadar):** IBM QRadar's AQL is reminiscent of SQL. Analysts write queries like SELECT * FROM events WHERE <conditions>. For example: SELECT * FROM events WHERE username='admin' AND eventId=4625. AQL queries run against QRadar's event database (named "Ariel"). It has its own functions and quirks (like case-sensitive string comparisons and limited regex support), and it requires knowledge of QRadar's schema (field names, event types). So, a QRadar-specific rule might be formulated quite differently than a Splunk rule, even if they aim to detect the same scenario.

Given these differences, a single detection concept can look quite distinct across SPL, KQL, and AQL. Sigma bridges this gap by acting as an intermediate representation. A Sigma rule is written in a neutral way – referencing a generalized event schema and using abstracted operators – and then a Sigma **backend** translates it to each platform's query language. The backend knows, for instance, that what Sigma calls Image (process name) in a Windows event should become NewProcessName in Splunk's knowledge of Windows Security logs, or perhaps FileName in Elastic's schema, etc. This mapping is how Sigma achieves "write once, run anywhere" for detections.

To make this concrete, let's walk through a real-world detection example and compare how it's expressed in Sigma versus in each native DSL. Consider a common attack technique: an attacker dumping the Windows LSASS process memory using the Sysinternals **ProcDump** tool (to extract credentials). We want to detect when the process **procdump.exe** is executed with **lsass.exe** as an argument. Here's how that detection would be described:

- **Sigma Rule (excerpt):** We specify the log source and the conditions in YAML. In Sigma's generic terms, this is a Windows process creation event where the process image ends with "procdump.exe" and the command line contains "lsass.exe". In the Sigma rule's detection section, it might look like:

```
detection:
  selection:
    Image|endswith: '\procdump.exe'
    CommandLine|contains: 'lsass.exe'
  condition: selection
```

This Sigma snippet says: *find any event where the process name (Image) ends with "procdump.exe" and its command line contains "lsass.exe"*. (The backslash in \procdump.exe is just escaping the path separator in Sigma syntax.) The rule's **logsource** would indicate something like product: windows and category: process_creation to target the right logs (e.g., Windows event 4688 or Sysmon event 1).

- **Splunk (SPL) equivalent:** In Splunk's SPL, assuming Windows Security Event logs are being collected, the query might be:

```
index=* source="WinEventLog:Security" EventCode=4688
    NewProcessName="*\\procdump.exe" CommandLine="*lsass.exe*"
```

This search looks for event ID 4688 (process start) in Windows Security logs where the NewProcessName ends in “procdump.exe” and the CommandLine field contains “lsass.exe”. Notice how the Sigma fields were translated: Sigma’s Image became Splunk’s NewProcessName (the field name used in that log source), and the condition endswith was implemented by using a wildcard * before the procdump.exe (to allow any preceding path). The contains 'lsass.exe' became *lsass.exe* wildcard search in the CommandLine. Splunk’s syntax required quoting and wildcards, but the logic is equivalent. (Sigma’s converter handles inserting the wildcards for startswith/endswith and so on.)

- **Azure Sentinel (KQL) equivalent:** In Azure’s Kusto Query Language, one might query the SecurityEvent table for similar conditions:

```
SecurityEvent
| where EventID == 4688
| where NewProcessName endswith "procdump.exe"
| where CommandLine contains "lsass.exe"
```

This KQL snippet first filters SecurityEvent logs to Event ID 4688, then applies two conditions: one that the NewProcessName ends with "procdump.exe", and another that the CommandLine has "lsass.exe" substring. KQL uses operators like endswith and contains that map closely to Sigma’s modifiers (indeed, the Sigma backend for KQL will output those functions for these modifiers). Note that in KQL the field names and data source (table) need to be known – Sigma’s logsource would guide the converter to target the SecurityEvent table and the appropriate field names (it aligns with Windows events field naming in Azure). If we were writing a similar query in Elastic’s Kibana Query Language, it might look like: event.code:4688 AND process.name:*procdump.exe AND process.command_line:*lsass.exe* – different syntax (colons and wildcards) but semantically the same filter. Sigma’s role is to abstract these differences.

- **QRadar (AQL) equivalent:** In QRadar’s AQL, we could write:

```
SELECT * FROM events
 WHERE EventID=4688
   AND NewProcessName ILIKE '%\procdump.exe'
   AND CommandLine ILIKE '%lsass.exe%'
```

Here we select from the events database with conditions that the EventID is 4688, and use ILIKE (case-insensitive like) to find procdump.exe at the end of the NewProcessName (the % before the backslash means anything before the \procdump.exe path) and lsass.exe anywhere in the CommandLine. Different SIEMs might use slightly different field names – for instance, QRadar might call the process name field something else – but we assume for illustration it’s similar. The Sigma to AQL converter would handle inserting the SQL wildcard % and adjusting for case sensitivity as needed (Sigma by default is case-insensitive unless specified otherwise).

Below is a summary table comparing the Sigma rule logic to each platform’s query:

Detection Logic: *ProcDump used on LSASS*	**Sigma (abstract)**	**Splunk (SPL)**	**Azure Sentinel (KQL)**	**QRadar (AQL)**
Description	YAML rule matching process execution of procdump.exe with lsass.exe argument.	SPL search filtering Windows Event 4688 for procdump usage.	KQL query on SecurityEvent for EventID 4688 and process criteria.	SQL-like query on events for EventID 4688 and matching fields.
Example Expression	`Image	endswith: '\procdump.exe' CommandLine	contains: 'lsass.exe'`	EventCode=4688 NewProcessName="*\\procdump.exe" CommandLine="*lsass.exe*"

Citations: The Sigma rule excerpt is adapted from a community Sigma rule for detecting LSASS dumps via ProcDump. The Splunk SPL query shown is the actual translation of that Sigma rule into Splunk's syntax (as generated by a Sigma conversion tool).

As the comparison illustrates, each platform's native query language has its own format and requirements, but Sigma serves as the unifying layer on top. The Sigma rule itself is concise and human-readable, focusing on *what* to detect (e.g., a process name and keyword) without tying it to a specific database or index. The heavy lifting of adapting that logic to each system's quirks is handled by Sigma's converters or backends. In practice, a security engineer could take the Sigma rule above and use an automated tool (like the Sigma CLI or a service like Uncoder.io) to obtain the Splunk SPL, Azure KQL, or QRadar AQL equivalent on the fly. This saves enormous effort and ensures consistency – all these queries will be hunting for the exact same behavior, just expressed in different dialects.

It's worth noting that platform-native languages often offer specialized capabilities (for example, Splunk's statistical functions or Azure's machine learning queries) that go beyond Sigma's scope. Sigma intentionally sticks to the common denominator of log search features for compatibility. If a detection use-case truly requires a SIEM-specific feature, one might still need to write a custom query for that platform. However, the vast majority of day-to-day threat detection needs (string matching, Boolean logic, counting events, basic correlations) can be captured in Sigma's universal format. By covering those, Sigma greatly reduces the need to write separate rules per platform.

In summary, **Sigma vs. native DSLs** is not a competition so much as a translation exercise: Sigma provides a *vendor-agnostic blueprint*, while SPL, KQL, AQL, etc., are the *implementations* of that blueprint on specific SIEM engines. Sigma rules abstract away environmental specifics like index names, database table names, and exact field identifiers. The result is that detection engineering becomes more about defining *what* you want to catch (the tactics, techniques, and patterns of attacks) rather than *how* each SIEM needs to be told to catch it. This abstraction

makes junior analysts more productive – they can write a Sigma rule without deep knowledge of every SIEM query language, and rely on Sigma's tooling to handle the rest. Meanwhile, senior analysts appreciate the agility: they can rapidly disseminate new detections across multiple platforms. Organizations running multiple logging tools especially benefit, as Sigma unifies their detection logic across, say, an on-prem SIEM and a cloud logging service.

To conclude this section, imagine you're tasked with improving detections in a heterogeneous environment with Splunk and Azure Sentinel. Instead of maintaining two separate sets of rules (one SPL, one KQL) and trying to keep them in sync, you could maintain **one set of Sigma rules**. Each Sigma rule can be automatically converted to SPL or KQL as needed, ensuring parity between platforms. This not only cuts down duplication, but also reduces the chance of error (you won't accidentally update a rule in Splunk and forget to update the Sentinel version, for example). The Sigma approach enables a *single source of truth* for detection logic, which is a big win for consistency and maintainability in security operations. In the ever-evolving cat-and-mouse game of cybersecurity, Sigma gives defenders a way to rapidly share and deploy detections without being bogged down by the idiosyncrasies of each tool's query language. It truly embodies the ideal of a universal detection-rule language – one that lets us focus on catching bad actors, not on rewriting queries.

2. Sigma Rule Fundamentals

2.1 Anatomy of a Rule (YAML)

Sigma rules are written in YAML and consist of **three main parts**: a **header** section with descriptive metadata, a **core** section defining what to detect and where, and additional **meta** fields providing context and classification. Understanding this anatomy is key for writing effective cross-platform detection rules. Below is an example Sigma rule (for an Okta user account lockout) to illustrate these components:

```
title: Okta User Account Locked Out
id: 14701da0-4b0f-4ee6-9c95-2ffb4e73bb9a
status: test
description: Detects when a user account is locked out.
references:
  - https://developer.okta.com/docs/reference/api/system-log/
  - https://developer.okta.com/docs/reference/api/event-types/
author: Austin Songer @austinsonger
date: 2021-09-12
modified: 2022-10-09
tags:
  - attack.impact
logsource:
  product: okta
  service: okta
detection:
  selection:
    displaymessage: Max sign in attempts exceeded
  condition: selection
falsepositives:
  - Unknown
level: medium
```

As shown above, the **header** provides basic identification and purpose of the rule. The **core** defines the log source and detection logic, and the **meta** fields give context like false positives, severity, and tags.

- **Header Fields:** The header includes fields like title, id, description, and references. The **Title** is a short, human-readable name summarizing what the rule detects. The **ID** is a unique identifier (typically a UUIDv4) for the rule, ensuring global uniqueness. The **Description** explains the rule's intent – what suspicious behavior it looks for, and sometimes why that behavior is significant. **References** provide URLs or citations to external sources that justify or inform the rule (for example, a blog post describing the threat, a CVE, or a threat intel report). These fields help anyone reviewing the rule quickly understand its purpose and origin. In the example above, the title and description make clear that the rule detects account lockouts, and the references point to Okta documentation for such events.
- **Core Detection Fields:** This is the heart of the Sigma rule. It consists of two parts: the **logsource** and the **detection** (which includes a **condition**). The **logsource** field tells Sigma *where* to look – defining the type of logs the rule applies to. It uses attributes like

product, service, and/or category to specify a source (e.g. Windows Event Log, Sysmon, Okta logs, Linux audit logs, etc.). For instance, logsource: product: windows, category: process_creation would target Windows process creation events. This scoping is crucial so that the SIEM or log platform knows which dataset to query and can prepend the appropriate index or source filter. In our Okta example, logsource is defined with product: okta and service: okta, meaning it will apply to Okta system log events.

The **detection** field defines *what* to look for in those logs. Sigma organizes detection logic into one or more named **selections** (essentially patterns or filters), and a **condition** that combines them logically. Each selection is a set of field-value criteria (or keyword searches) that may indicate the threat activity. The simplest case is a single selection; for example, in the Okta rule above, the selection looks for displaymessage: "Max sign in attempts exceeded", which is a specific log field indicating a locked-out account. The **condition** then specifies how to evaluate the selection(s) – in this case condition: selection means the rule triggers if the single selection is met. Conditions support logical operators (and, or, not) and even wildcards (like 1 of selection*) to combine multiple sub-detections. This flexible logic allows Sigma to represent complex queries. For example, a rule might define multiple patterns (selection_1, selection_2, ...) and use condition: selection_1 or selection_2 to fire on any of them, or selection_1 and not selection_2 to include one pattern while filtering out another. (We will see a practical example of multiple selections in the next section.) The detection section is the most important part of the rule – it precisely codifies the suspicious behavior we're hunting for.

- **Meta Fields:** Surrounding the core, Sigma rules include additional metadata to guide usage and triage. Key meta fields are **falsepositives**, **level**, and **tags** (among others like author, date, status which were already touched on). The **falsepositives** field lists known benign events or situations that might trigger the rule by accident. This is essentially advice to analysts: "if this alert fires, check if it could be one of these expected cases." For example, a rule detecting msra.exe spawning system tools might note a legitimate admin use of Remote Assistance as a false positive. Note that falsepositives entries are **not** used by the Sigma engine to suppress alerts; they are purely informational and for analyst awareness (actual filtering of events should be done via the detection logic or Sigma's condition/filter constructs if you want to exclude them programmatically).

Level indicates the severity or priority of the alert if the rule fires. Sigma defines standard levels: informational, low, medium, high, critical. A **critical** or **high** level rule implies the detected activity is likely an actual incident or a very important threat that needs immediate attention. Lower levels indicate suspicions or policy violations that might need review but are not necessarily malicious (or could even just be for compliance monitoring). In our example, the Okta lockout rule is marked level: medium, since an account lockout might indicate a brute-force attempt (suspicious, but not as dire as confirmed malware). Level helps SIEMs and analysts prioritize alerts.

Tags are a flexible list of labels that categorize the rule according to common frameworks, threat tactics, or other groupings. Tags can map the detection to the MITRE ATT&CK framework, reference specific adversary techniques, tools, or even mark compliance requirements. For example, tags starting with attack. denote MITRE ATT&CK tactics or techniques. A rule detecting process injection might have tags like attack.defense-evasion and attack.t1055 (for ATT&CK tactic "Defense Evasion" and technique T1055 Process Injection). These tags don't affect

detection logic, but they are incredibly useful for enrichment and integration: they enable automatic mapping of the rule to known tactics/techniques and can be used to filter or organize rules. (We will discuss tagging and ATT&CK mapping in depth in section 2.3.) Other common tags include references to **MITRE's CAR** (Cyber Analytics Repository, e.g. car.2013-05-009), **CVE IDs** (e.g. cve.2023-12345), or **TLP designations** (e.g. tlp.red). In summary, the meta fields like falsepositives, level, and tags enrich the rule with context that helps in downstream SIEM integration and analyst decision-making.

By combining a clear header, a targeted logsource, precise detection logic, and supportive metadata, a Sigma rule provides a *portable description of a threat detection*. Next, we'll walk through how to build a Sigma rule from scratch, applying this anatomy to a real-world threat scenario.

2.2 Building Your First Rule – Step-by-Step

Writing your first Sigma rule may seem daunting, but following a structured process will make it easier. In this section, we'll go step-by-step through creating a Sigma rule, from concept to final YAML, using a real-world threat scenario. Let's choose a concrete example: **detecting a credential dumping attempt on a Windows host**. Credential dumping (stealing password hashes or secrets from memory) is a common technique attackers use post-compromise (for instance, using tools like Mimikatz or dumping the LSASS process). We will build a rule to catch an attacker dumping LSASS memory using Sysinternals ProcDump – a technique mapped to MITRE ATT&CK technique *T1003.001 (LSASS Memory)* under the Credential Access tactic.

Step 1: Define the Detection Goal (Threat Scenario)

Every Sigma rule starts with a clear idea of *what malicious activity you want to detect*. In our scenario, the goal is to detect a suspicious process memory dump of LSASS (the Local Security Authority Subsystem Service, which holds credentials in memory). Normally, only certain tools or administrators would dump LSASS memory (for debugging or crash analysis). An attacker, however, might use ProcDump (a legitimate tool) or a similar program to dump LSASS and extract credentials. We consider this behavior malicious unless done by authorized personnel. So our use case is: *"Detect any process creating a memory dump of LSASS."* This is often manifested by a command-line execution of **procdump** (or a renamed version of it) targeting lsass.exe. Formulating it in plain language: *If any process is launched with parameters that indicate dumping lsass.exe memory, flag it.* Having this defined objective will guide all subsequent steps.

Step 2: Identify Relevant Log Sources

Next, determine where this behavior would appear in logs. In a Windows environment, process creation and command-line information are typically logged by tools like **Sysmon** (System Monitor) or in the Security Event Log (4688 events, if CommandLine logging is enabled). Sysmon is a common choice for detailed process monitoring. For our rule, we assume the organization is collecting Sysmon process creation logs. In Sigma's terms, we need the appropriate logsource that captures processes and their command lines on Windows. Sigma uses abstract logsource definitions; looking at Sigma's standard taxonomy, "process_creation" is the category for process execution events. So we set:

- product: windows (the platform)

- category: process_creation (generic category for process start events, covers Sysmon process create events and similar)

This ensures our rule will apply to any Windows process creation log source in the Sigma ecosystem. (If we knew it was specifically Sysmon, we could also specify service: sysmon, but just category: process_creation is broad enough and commonly used in many Sigma rules for Windows process start detections.) By scoping to Windows process creation events, we focus the SIEM query on the relevant data and avoid noise from other log types.

Step 3: Craft the Detection Logic

With the scenario and log source in mind, we now decide on the specific *indicators or patterns* to detect the LSASS dump. We know from our scenario that ProcDump (and similar tools) use a particular command-line syntax to dump a process: typically the **-ma** flag (meaning "write a full memory dump") and the name or process ID of the target (in this case "lsass"). For example, the command might look like: procdump.exe -ma lsass.exe C:\Temp\lsass.dmp. Attackers might rename the procdump.exe binary to evade static detection, but they would still likely use the -ma switch and reference lsass. Thus, two key pieces in the command line are the **"-ma" flag** and the string **"lsass"**. Our detection logic can leverage these.

In Sigma YAML, we will express this as two conditions that must *both* be present in a single event's data (logical AND): one indicating the presence of "-ma" and one indicating "lsass". We can implement this by defining two selection patterns and then combining them:

- selection_dumpFlag: match if the process CommandLine contains "-ma" (with appropriate spacing to avoid matching unrelated "ma" substrings – we might use a space or dash context).
- selection_lsass: match if the CommandLine contains "lsass" (case-insensitive by default, since Sigma patterns are generally case-insensitive unless specified, and this should catch "lsass.exe" or even just "lsass").

Using Sigma's syntax, we might write something like:

```
detection:
  selection_dumpFlag:
    CommandLine|contains: " -ma "
  selection_lsass:
    CommandLine|contains: "lsass"
  condition: selection_dumpFlag and selection_lsass
```

This means the event's CommandLine field must have " -ma " *and* have "lsass" somewhere. The condition and ensures both criteria are satisfied in the same log event. (We choose " contains" operator here for simplicity; Sigma also offers more advanced matching operators like wildcard or regex if needed. We included spaces around -ma to avoid matching unrelated strings like "ProgramArgs=..." etc., and because in command-line syntax -ma is typically delineated by spaces.)

If we anticipated that attackers might use a *different flag* or method to dump memory, we could broaden or add patterns. For instance, some tools might use rundll32.exe C:\Windows\System32\comsvcs.dll, MiniDump to dump LSASS. That would show up differently (with "comsvcs.dll" and "MiniDump" in command line). We could add selections for those indicators too. But to keep our first rule manageable, we'll focus on the ProcDump technique.

At this stage, it's good practice to consider if our detection might catch legitimate admin activity. Dumping LSASS is rare in normal operations, but a system administrator or security tool *might* do it for troubleshooting or forensic memory analysis. We should note that as a potential false positive (and indeed, our meta fields will capture that). However, because of the sensitivity of credential data, we likely want to alert on it every time and have an analyst verify if it was legitimate.

Step 4: Fill in the Rule Metadata

Now that the core logic is defined, we complete the Sigma rule by providing the metadata (header and meta fields):

- **Title:** Choose a concise title that conveys the detection. For example: **"Potential LSASS Process Dump via ProcDump"**. This clearly labels what the rule is about (dumping LSASS memory) and even hints at the method (using ProcDump). We avoid phrasing like "Detects X" in the title (since it's redundant); instead, we state the scenario as a noun phrase.
- **ID:** Generate a new UUID for this rule (e.g., 5afee48e-67dd-4e03-a783-f74259dcf998). Each Sigma rule needs a unique id. This is important for tracking the rule in different systems (and if we or others update it later).
- **Description:** Write a sentence or two describing what the rule detects and maybe why. For example: *"Detects suspicious use of Sysinternals ProcDump (or renamed variants) to create a full memory dump of the LSASS process, a technique used to harvest credentials from memory."* This answers *what* and *why* – it gives context that this behavior is tied to credential dumping (and thus likely malicious). A good description helps others quickly grasp the rule's intent without parsing the logic in detail.
- **Status:** We can mark this rule's maturity. Since this is our first attempt, we might set status: test or experimental, indicating it's new and might need tuning. Once it's verified to work well with minimal noise, it could be promoted to stable.
- **Author, Date:** Include your name (or handle) as the author, and set the date to today's date. (If the rule is later modified, the modified date can be added).
- **References:** It's good to cite sources or inspirations for the detection. We could reference a Microsoft article on ProcDump, a MITRE ATT&CK technique page, or a blog discussing LSASS dumping. For example:
    - https://learn.microsoft.com/en-us/sysinternals/downloads/procdump (documentation of ProcDump)
    - A blog post about LSASS dumping or a threat report (if available).
      These references show where you got information about the technique or confirm why the activity is risky. In community Sigma rules, references might include threat intel reports or tweets where the behavior was first noted.
- **False Positives:** We know dumping LSASS is rarely normal. We can state false positives as *"Unlikely; an administrator performing a manual memory dump could trigger this."* or *"Legitimate use of ProcDump on LSASS for troubleshooting."* In the example Sigma repository rule for this scenario, they noted it's unlikely to be benign. We list it anyway so analysts know what benign scenario could explain an alert.

- **Level:** Given the sensitivity, we set level: high. A compromise of credentials is high priority, and we expect almost no benign triggers (maybe only high-privilege admin actions). A high level means the SOC should treat alerts from this rule as important and investigate promptly.
- **Tags:** Finally, we add tags to map this detection to frameworks. For MITRE ATT&CK, this clearly falls under *Credential Access* tactic and the sub-technique for LSASS dumping. We add:
    - attack.credential-access – indicating the ATT&CK tactic (TA0006).
    - attack.t1003.001 – the specific technique ID for **OS Credential Dumping: LSASS Memory**.
      We might also include attack.defense-evasion and attack.t1036 (masquerading) if we consider that an attacker renaming ProcDump is an evasion tactic. In fact, the official community rule did tag T1036 as well, since renaming a known tool is a form of **masquerading** to evade detection. This illustrates that a single Sigma rule can map to multiple ATT&CK techniques if applicable. We will ensure at least the primary technique (T1003.001) is tagged for automatic ATT&CK mapping in our SIEM. We could also add a reference to MITRE's CAR if one exists for this analytic (in the example above, they included car.2013-05-009 which corresponds to a CAR detection for credential dumping). Tags like these don't affect the rule's logic but greatly aid in integration and threat hunting.

Putting it all together, here's what our rule looks like in full:

```
title: Potential LSASS Process Dump via ProcDump
id: 5afee48e-67dd-4e03-a783-f74259dcf998
status: test
description: |
  Detects suspicious use of the Sysinternals ProcDump utility (or
renamed versions) to dump the memory of lsass.exe.
  Attackers use this technique to obtain credentials from LSASS
memory (MITRE ATT&CK T1003.001).
references:
  - https://learn.microsoft.com/en-
us/sysinternals/downloads/procdump
  - https://attack.mitre.org/techniques/T1003/001/
author: Your Name (YourOrg)
date: 2025-06-23
tags:
  - attack.credential-access
  - attack.t1003.001
  - attack.defense-evasion
  - attack.t1036
logsource:
  product: windows
  category: process_creation
detection:
  selection_dumpFlag:
    CommandLine|contains: " -ma "
  selection_lsass:
```

```
    CommandLine|contains: "lsass"
  condition: selection_dumpFlag and selection_lsass
falsepositives:
  - Administrator troubleshooting (manual LSASS dump)
level: high
```

Let's briefly verify this rule against our design: The logsource restricts to Windows process events. The detection looks for the two indicators in the CommandLine. The condition uses a logical AND, meaning both must be present in the same event (which is what we want for a ProcDump command line). We've provided a helpful description and references (including the ATT&CK technique page for context). The tags include the relevant MITRE ATT&CK tactic and technique, so any platform that ingests this rule can automatically know it relates to credential dumping. False positives are noted as only very rare admin activity. And we mark it high severity, because if this fires, it likely indicates a serious incident (unless proved otherwise).

Step 5: Test and Integrate the Rule

With the rule written, the final step is testing it and integrating it into your SIEM or detection system. In a real setting, you would test the rule on log data to ensure it catches the bad behavior and doesn't overwhelm you with false positives. For example, if you have a lab, you might run ProcDump against LSASS on a test machine and see if the Sigma rule triggers (or use Sigma's converter to generate a SIEM query and run that query on your log data).

Using the Sigma CLI tool (which leverages the pySigma library), we can **convert** this Sigma rule to the query language of our SIEM. For instance, to get a Splunk query, one would run:

```
sigma convert -t splunk -p sysmon <path_to_rule.yml>
```

This would output something akin to a Splunk search query:

```
(CommandLine="* -ma *" AND CommandLine="*lsass*")
```

In Splunk Query Language, which matches our intended logic (the CLI automatically adds wildcards and formatting as needed). We could similarly convert to Elasticsearch DSL, QRadar AQL, or any supported backend. Testing the query on historical logs will show if the rule triggers appropriately. We might discover, for example, that our simple "lsass" substring also catches events where someone types a command like lsass (not likely) or that some benign monitoring tool also uses "-ma" for something. If so, we'd iterate – maybe refine the detection (e.g., require lsass.exe specifically, or add an exclusion pattern). This iterative tuning is a normal part of rule development.

Once satisfied, we can deploy the rule. Many SIEMs let you import Sigma rules or you can store the Sigma in your content management and use a pipeline (Sigma CLI or others) to continuously translate them to SIEM queries. In integration, remember to map fields as needed (Sigma uses abstract field names that the backend config must map to actual log fields – for instance, Sigma's CommandLine might map to ProcessCommandLine in one SIEM or a different field name in another, configured via Sigma backend). Our example being straightforward, most backends have mappings for CommandLine in Windows process events.

Congratulations – we've built a Sigma rule from scratch! The process involved understanding the threat, selecting the right logs, writing the detection logic, and adding informative metadata. This same process can be applied to countless scenarios, from detecting process injection to spotting persistent registry changes.

For example, if we wanted to detect **process injection (ATT&CK T1055)** as a next use case, we would follow similar steps: define how process injection appears in logs (perhaps a legitimate process spawning a known suspicious child process, or a Windows event indicating memory injection), choose the logsource (maybe Sysmon with process creation and access events), and then write the detection (e.g., parent process = msra.exe and child process = cmd.exe or arp.exe, as in a known Qakbot injection scenario). The metadata would tag it with attack.defense-evasion and attack.t1055 and so on. In fact, the Sigma community rule "Potential Process Injection via Msra.exe" does exactly that – it looks for msra.exe spawning unusual utilities, tags T1055, and marks level high. You can build rules for other real-world threats in the same way. The key is breaking down the problem: know what you're detecting, find it in the logs, express it in Sigma's structure, then test and refine.

By now, you've seen how to compose Sigma rules and create your own detections. The final piece we'll cover is how Sigma rules integrate with frameworks like **MITRE ATT&CK** through tagging, and how this enables automation and richer analysis in a SIEM.

2.3 Tagging & Automatic Mapping to MITRE ATT&CK

One of Sigma's powerful features is its use of **tags** to integrate with external frameworks and enrich detections – foremost among these is the **MITRE ATT&CK** framework. MITRE ATT&CK is a globally-recognized matrix of adversary tactics and techniques. By tagging Sigma rules with ATT&CK tactics/techniques, rule authors create a direct mapping between a detection and the standard terminology for that behavior. This has immense practical benefits: it helps SOC teams see which part of the attack chain an alert corresponds to, facilitates reporting on detection coverage, and allows SIEMs to automatically link alerts to ATT&CK information.

How Sigma Tags Represent ATT&CK: In Sigma rule YAML, ATT&CK mappings are typically added as tags in the format attack.TXXXX for techniques (where *TXXXX* is the technique ID) and attack.{tactic} for tactics (using the ATT&CK tactic name in lowercase). For example, our credential dumping rule has attack.credential-access (tactic) and attack.t1003.001 (sub-technique for LSASS dump). A rule for process injection might include attack.defense-evasion and attack.t1055. These tags correspond to entries in the MITRE ATT&CK knowledge base. Sigma doesn't magically know all the details of those techniques – it simply provides a consistent way for the rule to declare "this detection is related to ATT&CK technique T1055 (Process Injection) under Defense Evasion." The Sigma project maintains a **tags appendix** with naming conventions so that everyone uses the same tag format (for instance, ensuring you use attack.persistence and not attack.persistence_execution or some non-standard term). The community best practice is to tag each rule with *at least* the primary ATT&CK technique it addresses, and often the tactic as well for clarity. Indeed, many public Sigma rules have multiple tags covering primary and secondary techniques or related tactics.

Automatic Mapping in Practice: When you integrate Sigma rules into a SIEM or other security platform, those ATT&CK tags can be leveraged for automation. Modern SIEMs often have built-in support for ATT&CK. For example, a SIEM might allow you to mark a correlation rule with related ATT&CK techniques, and then it can generate a heatmap or show technique details when an alert fires. By importing Sigma rules with tags, this mapping can happen automatically. The LogPoint SIEM, as one case, explicitly maps Sigma tags to its ATT&CK fields: if a Sigma rule's tags include ATT&CK technique IDs, LogPoint's Sigma integration will populate the alert's "ATT&CK techniques" metadata accordingly. This means when our LSASS dump rule triggers in LogPoint, the alert would be annotated with *Credential Access (T1003.001)* in the interface,

without the engineer manually entering that. Similarly, other platforms like Splunk Enterprise Security or Microsoft Sentinel allow mapping rules to ATT&CK; if you convert a Sigma rule to those platforms, you can carry over the tags. In Splunk ES, for instance, one could script the import of Sigma rules such that the attack.t1003.001 tag populates the rule's MITRE technique field automatically (since Splunk ES supports technique ID fields). This creates a seamless link between detection content and the ATT&CK framework.

From an analyst's perspective, this is extremely useful. Imagine an alert comes in titled "Potential LSASS Process Dump via ProcDump" – if it's tagged with T1003.001, the SOC analyst (or the SIEM itself) can immediately display: *Technique: OS Credential Dumping: LSASS Memory (T1003.001), Tactic: Credential Access*. The analyst can click on that and see MITRE's description of the technique, learn about how it's used by attackers, and see recommended mitigations or detection notes from MITRE. It adds context that enriches the raw alert data. As a result, triage and investigation become faster and more informed.

Significance of Tags: Beyond just MITRE, tags in Sigma serve as a form of metadata that can drive automation. Security teams often track their detection coverage by ATT&CK tactic/technique. By tagging all Sigma rules, you can generate a coverage matrix easily: e.g., list all rules covering Defense Evasion or find if you have any rules for technique T1218 (Signed Binary Proxy Execution). Tools exist to facilitate this; for instance, SigmaHQ's own tools or community scripts can parse a folder of Sigma rules and tally techniques. Another example is filtering – you could run the Sigma CLI to *list rules by tag*. If an intel report says "watch out for attackers using technique X," you could filter your Sigma rules for that attack.TX tag to quickly find relevant detections (or identify gaps if none exist). The **pySigma** library enables such programmatic use: one could load Sigma rules as Python objects and cross-reference their tags with an ATT&CK dictionary, or even automatically update tags if MITRE IDs change (for example, when sub-techniques were introduced, many rules updated tags from T1003 to T1003.001).

There are also initiatives to **auto-tag** Sigma rules. For example, SOC Prime's Uncoder AI has been mentioned to predict MITRE ATT&CK tags for Sigma based on rule logic. This underscores the importance of tagging: everyone wants their detection rules aligned with MITRE because it's the lingua franca of adversary behavior. However, an important point is that tagging in Sigma is a manual, human-driven process in most cases – the rule author decides the tags. So one must ensure the tags are accurate; over-tagging or mis-tagging can reduce usefulness (imagine marking a rule with 5 techniques that aren't actually all covered – it could mislead analysts). Best practice is to tag only what truly applies, and treat tags as *"what technique does this rule address or detect?"*.

Using Sigma CLI and Tools for Tags: The Sigma CLI doesn't automatically fetch ATT&CK names, but it preserves the tags during conversion. If you convert a rule to a Splunk savedsearches configuration, the tags might be included in description or name fields, depending on the backend. Some backends (like Azure Sentinel) might allow injecting the technique IDs. For custom pipelines, one could use pySigma to read the Sigma YAML and then call the ATT&CK API or a local ATT&CK dataset to get full technique details, thereby creating enriched content. For instance, an internal tool could parse attack.t1003.001 and replace or augment it with "OS Credential Dumping: LSASS Memory" before inserting into a knowledge base. This kind of enrichment can be done because the tag format is predictable.

In summary, **tags bridge the Sigma rule to a larger context**. They map our universal detection rule language to the universal language of adversary tactics. By including tags in our rules, we

empower SIEM integrations to automatically classify and display alerts in an ATT&CK-aligned way. We also make our content more shareable – any organization that imports community Sigma rules can immediately sort or filter them by ATT&CK technique to build a defense coverage map.

To concretely illustrate, recall our Sigma rule example for LSASS dumping. We tagged it with attack.t1003.001. In a MITRE ATT&CK spreadsheet or dashboard, this falls under *Credential Access > OS Credential Dumping (T1003) > LSASS Memory (Sub-technique T1003.001)*. If our SOC has a dashboard of ATT&CK techniques with detection coverage, once this rule is deployed, the cell for T1003.001 would light up, indicating we have a detection in place for that technique. If an alert fires, the incident management system might automatically attach "Technique: T1003.001" to the incident record. Analysts could then, for instance, pivot to see all past alerts tagged with T1003 (maybe revealing a pattern of credential dumping attempts over time). This synergy is only possible because we included the proper tags in the rule. It's a small step during rule creation that pays off immensely in operations.

Lastly, tags are not limited to ATT&CK. Sigma rules often include tags for relevant **malware or threat actors** (e.g. a rule related to *Remcos RAT* might have tool.remcos or actor.fin7 if such conventions are used), or internal tagging schemes (some teams add policy.x or use-case.y tags). However, MITRE ATT&CK has become the most standardized tagging approach. The openness of Sigma's tag field allows organizations to include whatever labels they need alongside the standard ones. The **SigmaHQ conventions** repository provides guidance on naming so that community contributions remain consistent.

In conclusion, tagging in Sigma serves as the connective tissue between a raw log detection and a broader cybersecurity framework. Automatic mapping to MITRE ATT&CK via these tags helps junior and mid-level analysts understand the significance of an alert (e.g., "This is part of Credential Access – they might be trying to move laterally or elevate privileges next") and helps engineering teams ensure they have detections across the spectrum of tactics. Tools like Sigma CLI and pySigma facilitate using these tags in practice, whether converting rules to SIEM-specific formats or programmatically analyzing your rule set. As you develop Sigma content, always take a moment to add and verify tags – it will make your rules far more valuable in an integrated defense program.

With the anatomy of Sigma rules understood, the process of creating them demystified, and the power of tagging explained, we have a solid foundation. Sigma truly acts as a universal detection rule language – from the basics of YAML structure to integrating with SIEMs and aligning with ATT&CK, it enables security professionals to write once and deploy detections everywhere. The next parts of this book will delve into how Sigma rules are operationalized in SOC workflows and how to contribute to and leverage the community rule sets, but with what we've covered in sections 2.1 through 2.3, you are well on your way to being a Sigma rule author and adept detection engineer. Happy hunting!

3. Tooling & Workflow

3.1 Setting Up Your Workspace (Git, VS Code, linters)

Before writing Sigma rules, it's crucial to prepare a proper workspace with version control, a suitable editor, and linting tools. This ensures your detection rules are organized, syntactically correct, and easy to manage.

Git Version Control: Start by using Git to manage your Sigma rules. If you plan to leverage existing community rules or contribute back, clone the official SigmaHQ rules repository from GitHub. For example:

```
# Clone the main Sigma rules repository (read-only)
git clone https://github.com/SigmaHQ/sigma.git

# OR: Fork the repository on GitHub and clone your fork for
contributions
git clone https://github.com/<yourusername>/sigma.git
```

Cloning the repo gives you access to thousands of existing rules (over 3,000 as of this writing) in the rules/ folder. It's a great way to explore how rules are written and to use them as templates. If you're writing custom rules for your organization, you might instead initialize a new Git repository (often called a "detection-as-code" repo) to track changes to your own Sigma rules. Git enables collaboration and version history for your detections, so you can see who added what and roll back if needed. It also makes it easier to **integrate Sigma rules into CI/CD pipelines** or deploy them to SIEM platforms (though CI/CD specifics are beyond our scope here).

After cloning or creating a repo, consider setting up a branching strategy for rule development. For example, you might create a branch for each new rule or use feature branches for different projects, then merge into a main branch after review. If contributing to the community SigmaHQ repo, create a fork and work on a feature branch, then open a pull request as described in SigmaHQ's Contributing guide.

Visual Studio Code (VS Code) Setup: A good editor will make writing YAML-based Sigma rules much easier. VS Code is a popular choice for Sigma development – it's free, cross-platform, and has excellent YAML support. Ensure you have the YAML language extension (VS Code often includes the Red Hat YAML extension by default, which provides syntax highlighting, schema validation, and formatting). With YAML support, VS Code will alert you to basic syntax errors (like misaligned indents or improper list formatting) as you edit.

For an even better experience, install the **Sigma VSCode extension** (available on the Visual Studio Marketplace as "Sigma" by *humpalum*). This extension is tailor-made for Sigma rules. It offers snippet templates and live rule validation. For example, you can type newrule and hit Enter to expand a complete Sigma rule template with all the necessary fields, or type logsource to get a snippet for the log source section. The extension also performs on-the-fly **diagnostics** of your rule to ensure it conforms to Sigma's schema and conventions (e.g. it checks if your title is too long, if you accidentally put a contains modifier in the wrong place, or if there are trailing spaces). These "sanity checks" help catch common mistakes early. The extension even highlights formatting issues like an extra space at end of line or missing indent, and can auto-continue YAML lists as you press Enter for a smoother editing workflow. Setting your author

name in the extension settings (sigma.author) will automatically populate the author: field in new rule templates, saving time.

In VS Code, it's also helpful to enable **editorconfig** or workspace settings to enforce YAML indent style. Sigma uses 2 spaces or 4 spaces? (SigmaHQ uses 2 spaces in some documentation examples, but the official repo uses 4-space indentation consistently). In fact, the SigmaHQ repository includes a configuration for **YAML lint** (.yamllint file) that enforces 4 spaces per indent, no tabs, and other style points. You can mirror these settings in your editor to automatically trim trailing whitespace and add a newline at end of file on save. Consistent formatting is not just style – in YAML, a stray tab or wrong indent can break the rule.

Linters and Pre-commit Hooks: Using a linter will catch format issues or simple errors in your Sigma rules before you push them. A recommended tool is **yamllint**, which checks YAML files against style and syntax rules. The Sigma project's yamllint config (located in the repo's root) requires, for example, that comments have a starting space, no more than 2 consecutive blank lines, and no trailing spaces. You can install yamllint (pip install yamllint) and run it on your rule files to ensure compliance:

```
yamllint -c sigma/.yamllint ./rules/my_new_rule.yml
```

It's wise to integrate such checks into your workflow. If you use Git, set up a **pre-commit hook** to run yamllint (and potentially other tests) on every commit. This way, if a rule file violates the formatting guidelines or has basic YAML errors, the commit will be blocked until you fix it – preventing broken or sloppy rules from slipping into your repository history. You can use the **pre-commit** framework to manage this easily. For example, create a .pre-commit-config.yaml in your repo that includes yamllint, then run pre-commit install. This will automatically lint your Sigma rules on commit. Another useful check is simply trying to **parse and convert the rule** (more on conversion in section 3.3) to catch schema errors – for instance, if you forgot a required field like detection or made a typo in field names, the converter will throw an error. You might create a lightweight script or use sigma-cli in a no-output mode to validate rules as part of pre-commit. In short, automating quality checks ensures your rules remain consistent and valid as you build your detection library.

Finally, keep your workspace organized. You might structure your own rule repository similar to SigmaHQ's (more on that below), by grouping rules into folders by platform or category. This makes navigation in VS Code easier – you could open the whole repository folder in VS Code and use its search to find if a similar rule already exists (a common practice is to search the SigmaHQ repo for a technique or event ID to avoid reinventing the wheel). By setting up Git, VS Code, and linters at the outset, you create a reliable environment to develop Sigma rules efficiently and correctly.

3.2 The SigmaHQ Repository Structure

The official SigmaHQ repository (sigma on GitHub) is the central hub for community-contributed Sigma rules. Understanding its structure will help you navigate existing detections and organize new ones. In the past, **all Sigma rules lived under a single** rules/ **directory**, but as the project grew, this became unwieldy. In 2023, the maintainers restructured the repository to categorize rules by their purpose and use-case. Now the repository is organized into multiple top-level rule folders:

- **rules/** – **Generic Detection Rules.** This is the primary collection of Sigma rules, focusing on general, threat-agnostic detections. These rules aim to catch behaviors or tactics that adversaries might use, regardless of who the actor is. In other words, generic rules look for techniques (e.g. a process injection, a suspicious registry change) that could appear in many attacks. They are broadly applicable and form the backbone of detection content. (In SigmaHQ's terms, these are "generic detection rules" that detect a behavior or technique that *was, can, or will be used by a threat actor*.)

- **rules-threat-hunting/** – **Threat Hunting Rules.** Rules in this folder are geared towards hypothesis-driven hunting and tend to be broader or more forgiving of noise. When analysts are in hunting mode, they often prefer to cast a wider net (accepting a higher false positive rate) to discover anomalies. Sigma threat-hunting rules reflect this by capturing potentially suspicious patterns that aren't yet confirmed threats. They are separated from the main rules so that users know these may require more analysis or tuning. By isolating hunting rules, the Sigma project sets the right expectations – these rules give starting points for investigations rather than immediate "alert actionable" signals. *(At the time of writing, the migration of older generic rules that really serve hunting purposes into this folder is ongoing.)*

- **rules-emerging-threats/** – **Emerging Threat Rules.** This folder contains rules for *timely, specific threats* – things like newly disclosed zero-day exploits, active malware campaigns, or recent APT tactics. These rules tend to be more specific indicators tied to current or recent events. Their relevance can be short-lived (a year from now, that malware might disappear), but during an outbreak or high-profile incident, they are critical for coverage. Sigma's vendor-agnostic format really shines here: as soon as a new threat emerges, researchers (like those at Nextron or the Sigma community) can contribute a Sigma rule for it, and defenders everywhere can immediately deploy it to *their* SIEM, whatever it may be. The emerging-threats rules are often grouped by threat; for example, if a new malware called **"COLDSTEEL RAT"** is discovered, all Sigma rules related to that malware might be put under a common subfolder or share a common prefix, along with context links in the rule metadata. This makes it easier to share a set of related detection rules during an outbreak.

- **rules-dfir/** – *(Planned)* **DFIR (Digital Forensics and Incident Response) Rules.** The repository maintainers have reserved this folder for DFIR-specific rules. These might include rules that assist in post-incident investigations or triaging (for example, detecting traces of known attacker tools or behaviors that are primarily of interest during incident response, not for continuous monitoring). At present, rules-dfir is marked *"TBD"* (to be developed) – indicating it's a placeholder for future content as the Sigma project expands.

- **rules-compliance/** – *(Planned)* **Compliance Rules.** Similarly, this placeholder folder is meant for rules that help detect compliance-related events (for instance, unauthorized changes in a system that would violate PCI or GDPR requirements, or detecting when auditing is turned off on a critical system). As of now, it's also *TBD*. Once populated, organizations could leverage these rules to meet security control monitoring for various standards.

Under each of these top-level categories, the rules are further organized by technology, platform, or log source domain. For example, within the generic rules/ folder, you will find

subdirectories like windows/, linux/, cloud/, network/, etc., grouping rules by the environment or product they pertain to. Drilling down further, each platform folder may have categories or service-specific groupings. The structure tends to follow the **logsource** values inside the rules. For instance, Windows rules are grouped by categories like windows/process_creation/, windows/powershell/, windows/sysmon/, etc., aligning with Windows Event Log channels or data sources (Sysmon, PowerShell logs, etc.). In the Sigma rule metadata, the logsource field might be product: windows and service: sysmon – correspondingly, that rule would reside in the rules/windows/sysmon/ directory for easy discovery. Likewise, rules for cloud services live under rules/cloud/ (the example rule **"Okta User Account Locked Out"** resides at rules/cloud/okta/okta_user_account_locked_out.yml). This hierarchy helps practitioners quickly find relevant detections (e.g. all rules for AWS CloudTrail would be under rules/cloud/aws/).

Beyond the rule folders, the SigmaHQ repository contains a few other notable directories and files:

- **tests/** – Contains test cases and log samples for certain rules. The Sigma maintainers use these to validate that rules trigger appropriately. If you contribute a new rule, you might also add a test log event here to demonstrate the rule in action. Running the test suite helps catch logical errors (for example, a rule that is too broad or doesn't match what it's supposed to).

- **unsupported/** – Houses Sigma rules or content that are not officially supported. Often this includes older rules that might rely on deprecated fields or experimental ideas that haven't been fully adopted. Treat these as "use at your own risk." They're available for reference but might need modification to work in your environment or might not comply with the latest Sigma specification.

- **deprecated/** – As the name suggests, this folder holds rules that have been retired from active use. A rule might be deprecated if it was replaced by a more effective one, if the technique is no longer relevant, or if maintaining it became problematic. Deprecated rules serve as historical reference. In many cases, the rule files themselves might contain a notice about where the updated detection lives. When using SigmaHQ content, prefer the active rules in the main folders over anything in deprecated/.

- **documentation/** – Contains additional docs, such as the Sigma **rule specification** and conventions. For example, SigmaHQ has markdown files describing field naming conventions, how to contribute, release notes, etc. If you are contributing or want to deeply understand the expected content of fields like description, fields, falsepositives, etc., the documentation folder and the official Sigma website are good resources.

- **Key files:** At the repository root, take note of **CONTRIBUTING.md** (which outlines how to write and submit a rule – e.g. requiring a certain title format, proper tags, etc.), **Releases.md** (which might detail the periodic rule package releases and versioning), and the **.yamllint** config (as mentioned, defines YAML style rules for the repo). There is also a **LICENSE** (Sigma rules are usually licensed under a permissive license, allowing organizations to use and adapt them freely), and a **README.md** which introduces Sigma and points to important links. The README also summarizes the rule types and current project status – for instance, it notes that the repository offers *"more than 3000 detection rules of different type"* and briefly explains the categories.

In summary, the SigmaHQ repository is structured to make it easier to find the detection content you need. Generic, hunting, and emerging threat rules are separated so you can decide which are appropriate for your use case (you might deploy all generic detections to your SIEM, but only run hunting rules on-demand, for example). Within each category, the rules are organized by platform or service (matching the logsource definitions in the rules themselves). This logical layout not only helps in navigation but also is mirrored in Sigma's **release packages** – downloadable bundles of rules filtered by type, security level, or status. By familiarizing yourself with the repo structure, you'll quickly know where to look for a Windows security event rule versus a Linux auditd rule, or how to locate all rules related to a product like Okta or Cisco ASA. It also informs how you might organize your own Sigma rule repository internally: many teams use a similar breakdown (e.g., separating Windows rules from network device rules, and perhaps tagging or grouping rules that are for hunting vs. continuous monitoring). The SigmaHQ structure thus serves as a reference architecture for detection-as-code repositories.

3.3 pySigma & sigma-cli – Modern Converters and Plugins

Sigma rules are written in a generic format, but to actually use them in a SIEM or log management system, they need to be translated into the query language or detection format of that target platform. In the early days of Sigma, this conversion was handled by a tool called **sigmac** (Sigma converter) which was a monolithic script. However, Sigma has since evolved a more powerful and flexible conversion toolchain centered around the **pySigma** library and the **sigma-cli** utility. These represent the "modern" way to convert Sigma to SIEM queries, with a plugin-based architecture for extensibility.

pySigma is a Python library (available on PyPI as pysigma) that provides the core functionality to parse Sigma rule YAML and transform it into queries. Think of pySigma as the engine under the hood – it knows how to read Sigma rules, apply transformations, and produce output. Importantly, pySigma was designed to be modular and extensible: it does not hard-code all the query language conversions itself. Instead, it defines a framework where **backends** (for specific SIEM query languages) and **pipelines** (for data model or field mappings) can be implemented as separate plugins. This keeps the core library vendor-agnostic and lean.

sigma-cli is the command-line interface built on pySigma (the "official" CLI tool). If you've used the old sigmac, sigma-cli is its direct successor – essentially *"sigmac 2.0"*. In fact, the Sigma team notes that **sigma-cli is the equivalent of sigmac for command-line conversion** tasks. The big difference is that sigma-cli + pySigma use a plugin system rather than a single static codebase for all conversions. This means easier maintenance and the ability to add new conversions without changing the core tool.

To get started with sigma-cli, you first need to install it. It's distributed as a Python package, so you can install it via pip:

```
pip3 install sigma-cli
```

This will install the sigma command-line tool (and the pySigma library it depends on). Ensure that your Python 3 installation is accessible and updated (Python 3.9+ is required). After installation, you can check it by running sigma --help or sigma version.

Plugin-Based Conversion: Out of the box, sigma-cli may not yet know how to convert to every SIEM – you have to add the specific "backend" plugins you need. The Sigma project decoupled backends so that you only install what you use. You can list available backends by running:

```
sigma plugin list
```

This will display a table of all officially available backend plugins (and other plugin types) published for Sigma. Each backend has an identifier name. For example, common backend plugins include splunk (for Splunk's SPL query language), qradar (IBM QRadar's AQL), elastic (Elastic's Query DSL/EQL), insightidr (Rapid7 InsightIDR's LEQL), crowdstrike (CrowdStrike LogScale queries), sentinel or ala (Azure Log Analytics / Microsoft Sentinel KQL), carbonblack, cortexxdr, and many more. The list is growing as the community contributes; as of 2025, dozens of backends are available, covering most major security platforms. Each plugin is essentially a Python package that extends pySigma with the logic to convert a Sigma rule into that product's query format.

To use a backend, install it via the plugin system. For example, to install the Splunk backend (which is one of the most popular, given Splunk's widespread use):

```
sigma plugin install splunk
```

This command downloads and installs the Splunk plugin into your sigma-cli environment. Under the hood, plugins are often hosted on PyPI or GitHub; sigma-cli will fetch the correct version compatible with your pySigma. You only need to do this once per backend. After installation, running sigma plugin list again will show the Splunk backend as installed on your system.

(If you ever need to update or remove a plugin, you can use sigma plugin upgrade <name> or sigma plugin uninstall <name> similarly. But typically, plugins will specify a compatible pySigma version and you manage them via pip or sigma's commands.)

Converting Sigma Rules: With the necessary backend plugin installed, you're ready to convert Sigma rules into queries. The primary command is:

```
sigma convert ...
```

You will provide at least two important parameters to this command: the **target backend** and the path to the Sigma rule or rules you want to convert. The target is specified with -t <backend_name> or --target <backend_name>. For example, -t splunk tells sigma-cli to use the Splunk plugin and produce a Splunk query (SPL). The path can be a single .yml rule file, or a directory of rules, or even a glob pattern.

Usually, you will also specify a **pipeline** (-p <pipeline_name> or --pipeline <pipeline_name>). Pipelines in Sigma are configurations that map generic Sigma log fields to the specific fields used in your environment or SIEM. For instance, Sigma's rules might use abstract field names like FilePath or EventID. The Splunk backend's default pipeline for Windows events (called splunk_windows) knows how to map common Windows Event Log fields to Splunk's data model (e.g., Sigma's EventID becomes EventCode in SPL, Sigma's ComputerName might map to Splunk's host field, etc.). Using the right pipeline ensures the query aligns with how your logs are ingested. In our Splunk example, we'll use --pipeline splunk_windows to apply Splunk's Windows field mappings. (Each backend comes with one or more pipelines; many have a default named after the product or data source, and you can list them with sigma list pipelines.)

Let's walk through a real-world example. Suppose we have a Sigma rule (as discussed earlier) that detects if someone disables key protections in Windows Defender. The rule (simplified) might look like this:

```
title: Windows Defender Threat Protection Disabled
logsource:
  product: windows
  service: windefend  # Windows Defender log
detection:
  selection:
    EventID:
      - 5001  # Real-time protection disabled
      - 5010  # Anti-spyware disabled
      - 5012  # Anti-virus disabled
      - 5101  # Defender platform expired
  condition: selection
```

Now, to convert this Sigma rule to a Splunk query, we run:

```
sigma convert --target splunk --pipeline splunk_windows
./rules/windows_defender_threat_detection_disabled.yml
```

Sigma-cli will parse the YAML, apply the Splunk backend conversion logic with the Windows pipeline, and output a Splunk search query. The result would be something like:

```
source="WinEventLog:Microsoft-Windows-Windows Defender/Operational"
EventCode IN (5001, 5010, 5012, 5101)
```

This SPL query can then be used in Splunk to search for those event codes. You'll notice a couple of things in the output: the EventID field from Sigma was converted to EventCode (as per the pipeline mapping for Splunk), and the query includes the specific Windows Event Log source (Microsoft-Windows-Windows Defender/Operational) relevant to Windows Defender events. Sigma's logsource information guided that part of the conversion; the plugin knows, for example, that if service: windefend, it should restrict the source to the Windows Defender event log in Splunk.

If you had instead wanted to convert *all* Sigma rules in a directory to Splunk, you could point sigma convert at a folder. For instance, if you maintain a local repository of Sigma rules in ./myrules/, you could run:

```
sigma convert -t splunk -p splunk_windows ./myrules >
all_rules_splunk_queries.txt
```

This would iterate through every .yml in myrules and output their converted SPL. (By default, multiple rule conversions will be concatenated to STDOUT. Sigma-CLI has options like -o to output each rule to a separate file or to a directory if needed, which is useful for large-scale conversions.)

Output Formats: The default output for each backend is typically a plain query that you can paste into the SIEM's search bar. However, sigma-cli allows switching output formats to integrate with the target platform's preferred format. For example, the Splunk backend supports an output format called savedsearches which produces a Splunk saved search configuration (in Splunk's .conf file syntax) rather than just a raw query. This would include the rule title and description embedded as the saved search name and description. To use this, you add --format savedsearches (or -f savedsearches) to the convert command. Using the Okta rule example from earlier, if we convert it with the savedsearches format:

```
sigma convert --target splunk --pipeline splunk_windows --format
savedsearches ./rules/cloud/okta/okta_user_account_locked_out.yml
```

The output would be a snippet like:

```
[Okta User Account Locked Out]
description = Detects when a user account is locked out.
search = displaymessage="Max sign in attempts exceeded"
```

Here, the Sigma rule's title, description, and detection logic have been formatted as a Splunk saved search stanza. This is ready to be added to Splunk's savedsearches configuration, preserving the context provided by the Sigma metadata. Different backends have different output formats available (another example: an Elastic backend might output either an Elasticsearch DSL query, or an Kibana NDJSON export, etc., depending on what's useful).

Why Plugins Matter: The plugin system in sigma-cli provides flexibility. If a new SIEM platform emerges or you have an internal query language, you (or the community) can write a new backend plugin without having to alter the core Sigma code. In fact, the Sigma community has already created numerous plugins – not only for SIEM products, but also for things like EDRs and cloud platforms. For example, there are backends for Azure Sentinel, Google Chronicle, Amazon CloudWatch Logs, IBM QRadar, ArcSight, Sumo Logic, etc., each maintained in its own repository (often under the SigmaHQ GitHub or by trusted contributors). pySigma and sigma-cli treat these as add-ons. This modular design also means you can **update a single backend** converter independently. If Splunk releases a new feature that requires Sigma conversion changes, the Splunk plugin can be updated on its own schedule. You, as a user, can upgrade that plugin via sigma plugin upgrade splunk without touching your other plugins or the core sigma-cli.

Another benefit is that sigma-cli also supports **pipeline plugins** (for data models) and **validator plugins**. Validators can be used to check Sigma rules for certain conditions (for example, a plugin could implement a rule linting that ensures you didn't use an obsolete field name). By running sigma list validators, you can see if any are installed. These are more advanced features, but they show how the Sigma ecosystem is growing beyond just simple conversion. The **pysigma** core cleanly separates concerns: the **Targets** (backends that define how to convert Sigma to a target query language), **Pipelines** (optional transformations on rules for field/logsource mapping), **Output Formats** (various ways to emit the query), and **Validators** (rule checkers). As a rule author or user, you mostly interact with targets and pipelines via the CLI.

Using sigma-cli in Practice: In day-to-day use, you might integrate sigma-cli into your security operations as follows:

- A detection engineer writes a Sigma rule (using the environment set up in section 3.1). They might test the rule by running sigma convert with a relevant backend to see the output query and ensure it looks correct for the intended SIEM. For instance, if the organization uses Splunk and Azure Sentinel, they might test conversion to both SPL and KQL.
- The team can maintain a library of Sigma rules (perhaps a fork of SigmaHQ plus their own custom rules). They could use sigma-cli to **bulk convert** all Sigma rules to their SIEM's queries whenever the rule set updates. This is especially useful if using Sigma

rules as a feed – e.g., you pull the latest SigmaHQ "emerging threats" rules and convert them to your SIEM to deploy quickly for new threats.

- Security analysts can take a Sigma rule from a blog or tweet (Sigma is often shared in threat intel reports) and run sigma convert themselves to get a query to hunt in their logs immediately. This saves time versus manually translating detection logic.
- Because sigma-cli is a command-line tool, it can be automated. Some organizations include a step in their detection engineering CI pipeline that, after writing or updating Sigma rules, automatically converts them to the appropriate format and then pushes them to the SIEM (via API or other mechanism). This concept of "detections as code" is greatly enabled by sigma-cli: your source of truth is the Sigma YAML, and the tool handles translating that into the live SIEM content.

A concrete example of sigma-cli automation: Let's say we have an internal Git repository with Sigma rules. We could write a small script or Makefile target like make deploy-splunk that does: sigma convert -t splunk -p splunk_windows ./rules -o ./converted/splunk_queries.txt and then calls a Splunk API to load those queries or compares them with what's running. Similarly for other platforms. This way, the Sigma rules in Git become the single source, and sigma-cli ensures all platforms get the updated detection logic in their native tongue.

Modern vs Legacy Tooling: It's worth noting that while sigma-cli/pySigma is the current and forward-looking approach, the older **sigmac** tool still exists (in a sigmatools Python package) but is considered deprecated. The Sigma VS Code extension even had a feature to call sigmac, marked as deprecated now. All new development is happening in pySigma and the plugin repos. Therefore, as a junior/mid-level security engineer, focusing on sigma-cli is the best path. Its usage is well-documented on the SigmaHQ site and in each plugin's README. If you encounter blog posts or guides referencing sigmac or the old tools/ directory in the Sigma repo, be aware that you should transition those workflows to sigma-cli. The concepts are similar (convert Sigma to query), but the commands and capabilities of sigma-cli are more robust (e.g., multiple output formats, better error messages, support for correlations and Sigma meta-features, etc.).

Plugins for Custom Use-Cases: If your SIEM or query language isn't supported yet, you can create your own plugin. This typically involves writing some Python code to define how Sigma AST (abstract syntax tree of the rule) maps to your query syntax. The Sigma community has provided a cookie-cutter template for building new backends, and there's even a blog post by a community member titled "Creating a Sigma Backend for Fun (and no Profit)" that walks through an example. While developing a backend is an advanced topic, the plugin model means even niche log systems can potentially use Sigma as long as someone writes a converter for it. For most users, however, the existing roster of plugins covers the common needs.

In summary, **pySigma and sigma-cli** represent the evolution of Sigma from a simple rules repository to a full-fledged detection engineering platform. With sigma-cli, you hold the power to **integrate Sigma rules into virtually any security analytics tool**. You can write a rule once in Sigma's universal format and convert it to Splunk SPL, to Elastic EQL, to Microsoft Sentinel KQL, or to whatever query language you need – all with a single command and the appropriate plugin. This greatly reduces the toil of maintaining multiple detection queries for different systems. It also means the community's contributions in the Sigma HQ repo are immediately actionable across many platforms. As a security professional, you should become comfortable with sigma-cli: practice installing a backend and converting some rules. Try tweaking a rule's detection logic

and see how the output changes. This will build intuition on how Sigma's constructs (like wildcard searches, lists for OR conditions, etc.) translate into each query language syntax. By using Sigma as your intermediate language and sigma-cli as the translator, you'll streamline your detection development workflow and ensure consistency across the diverse tools in a modern security operations stack.

4. Sigma Rule Conversion and SIEM Integration

Sigma's power lies in its ability to translate generic detection logic into the specific query languages of various security platforms. Each SIEM or log management system has its own query syntax – Splunk uses SPL, Elastic uses Kibana Query Language (KQL) or the Elasticsearch DSL, Microsoft Sentinel uses Azure's Kusto Query Language (also called KQL), IBM QRadar uses AQL, and so on. Sigma provides a bridge between these dialects, allowing you to "write once" and deploy across many tools. In this section, we walk through how Sigma rules are converted and deployed on several popular targets, highlighting field mappings, normalization, and performance considerations for each.

4.1 Splunk (SPL)

Splunk's Search Processing Language (SPL) is a powerful query language for searching and filtering logs. Sigma includes an official Splunk backend that converts Sigma rules into SPL queries. The conversion process must account for Splunk's data schema – for example, what field names are used for a given data source – and craft an efficient search query. Two major considerations when converting Sigma to Splunk are **field mapping** and **search scope**.

Field Mapping for Splunk: Sigma rules use abstract field names (e.g. Image for process path, CommandLine for process arguments, destination.ip for an IP address, etc.). In Splunk, actual field names depend on the data source and any normalization (such as the Common Information Model). The Sigma converter, especially when used with appropriate pipelines, will map these abstract fields to Splunk's field names. For example, in Windows Security Event logs (event ID 4688, process creation), the field corresponding to Image is NewProcessName, and the field for CommandLine is Process_Command_Line *if* that information is captured (e.g. via Sysmon or enhanced logging). In Sysmon logs (event ID 1), the field names align more closely with Sigma's defaults – the process path might appear in a field literally called Image and the command line in CommandLine, if you use Splunk's Sysmon Add-on. The Sigma Splunk backend can apply a *pipeline* (mapping schema) like splunk_windows or splunk_sysmon to automatically choose the correct field names for your log source. It's important to verify these mappings against your Splunk environment. If needed, you can customize the mappings or pipeline so that, for instance, Sigma's Image field is mapped to whatever your Splunk data uses (maybe process_path or a CIM field).

Search Scope and Index Selection: The raw SPL produced by Sigma is often broad to ensure the rule can run in diverse environments. By default, the query might apply to index=* or a very general index and source. In practice, you should *restrict the search scope* to improve performance. For example, if your Windows logs are in index WinLogs, the query should specify index=WinLogs (or whatever index is appropriate) instead of searching all indexes. Likewise, specify the source type if known (e.g. sourcetype=WinEventLog for Windows event logs). Narrowing the search scope ensures the SPL query runs efficiently on large data volumes. Another performance consideration is whether to use accelerated data models. The Sigma Splunk backend can generate queries that use Splunk's tstats command against data models (for example, the CIM *Endpoint* model for process events). These data-model queries can dramatically speed up searches by using summary indexes, but require that your Splunk environment is populating the corresponding data model. It's an option to consider for frequent or high-volume rules.

Example – Converting a Sigma Rule to SPL: Consider a Sigma rule that detects dumping of the LSASS process memory using Sysinternals ProcDump (a technique for credential theft). In Sigma (YAML), the detection might look like this:

```
title: LSASS Dump via ProcDump
logsource:
  product: windows
  category: process_creation
detection:
  selection:
    Image|endswith: '\procdump.exe'
    CommandLine|contains: 'lsass.exe'
  condition: selection
```

After running the Sigma converter with the Splunk backend (and using the appropriate pipeline for Windows Security logs), we might get an SPL query like the following:

```
index=winlogs source="WinEventLog:Security" EventCode=4688
(NewProcessName="*\\procdump.exe" AND CommandLine="*lsass.exe*")
```

This query searches the Windows Security event log for process creation events (EventCode 4688) where the new process name ends in procdump.exe and the command line contains lsass.exe. Note how the Sigma fields Image and CommandLine have been translated to NewProcessName and CommandLine in the SPL, and the log source was narrowed to WinEventLog:Security with EventCode 4688 (since Sigma's process_creation on Windows implies those events). The wildcard *\\procdump.exe is used to ensure the match even if the process was executed from a path (the double backslash in the query represents a literal backslash in the value). In Sigma we specified endswith '\procdump.exe', and the converter implemented that with a wildcard search in Splunk.

Deployment in Splunk: Once you have the SPL query, deploying it in Splunk involves creating a saved search or an alert. It's good practice to test the query first in Splunk's Search app to ensure it returns the expected results (and to check how often it triggers). After testing, you can **save the search as an alert** (if using core Splunk) or as a **correlation search** in Splunk Enterprise Security. In Splunk Web, you would go to **Save As -> Alert**, give it a name (e.g. "ProcDump LSASS Dump Attempt"), set a schedule (or real-time), and define trigger conditions (e.g. trigger when number of results > 0). You would also configure actions – such as sending an email, creating a Splunk alert event, or executing a script. Once enabled, Splunk will continuously monitor for this pattern and generate alerts when the Sigma rule's conditions are met. Essentially, the Sigma rule has become a live Splunk detection.

Splunk Integration Tips:

- **Index/Sourcetype Tuning:** Always adjust the index=... and other source qualifiers in the generated SPL to fit your environment. The example above uses a generic winlogs index for illustration; in your case it might be index=main or a specific index for Windows security events. Restricting the search by index and sourcetype improves performance and reduces noise.
- **Use of Data Models:** If you have Splunk Enterprise Security or CIM data models, consider using Sigma's ability to output data model queries. For example, a Sigma rule looking for process events could be converted to an SPL query against the

Endpoint.Processes data model via tstats. This can make the search much faster on large Splunk deployments. However, verify that the fields used (CIM fields) align with your data.

- **Escape Characters:** Sigma rules may include wildcards or regex. The converter will handle escaping special characters in SPL, but it's wise to double-check. In SPL, backslashes and quotes must be properly escaped. The above query shows an escaped backslash \\ in the string. If you modify or extend the query (e.g. add regex), ensure the syntax is correct for Splunk.
- **Field Name Customization:** If your Splunk uses custom field names (for instance, a different TA that names fields differently), you might need to tweak the generated query. Sigma's default mappings cover common cases, but every Splunk deployment can have slight differences. Be prepared to adjust field names or add aliases in the query to match your environment.

By considering these factors – correct field mapping, scoping your search, and tuning the query – you can effectively deploy Sigma-derived rules in Splunk. Splunk will then continuously monitor your data for the conditions described by the Sigma rule and alert you in real time when suspicious events occur.

4.2 Elastic Stack (KQL / ES DSL)

Elastic Stack's security analytics (formerly Elastic SIEM) uses Elasticsearch as the search engine and allows detections to be written in Kibana Query Language (KQL) or EQL (Elastic Query Language) for sequences. Sigma's Elastic backend can produce queries in multiple forms to suit these options: it can output a simple KQL filter query, a Lucene query, an EQL sequence, or even a fully formed JSON DSL query for Elasticsearch APIs. The most common route is to generate a KQL query that can be used directly in Kibana's Detection engine.

Conversion to KQL – Example: Let's use the same ProcDump/LSASS Sigma rule scenario. Using the Sigma CLI with the elasticsearch backend (and an appropriate config/pipeline for Elastic), the Sigma rule can be translated into a Kibana query. For instance, one output might be:

```
event.code:4688 and process.name:"procdump.exe" and
process.command_line:*lsass.exe*
```

This KQL query is looking for events where the event code is 4688 (Windows process created), the process name is "procdump.exe", and the process command line contains the substring lsass.exe. In Elastic Common Schema (ECS), Windows Security event 4688 is typically indexed with event.code: 4688. The process name and command line fields are normalized as process.name and process.command_line in ECS (assuming Winlogbeat or Elastic Agent is parsing the event). The syntax here uses : for field-value matching and * wildcards for partial matches. In KQL, writing multiple conditions separated by and (or simply space-separated) implies all must be true, similar to a logical AND. In the above example, we included and explicitly for clarity, but it could be omitted.

Elastic's Sigma backend is smart about using the right fields based on the logsource. Since our Sigma rule's logsource was Windows process creation, the converter knew to include event.code:4688. If we were targeting Sysmon logs instead, the query might use

winlog.event_id:1 (Sysmon's process create event) and possibly different field names. For example, a Sysmon variant could be:

```
winlog.event_id:1 and process.executable:*\\procdump.exe and
process.command_line:*lsass.exe*
```

where process.executable is an ECS field for the full process path in Sysmon logs. The Sigma converter will adjust the query based on the logsource in the rule – whether it's Windows Security Event or Sysmon – as long as you use the correct pipeline or config mapping. It will try to match the condition to the expected index fields automatically, which is a big advantage.

Deploying Sigma Rules in Elastic Security: Once you have a KQL query, integrating it into Elastic is straightforward. In Kibana's Security app, go to **Detections** and create a new rule (Custom query rule). You'll specify the index pattern that contains your logs (e.g. logs-security-windows* if that's where your Windows event logs reside, or winlogbeat-* by default). Then you paste the KQL query into the rule's query field. You will also configure the schedule (how often to run the query, say every 5 minutes) and the timeframe it looks at (e.g. search the last 5 minutes of data on each run). Set the rule actions such as creating alerts or sending notifications. Once activated, the rule will run the KQL query on incoming data and generate alerts for any matches. Essentially, Kibana will continuously execute the Sigma-derived query in the background.

Elastic/KQL Considerations: Elastic's search has its own nuances:

- **Index and ECS Schema:** Sigma queries themselves don't include index names; you must specify the index pattern when creating the rule (or in the Kibana query form). Ensure you point to the correct indices (e.g. winlogbeat-* or whatever index holds the data). Also confirm that your data is mapped to the field names the Sigma query uses. If you're using Elastic Common Schema (as Beats and Elastic Agent do), fields like process.name and process.command_line will exist. If not using ECS, you might need to adjust the query to your field names (this is where a custom Sigma config could map to your schema).
- **Case Sensitivity:** By default, KQL matching on text is case-insensitive. In our example, it will catch "lsass.exe" regardless of case. This is convenient, but be aware it depends on Elasticsearch analyzers. If a field is a keyword (untouched) versus text, the matching rules differ. In general, you don't need to worry about case for most string fields in KQL queries.
- **Regex vs Wildcards:** Sigma allows regex patterns in conditions, but Kibana KQL does **not** support full regex in the query (Lucene syntax can, if you use Lucene queries in Kibana). The Sigma converter will typically convert regexes to wildcards or Lucene where possible. For example, if a Sigma rule had CommandLine|contains:-Enc (looking for -Enc in a command line), the KQL output might be process.command_line:* -Enc * (wildcard around -Enc). Complex regex from Sigma might not translate cleanly to KQL; in some cases the tool may fall back to Lucene DSL or require manual tuning. Always review the output if your Sigma rule used regex or special modifiers.
- **Testing and Tuning:** Just as with Splunk, it's wise to test the converted KQL in Kibana's Discover or the query console before relying on it in a rule. This lets you verify that it returns the expected events and not an overwhelming number of false positives. You

might discover you need an additional filter (e.g. host name, process parent, etc.) to make it more precise. Fortunately, editing the query in Kibana is easy – you can iterate quickly to tune it.

By converting Sigma rules to KQL and integrating them into Elastic Security, you leverage Elastic's speed and scalability for detection. The underlying Elasticsearch engine will execute the query on each interval. Keep an eye on performance: a query with broad wildcards (like *lsass.exe* without any other filtering) could be heavy. In our example, we included event.code:4688 as an early filter, which is good practice. Always filter on indexed fields (like event code, log category, etc.) first to reduce the data that needs wildcard matching.

Finally, note that Elastic also supports EQL for more complex sequence rules (e.g. detect A then B within X minutes). The Sigma elasticsearch backend can output EQL if the Sigma rule was written as a correlation rule. Implementing those in Elastic would involve choosing the "EQL" rule type in Kibana and pasting the EQL query. EQL sequence detections are powerful but beyond our simple example. For many single-event detections, KQL is sufficient and more straightforward.

4.3 IBM QRadar (AQL)

IBM QRadar uses the Ariel Query Language (AQL) to search its event database. Sigma's QRadar backend converts Sigma rules into AQL queries, typically a SELECT ... FROM events WHERE ... statement that captures the Sigma logic. Deploying a Sigma rule in QRadar is a bit different from Splunk/Elastic, because QRadar's real-time detection is usually done through its rule engine (offense generation) rather than scheduled queries. However, the AQL output from Sigma is extremely useful for creating saved searches, reports, or as a basis for QRadar rule conditions.

Field Mapping in QRadar: In QRadar, a lot of raw log data is stored in the payload field, and important attributes are extracted into normalized fields by DSMs (Device Support Modules). For example, QRadar's Windows Security DSM will pull out fields like EventID (4688), maybe New Process Name, Creator Process Name, user account, etc., if those are present in the event. However, not every field that Sigma might reference is guaranteed to exist as a discrete field in QRadar. For instance, Windows Security event 4688 **does not** include the full command line by default (without Sysmon, the OS doesn't log command line in event 4688). So a Sigma rule that looks at CommandLine might not have a direct field in QRadar's 4688 events. In such cases, the Sigma converter will often resort to text search on the payload. In general, the QRadar Sigma backend tries to map known fields to QRadar's properties, but if something isn't mapped, it may leave the generic name or use payload contains/LIKE clauses. It's important to verify that the fields in the AQL correspond to actual QRadar event properties in your system. QRadar field names sometimes differ (e.g. Sigma's Image might map to QRadar's New Process Name property if available, which contains the process file name). The Sigma community and IBM have provided mappings for many common log sources in the QRadar backend, but you should double-check especially if using custom logs.

Example AQL Conversion: Using the ProcDump/LSASS rule example, a possible Sigma-to-AQL conversion could be:

```
SELECT QIDNAME(qid) AS EventName, logsourcename(logsourceid) AS
LogSource, UTF8(payload)
FROM events
WHERE EventID = 4688
```

```
  AND UTF8(payload) ILIKE '%\\procdump.exe%'
  AND UTF8(payload) ILIKE '%lsass.exe%'
```

This AQL query searches the events database for events with EventID = 4688 (i.e. a process creation event, as parsed by the Windows DSM) and uses ILIKE (case-insensitive substring match) on the UTF-8 decoded payload to find occurrences of "procdump.exe" and "lsass.exe". The UTF8(payload) function is used to ensure the text search works on the log message, and the percent % wildcards are equivalent to * in SQL patterns. Essentially, this is looking for any log event 4688 where the raw log message contains those two strings. We select a few fields like the QID Name (a descriptive event name in QRadar) and LogSource just for context in results – those selections aren't strictly required, but often helpful.

What if QRadar has specific parsed fields for these values? In some environments, if Sysmon logs were ingested or the DSM was extended, there might be custom properties. For example, QRadar might have a custom property for the process name. If so, the query could be more precise, like:

```
WHERE EventID = 4688
  AND "New Process Name" ILIKE '%procdump.exe'
  AND "Creator Process" ILIKE '%lsass.exe%'
```

(using whatever the exact field names are in QRadar for "New Process Name" and perhaps the process that was targeted). However, in standard Windows events, "Creator Process" and command-line arguments are not available without Sysmon or enhanced logging. So the safer bet is often to search the payload. The Sigma default approach above with payload ILIKE will catch strings anywhere in the log message, which is broad but ensures detection even if fields aren't parsed.

Deploying the Rule in QRadar: Running the AQL query as an ad-hoc search in QRadar's Log Activity tab can help verify the Sigma rule logic. You'd set the time range and execute the query to see if it finds the relevant events. For operational detection (generating offenses or alerts), you would typically create a QRadar rule (in the Rule Editor) that uses conditions corresponding to the Sigma logic. For example, you could create a rule that triggers when an event matches EventID = 4688 **and** the payload contains "procdump.exe" **and** "lsass.exe". QRadar rules allow payload content conditions and property conditions. The Sigma AQL essentially tells you what to look for, and you implement that as a QRadar rule so that when such an event arrives, QRadar will fire an offense. The converted AQL can guide how you build the rule (or you could schedule the AQL as a saved search if you prefer periodic detection). Keep in mind that QRadar's real-time engine is optimized for property-based rules; doing multiple payload ILIKE matches in a rule might be slightly heavy, but it is feasible if the volume is not too high. You might also incorporate QRadar's reference sets or building blocks for performance (beyond our scope here).

QRadar Integration Tips:

- **Field Availability:** Ensure the events you need are parsed. If the Sigma rule references a field that QRadar doesn't parse by default (like command line), consider sending those logs in a way that DSM can parse them (e.g. forwarding Sysmon logs to QRadar with a DSM or custom properties). Otherwise, be prepared to rely on payload ILIKE searches.
- **Field Naming:** The Sigma converter might output generic field names in quotes (like "New Process Name"). As the CardinalOps example notes, these field names can vary

by QRadar version or language pack. You may need to adjust them to exactly match your QRadar environment's property names. Use the QRadar Log Activity filter or payload viewer to confirm actual field labels.

- **Performance:** Searching the payload with multiple ILIKE conditions can be less efficient. Try to include at least one indexed condition. In our query, EventID = 4688 is an indexed filter (EventID is a normalized field), which is good – it limits search to process creation events. Always include such a narrowing condition if possible (e.g. a QID or EventName filter). This way QRadar scans a smaller subset of events for the string matches.
- **Testing in Smaller Windows:** When writing a new rule or query, test on a limited time range or specific log source to gauge performance and results. You don't want to accidentally run an unbounded payload search over weeks of data.

By converting Sigma rules to AQL, you can accelerate the creation of QRadar content. It saves time in figuring out the exact query syntax. After conversion, you will typically implement it as a QRadar rule for continuous monitoring. The result is that the Sigma detection logic (e.g. "ProcDump used on LSASS") becomes part of QRadar's offense generation, alerting the SOC whenever that pattern appears in the logs.

4.4 Microsoft Sentinel (KQL)

Microsoft Sentinel (formerly Azure Sentinel) is a cloud-native SIEM that uses Azure Log Analytics as its data platform. Sentinel's detection rules are written in Kusto Query Language (KQL), the same query language used by Azure Data Explorer and Microsoft Defender for Endpoint's Advanced Hunting. Sigma rules can be converted into Sentinel KQL queries which can then be used in Sentinel Analytics Rules.

Data Source and Field Mapping in Sentinel: Data in Sentinel is organized into tables in Log Analytics workspaces. Each data source (Windows events, Azure AD logs, Office 365, Defender ATP, etc.) has its own table name. For example, Windows Security events are in the SecurityEvent table, Azure AD logs in AzureActiveDirectory table, Defender ATP incidents in DeviceEvents or more specific tables like DeviceProcessEvents for process data. Sigma's logsource metadata (e.g. product: windows, service: security) gives a hint, but the converter does not inherently know the table name – you will choose the correct table when writing the KQL. Fortunately, many field names in Azure Sentinel are intuitive and often match Sigma's abstract names or Windows event schema. For instance, in the SecurityEvent table, there are fields EventID, NewProcessName, CommandLine, etc., which correspond to the Windows event attributes (the Azure monitoring agent parses those out). In Defender's DeviceProcessEvents (Advanced Hunting schema), fields are named slightly differently (e.g. FileName for the process image, ProcessCommandLine for the command line). The Sigma Kusto backend will use a default schema (often assuming Azure SecurityEvent or the Defender Advanced Hunting schema depending on context). You may need to adjust the table and possibly field names when deploying the query, but in many cases it's a close match.

Example KQL Query (Sentinel): For our ProcDump on LSASS example, if we target Windows Security logs in Sentinel, we would use the SecurityEvent table. A KQL query could be:

```
SecurityEvent
| where EventID == 4688
```

```
| where NewProcessName endswith @"\\procdump.exe"
| where CommandLine contains "lsass.exe"
```

Let's break this down. We explicitly start with the SecurityEvent table (which contains all Windows security log events). Then we pipe to a where clause filtering EventID to 4688, to select only process creation events. Next, we pipe to another filter on NewProcessName using the endswith function to match processes ending in "\procdump.exe". We have to escape the backslash in the string by writing \\"\\procdump.exe" or use the verbatim string literal as shown with the @ symbol (Kusto allows @"C:\path" string literals for convenience). Finally, we filter where CommandLine contains "lsass.exe". The effect is that all three conditions must be true for an event to pass through. The query is easy to read: it's essentially the same logic as our Sigma rule expressed in KQL. We have ensured the filters are applied in order – filtering by EventID first (which is efficient) and then by the text contents of fields. KQL by default is case-insensitive for string operators like contains, so this will catch "LSASS.exe" or any case variation as well.

If we were instead looking at Defender Endpoint data, we might query the DeviceProcessEvents table. An equivalent KQL for Defender data could be:

```
DeviceProcessEvents
| where FileName =~ "procdump.exe"
| where ProcessCommandLine contains "lsass.exe"
```

Here =~ is the case-insensitive equals operator in KQL, used to match the FileName (process name) exactly to "procdump.exe" regardless of case. And we use ProcessCommandLine contains "lsass.exe" similarly. The example shows that whether logs come from Sentinel's SecurityEvent or Defender's DeviceProcessEvents, KQL can express the Sigma rule clearly. The main difference is just the table and field names.

Implementing the Rule in Sentinel: To deploy this query, you create a new Analytics Rule in Microsoft Sentinel (via Azure portal or Defender portal). Choose a Scheduled Query Rule. In the rule wizard, you paste the KQL query and then specify the rule details: the schedule (e.g. run every 5 minutes looking at the last 5 minutes of data), the trigger threshold (typically > 0 results to trigger an incident), and the severity, alert grouping, etc.. You would also map the Sigma rule's tactics/techniques if you want (Sentinel allows you to tag the rule with MITRE ATT&CK tactics for context). After creating and enabling the rule, Sentinel will run the query on incoming data and generate an alert (incident) whenever the query returns a result. That alert can then be investigated in Sentinel, just like any built-in detection.

Sentinel/KQL Tips:

- **Optimize Query Order:** KQL can handle large data, but you should filter early by using specific conditions first. In our example, putting EventID == 4688 at the top of the query drastically reduces the events to inspect for string matches – this is a good practice. Avoid using broad contains on huge tables without narrower filters, as it will cost more and run slower.
- **String Matching Functions:** We used contains and endswith which are fine here. KQL also has operators like has (which looks for whole term matches) and contains_cs (case-sensitive contains) if needed. Sigma's converter typically chooses contains or explicit functions like endswith to reflect the rule. Be aware that contains in KQL is case-

insensitive by default (as noted). If you need case-sensitive matching, there are _cs variants.

- **Verify Field Names and Schema:** Use Sentinel's schema reference or the Log Analytics schema browser to confirm field names for the data you care about. For example, Sigma might use UserPrincipalName in a rule, but in Sentinel's Azure AD logs the field might be TargetUserUpn – a different name. In our case, NewProcessName and CommandLine in SecurityEvent were straightforward. Always adjust the query to the actual column names in the table. The Sigma converter's output is a starting point; small tweaks might be needed if there's a naming mismatch.
- **Testing Queries:** Just like with other platforms, test your KQL query in the Log Analytics query window first. See if it returns the expected events, and tweak it if you get too many/too few results. Maybe you find you need to add | where Computer contains "DC" if the event should only come from domain controllers, for instance – that kind of tuning is typical.

With Sigma-to-KQL conversion, you can quickly build Sentinel detections that align with community rule logic. Sentinel's advantage is that KQL is quite expressive and the platform is scalable, but you must always keep an eye on query efficiency. A poorly written query (for example, scanning an entire table for a common string) could incur high resource usage. The Sigma converter generally provides a reasonable query, but it's up to the analyst to ensure it's optimized and contextually appropriate before enabling it in production. Once tuned, the Sigma rule runs as a native Sentinel analytic, continuously monitoring your cloud and on-premise logs for the defined threat behavior.

4.5 Other Targets (Sumo Logic, ArcSight, etc.)

One of Sigma's biggest strengths is its broad ecosystem support. In addition to the platforms above, the community has developed Sigma backends for many other SIEM and EDR products. This means you can often convert a Sigma rule into whatever query or rule format your tool requires. We'll briefly discuss a couple of other common targets – Sumo Logic and ArcSight – and mention others in passing.

Sumo Logic: Sumo Logic is a cloud-based log management and analytics platform with its own query language, which is structured as a sequence of pipeline operators (somewhat akin to Splunk or KQL). Sigma's Sumo Logic backend can translate rules into Sumo query language, using constructs like parse and where to filter logs. Field mapping is again the key consideration – Sumo may require explicit parsing of JSON or raw text to get the fields. For example, if you have Windows Event logs in JSON format in Sumo, a Sigma rule might become a Sumo query that first parses the JSON and then applies where conditions. A concrete instance: a Sigma rule for detecting suspicious Run Key registry entries (persistence via Windows registry) could be converted to a Sumo query that uses the json operator to extract fields from log data and then a where to apply the conditions. Sumo's query might look like:

```
_sourceCategory=Windows/Security "4688"
| json field=_raw "EventID" as EventID nodrop
| json field=_raw "EventData.NewProcessName" as NewProcessName
nodrop
| json field=_raw "EventData.ParentProcessName" as ParentProcessName
nodrop
```

```
| where EventID = 4688 and NewProcessName matches "*\\procdump.exe"
and ParentProcessName contains "lsass.exe"
```

(This is illustrative, combining concepts from a Sumo query example.) Here we restrict to logs containing "4688", parse out JSON fields for EventID, NewProcessName, etc., then filter on those. The Sigma backend for Sumo will attempt to produce such structured queries. It leverages common Sumo operators like json auto or parse to get at fields, because unlike Splunk/Elastic, Sumo might not automatically extract everything without instructions. You may need to tweak the parsing depending on your log format, but the core logic from Sigma will be preserved. The performance implications for Sumo are similar to others: ensure you narrow the scope (_sourceCategory or _collector in Sumo) and use parsing efficiently. Sumo is quite capable of handling large data, but unnecessary wildcards or not using indexed terms can slow it down. The Sigma-converted query gives you a baseline which you can optimize (for example, adding | where _timeslice between (...) for time, or ensuring the JSON parsing is done only once).

ArcSight (Micro Focus / OpenText ArcSight ESM): ArcSight has been a traditional SIEM platform which uses a proprietary query/filter syntax. ArcSight content is typically created through its GUI as filters and rules (for real-time correlation). There is a Sigma backend for ArcSight that converts Sigma rules into ArcSight filter conditions. Instead of a single search query, you might get a set of conditions that can be applied in an Active Channel or rule. For instance, the Sigma rule conditions (like process name and command line) would be mapped to ArcSight field names such as TargetProcessName, EventName, etc., in an ArcSight filter. You could then import or manually create a filter in ArcSight ESM with those conditions, and build a correlation rule around it. One challenge is ArcSight's field naming conventions: for example, Sigma's FileName might correspond to an ArcSight field labeled File Name (with a space) or some similarly named field in the event schema. You often have to adjust the output slightly because ArcSight may use different capitalization or spacing. The Sigma integration pack provided by SOC Prime allowed analysts to automatically generate ArcSight content from Sigma. In practice, to deploy a Sigma rule in ArcSight, you'd convert it, get the filter conditions, then create a content rule in ESM that uses those conditions to trigger an alert. ArcSight's normalization (via SmartConnectors) means if the log source is known (e.g. Windows events), many fields like process name, event ID, etc., will be available with standard names. If some aren't, you might have to use ArcSight's payload contains logic or FlexConnector mapping to expose them. The key is that Sigma gives you a head start so you're not writing the ArcSight filter from scratch.

Other Platforms: The list of Sigma-supported targets is growing and covers nearly the entire spectrum of SIEM and EDR tools. To name a few: **Logpoint**, **RSA NetWitness**, **Google Chronicle**, **CrowdStrike Falcon LogScale (Humio)**, **Graylog**, **Rapid7 InsightIDR**, **Devo**, **Snowflake**, and even formats like **STIX/Shifter** or YARA-L. For many of these, Sigma backends or community contributions exist. For example, Google Chronicle has a Sigma translation tool (Chronicle can even import Sigma rules directly in some cases), Graylog has a plugin to consume Sigma rules, and others have their own conversion utilities. Even if a platform doesn't natively support Sigma, the open-source community often provides a converter or at least a guide to mapping Sigma fields to that system. This ubiquity is what makes Sigma truly a universal detection language – a well-written Sigma rule can be shared and used across organizations regardless of the SIEM they use.

When using Sigma on these varied targets, keep a few general considerations in mind:

- **Conversion Quality Varies:** Not all backends are equally mature. Common log sources (Windows, Linux, common applications) tend to be well-supported, whereas niche logs might need manual tweaking. Always review the output query or rule. If something looks off (e.g. a field name that doesn't exist in your platform), adjust it.
- **Verify Field Mappings:** This cannot be overstated – correct field mapping is essential. If the default mapping in Sigma's backend doesn't match your environment, utilize Sigma's configuration options or pipelines to define your own mappings. For instance, map Sigma's Image to proc_image if that's your field. A Sigma rule will fail silently (no detections) if it's looking for data in the wrong field name, so invest time in mapping alignment.
- **Data Availability:** Sigma rules assume certain data is being collected. Ensure your SIEM actually has those logs. For example, a Sigma rule for PowerShell script block logging is useless if those logs aren't ingested. Before converting a batch of rules, verify you have the coverage – otherwise you'll need to onboard that log source or modify the rule for available data.
- **Performance Implications:** Sigma rules are written to be general, which sometimes means the converted queries are not the most optimized for your environment. Use them as a starting point and then optimize. This could mean adding an index/time filter, breaking a single rule into multiple narrower ones, or leveraging indexed fields instead of regex where possible. Test the query on a large dataset to see how it performs. For instance, a rule with many OR conditions or wildcards might need refinement (maybe splitting into separate detections or using a different approach) to avoid heavy load on your SIEM.
- **Tuning Alerts:** Once deployed, monitor the Sigma-based rules for false positives or false negatives. Sigma often comes with a falsepositives section in the rule description (as text). It's up to you to implement those in your query (e.g. filtering out known benign processes). Adjust the queries over time – add exclusions, or broaden conditions if you missed some variants. This tuning is part of operationalizing Sigma content.
- **Rule Updates:** The Sigma community continuously updates and adds new rules. Keep an eye on the SigmaHQ repository and other sources. You might set up a process to periodically pull new Sigma rules or updates and convert them to your SIEM format. Automation is possible (for example, a script running sigma-cli to convert the latest rules to your target). This ensures you benefit from the latest threat detection knowledge.

In summary, Sigma's flexibility extends to a wide array of security platforms. Whether you are using a modern cloud SIEM or an older on-premises solution, Sigma can likely help standardize and share detections. By understanding the platform-specific considerations – how fields are named, how queries are structured, and how to optimize them – a security analyst can take a Sigma rule from a repository like SigmaHQ and deploy it as a working detection in their tool of choice. This capability greatly accelerates detection engineering and fosters a community-driven approach to cyber defense, where a detection idea is not confined to one platform but can be leveraged by many.

5. Advanced Engineering & Best Practices

5.1 Performance Tuning and False-Positive Reduction

Writing effective Sigma rules isn't just about detecting threats – it's about doing so efficiently and accurately. A poorly optimized rule can slow down your SIEM searches or overwhelm analysts with benign alerts. As one Sigma engineer noted, detection engineering is a balancing act: **"Build rules too specific, and attackers might evade your detections. Build rules too generic, you and the rest of your SOC are going to [spend too much time on] triage"**. In this section, we explore how to tune Sigma rules for performance and reduce false positives, ensuring your detections are both speedy and precise.

Selecting the Right Logs and Fields: The first step in tuning a rule is narrowing its scope to the relevant log data. In Sigma, the logsource field (with product, service, category) should be as specific as possible to restrict the search to the logs that matter. For example, if your rule is meant for Windows process creation events, specify product: windows and category: process_creation (and even a specific service like Sysmon if applicable) so that the SIEM doesn't search across all log types. Within the detection **conditions**, use selective fields to narrow down candidates early. For instance, include an Event ID or event type code if known – checking for a specific Windows EventID (like 4688 for process creation or 4625 for failed logon) can drastically reduce search volume by filtering out irrelevant events upfront. By choosing fields that are indexed or highly selective (such as source IP, username, process name, etc.), your converted queries will execute faster.

Condition Simplification: Simplify your matching conditions to avoid unnecessary complexity. Sigma offers a variety of match operators (called *modifiers* in Sigma syntax) such as contains, endswith, startswith, regex (re), etc. Each has performance implications on the backend. A good practice is to **avoid heavy regex patterns when a simpler operator will do**. For example, instead of a regex like CommandLine|re: '.*\\.exe', use a suffix match: CommandLine|endswith: '.exe'. This yields a more efficient query (e.g. a wildcard search for "*.exe") and is easier to read. In general, prefer direct string matches or wildcards anchored to one side (prefix/suffix) over full substring searches or regex, since many SIEMs can optimize the former. Also, try to combine conditions logically to reduce false hits – for instance, require two indicators of malicious activity rather than one. If a rule currently triggers on any use of nslookup.exe, consider adding an additional condition that the command line includes suspicious domains or that the parent process is unusual, so benign uses are filtered out. Every extra condition that meaningfully narrows the result set can both improve precision and reduce the work your SIEM has to do.

Optimizing Key Fields: Be mindful of which fields you include in the output query. Some SIEM backends have known performance characteristics – for example, searching by an indexed IP address or event code is fast, whereas searching within a raw message or unindexed text field is slow. While Sigma itself is platform-agnostic, a rule writer should still **optimize for common denominators**: use fields that most backends would treat as searchable keys (usernames, process names, IDs) and avoid free-text whenever possible. If you must search text (like an error message or command line), see if there's a way to anchor it (e.g. |contains with a specific keyword) rather than using broad wildcards. Likewise, be cautious with the all modifier (which requires all items in a list to match) or long lists of OR values – these can bloat the translated query. It's often better to split extremely broad detections into multiple narrower Sigma rules, each tuned for a specific scenario, which can then be maintained and optimized individually.

False-Positive Reduction Techniques: Even a performant rule is of little use if it fires on benign activity too often. Reducing false positives is critical. Sigma rules support an optional falsepositives list in the metadata where you document known benign triggers (e.g. *"administrators often run this tool during maintenance"*), but **to actively filter false positives, you need to reflect that logic in the detection section**. A common pattern is to include an **exclusion** condition using a filter or a not clause. For example, consider a Sigma rule intended to detect suspicious network connections initiated by Notepad (perhaps indicating DLL hijacking or misuse of notepad.exe). Notepad might legitimately make network connections when printing (to port 9100, a common print protocol). We can build the rule with a filter to exclude that scenario:

```
title: Network Connection Initiated via Notepad
logsource:
  category: network_connection
  product: windows
detection:
  selection:
    Image|endswith: '\notepad.exe'
  filter:
    DestinationPort: 9100
  condition: selection and not filter
falsepositives:
  - Printing documents via Notepad might trigger connections on port
9100 (common network print port).
```

In this rule, the selection catches any network connection where the process image is Notepad, and the filter then excludes events where the destination port is 9100 (which we expect when printing). The condition selection and not filter ensures that any Notepad network connections *except* those to port 9100 will trigger. The falsepositives field explains the rationale. This tuning greatly reduces noise in an environment that uses port 9100 for legitimate printing.

Now, what if your environment's printing uses a different port or protocol (say IPP on port 631)? In a vanilla Sigma rule, you'd have to modify or add another filter for port 631, and maybe another to ignore connections to your file share servers (e.g. port 445) if Notepad can load files from a network drive. You can certainly do that – adding environment-specific exceptions as additional filter_* clauses and combining them (as in condition: selection and not (filter_printing or filter_file_share)) – but maintainability suffers. **Each new exclusion makes the rule less generic and harder to share**, and you might need to update many rules if the same false-positive condition affects multiple detections. This is where Sigma's new *Filters* feature comes in (introduced in 2024). Sigma Filters are separate YAML files that define reusable exclusion logic, which can be applied to one or many rules at conversion time. For instance, you could create a filter rule that says "exclude events from hosts starting with DC- (Domain Controllers)" and apply that filter to all authentication Sigma rules that tend to fire on DC activity. Because filters are maintained outside the core rule, the original detection logic stays "clean" and generic, and your environment-specific noise reduction can be toggled on as needed. Filters support the same syntax as Sigma detections (they have a filter section with selection and condition: not selection), plus they list which Sigma rule(s) they apply to by name. This modular approach helps performance (by cutting out known noise) *and* avoids false positives without hard-coding every rule for one environment.

Even without using the new filters feature, you can achieve a lot of false-positive suppression through careful rule design. Use **negative conditions** (the and not pattern) to subtract known benign events. Maintain allow-lists where appropriate – for example, if only certain admin accounts or processes should legitimately perform an action, consider building that into the rule (either as a filter or by scoping the logsource/user in the rule itself). Another practical example: a brute-force login detection might ignore authentication attempts from service accounts. Many Windows service accounts have names ending with a $. Your Sigma rule for failed logon attempts could include an exclusion like SubjectUserName|endswith: $ in a filter section, so that you don't count those machine account logon failures in your alert. This way, an automated process failing to authenticate doesn't set off your "possible brute force" alarm. The translated query will contain a condition like NOT SubjectUserName="*$" (in Splunk syntax, as shown in the Sigma docs). Such tuning, combined with threshold conditions (e.g. "10 failed logins from the same user within 5 minutes"), drastically improves fidelity.

Performance-Conscious Sigma Patterns: In summary, to optimize Sigma rules:

- *Scope your searches narrowly:* Use specific logsource and event attributes so the SIEM isn't trawling through irrelevant data.
- *Favor simple operators:* Use exact matches, prefixes/suffixes or contains for known keywords. Avoid full regex or unbounded wildcards unless truly necessary.
- *Reduce data scanned:* If a rule can be expressed as "X and Y occurring together," consider if one of those can be used to pre-filter the data (e.g. filter by event type X, then within those events look for Y). This often can be done by structuring the Sigma rule into multiple selections and a composite condition.
- *Exclude the obvious noise:* Implement **filters or negative conditions** for well-understood benign patterns (software update processes, admin activity, etc.). It's better to miss an alert on a known safe condition than to bombard analysts with alerts they will always dismiss. Document these in the falsepositives field for transparency.
- *Leverage Sigma's metadata:* Use the rule's level appropriately to reflect confidence – e.g. rules with higher levels (high/critical) should be tuned so that false positives are nearly nonexistent. If a rule can't avoid occasional benign triggers, it might be marked as medium severity or informational, guiding how it's handled when it fires. This indirectly improves "performance" by focusing analyst time on the truly critical alerts.

By thoughtfully tuning queries and conditions, you ensure Sigma rules run lean and deliver actionable results. Now that we've covered making individual rules efficient and accurate, the next step is to enrich those detections with more context.

5.2 Contextual Correlation & Enrichment Strategies

No security event happens in isolation. A single log entry can be suspicious, but its true significance often emerges only when you consider context: what else happened around that event? who or what was involved? was it part of a larger sequence? In this section, we discuss how to extend Sigma detections beyond single events, through correlating multiple signals and enriching alerts with additional data. These strategies help differentiate true incidents from noise and provide analysts with richer information to respond effectively.

Multi-Event Correlation with Sigma: Traditionally, Sigma rules have described single-event patterns (e.g. "a process creation with these command-line parameters" or "a login failure from this source"). However, advanced detections often require linking **multiple events** together. For example, ten failed logins for one user in 5 minutes might indicate a brute force attempt, even if each individual failed login could be benign. To address this, the Sigma specification introduced **Correlation Rules** – a standardized way to compose detections that analyze relationships between events. Sigma correlation rules allow you to reference multiple base Sigma rules and define a condition on their combined occurrence. Unlike a normal rule's detection section, a correlation rule uses a correlation section with fields like rules (which lists the component rule names/IDs), a group-by key to tie events by common fields, a timespan to define the time window, and a special correlation condition.

For instance, imagine we want to detect a possible brute-force login attack. We could create a base rule for a failed login event, then a correlation rule that looks for a count of that event repeating many times for the same account. In Sigma correlation YAML, it might look like:

```
title: Multiple Failed Logons for a Single User (Brute Force)
correlation:
  type: event_count
  rules:
    - failed_logon     # reference to a base Sigma rule by name
  group-by:
    - TargetUserName
    - TargetDomainName
  timespan: 5m
  condition:
    gte: 10
```

In this example, type: event_count means we are counting occurrences of the referenced rule's events. The rule failed_logon (defined elsewhere as a Sigma rule matching EventID 4625, i.e. Windows failed login) is applied over a 5-minute window, grouping by username (and domain) – effectively counting failed logins per user. The condition gte: 10 triggers the correlation alert when **10 or more** failed logons for the same user occur within 5 minutes. The power of correlation rules is that the Sigma tooling will convert this into the appropriate query for your backend (for example, a Splunk query that uses a stats count by user over 5-minute bins and then filters count>=10). This turns what would have been a manual or platform-specific correlation search into a portable Sigma-defined detection.

Sigma currently supports several correlation types out of the box:

- **event_count:** Count of events in a time window (as above). Useful for threshold-based alerts like multiple failures, repeated malware beaconing, etc.
- **value_count:** Count of distinct values of a field in a time window. For example, if a single host communicates with an unusually high number of unique IP addresses in a short span (potentially indicative of port scanning or malware trying many C2 servers), a value_count correlation could catch that. An illustrated use-case is detecting Active Directory enumeration: one base rule detects any query for high-privilege groups, and a correlation rule triggers if a single user queries 4 or more different privileged groups in 15 minutes – a pattern that suggests a tool like BloodHound scanning for admin privileges.

- **temporal:** Checks if *different* event types occur close together in time. This is essentially a sequence or co-occurrence detection. For example, a temporal rule could watch for a *failed login* event followed by a *successful login* from the same source within, say, 1 minute – possibly indicating a successful credential spraying attack (the attacker finally hit the correct password). Another example: a "vulnerability exploit" correlation might trigger if a web server error log for a known exploit path is immediately followed by a new process on that server (the webshell launching a process), linking two different logs to catch exploitation. Temporal correlations don't necessarily count events, but rather flag when multiple specified patterns happen within the given timespan.

Correlation rules omit a logsource of their own – they rely on the log sources defined in the referenced base rules. When converting correlations, the Sigma engine will require those base rules to be present (often you include them in the same file, separated by ---). It's worth noting that correlation queries can become complex, and not all SIEMs support them yet. At the time of writing, only some backends (Splunk SPL, Elastic's EQL, Grafana Loki, etc.) fully support Sigma correlations natively. If your platform doesn't, you might still implement correlation by having Sigma rules feed alerts into the SIEM and then using the SIEM's native correlation capability. But as Sigma evolves, expect broader support for this powerful feature that lets you codify multi-step attack detection in a single, portable rule.

Contextual Enrichment of Alerts: Correlation is one form of context (the context of related events). Another equally important form is data enrichment – bringing in external or environmental data to improve detection accuracy and provide more information in alerts. Sigma rules are primarily about pattern matching in logs, and they remain *stateless* and *context-agnostic* by design (to stay generic). However, you can incorporate context in a few ways:

- *Field relationships:* Sometimes one log alone isn't enough to decide malicious vs benign. For example, a PowerShell execution event might be suspicious only if the calling user account is a normal user (not an admin) or if the machine isn't a known management server. If your log source includes such context (like a field for user role or host role), you can use that in the Sigma rule (e.g. add UserRole != "IT Administrator" as a condition). In many cases, though, such context isn't in the event itself. This is where your SIEM or pipeline can help by **enriching logs before detection** – for instance, tagging events with asset criticality, organizational unit, geolocation of IP addresses, etc. While Sigma doesn't query a CMDB or GeoIP database directly, you might feed enriched logs to Sigma detection. A rule could then simply check a boolean flag like asset_is_critical: true combined with another condition (ensuring you only alert on critical assets). Thus, when writing Sigma rules, consider what fields are available that provide context (user group, host tags, process reputation) and use them to refine logic.

- *Threat intelligence integration:* One powerful enrichment is to leverage threat intelligence feeds – lists of known malicious indicators (IP addresses, domain names, file hashes, etc.). For example, if you have a TI feed of known malicious IPs and your network logs tag any connection involving those IPs, your Sigma rule can just look for ThreatListHit: true or match the IP field against the list of bad IPs. Some organizations pre-filter logs by threat intel, but you can also do it within Sigma by using an **OR list of values** for the indicator field. For instance, a Sigma rule could detect outbound connections to any of a list of suspicious domains by specifying those domains in an array under the detection section. This works, though note that extremely large lists can

hurt performance. Typically, it's better to do TI matching in your ingestion pipeline or SIEM and then have Sigma rules react to a positive match. Regardless of implementation, the idea is to **enrich Sigma detections with threat intel context** so that you catch known badness. In fact, the Sigma team and community have recognized this need – future enhancements aim to allow more direct use of threat intel in rules. For example, AlphaSOC (a security analytics provider) mentions plans to *"utilize threat intelligence enrichment"* in Sigma rules, meaning **strengthening rules with real-time data like known malicious IPs or domains for greater precision**.

- *Joins and multi-source context:* Sometimes correlating an alert with a different data source can provide context post-detection. For example, if a Sigma rule detects a suspicious process on a host, you might automatically pull in recent authentication logs for that host or check if the file executed has a known malware hash. While this isn't done *inside* the Sigma rule, it's an operational practice: the alert generated by the Sigma rule can trigger additional queries or automated enrichment actions. From a Sigma rule engineering perspective, you design your rule to include key fields (like process hash, IP, user) so that those can be used as pivot points for enrichment. Some SIEMs support **alert enhancement** workflows (like attaching WHOIS info for an IP or pulling user details from Active Directory when an alert fires). As a rule author, be aware of these possibilities and ensure your rule's output (the log fields captured) will facilitate it. For instance, tagging your Sigma rule with the MITRE ATT&CK technique or a custom tag like needs_enrichment: true could signal your SOC playbooks to do additional processing.

In summary, contextual strategies in Sigma fall into two categories: (1) **Correlating events** – linking multiple log entries to detect patterns that single logs can't reveal; and (2) **Enriching events** – incorporating external knowledge (environment data, threat intel) to make detections smarter. Sigma's correlation rules address the first by allowing more complex multi-event logic in a portable way. For enrichment, you often rely on your security data stack (SIEM, SOAR, etc.) to integrate the context, but you design Sigma rules to leverage those enriched fields or to be ready for post-detection correlation. By applying these strategies, you significantly increase the signal-to-noise ratio of your Sigma-based detections – catching the subtle attacks that reveal themselves only through patterns or context, while providing analysts with the information they need (e.g. "this IP was flagged as malicious by threat intel") to respond decisively.

5.3 Testing, Debugging, and Quality Assurance

Developing Sigma rules is as much an engineering discipline as it is an analytical one. Just like software code, detection rules benefit from testing and quality assurance processes. In this section, we cover how to test Sigma rules to ensure they work as intended, how to debug issues when a rule doesn't behave as expected, and best practices for maintaining high-quality rule sets over time. The goal is to give junior and mid-level security engineers a roadmap for **Sigma rule QA** – so that when a rule is deployed to production, there are no unpleasant surprises (like floods of false alerts or, conversely, silent failures to detect).

Functional Testing of Rules: Before considering a Sigma rule "done," you should test it with both **expected malicious data and benign data**. This usually means obtaining or generating log samples. For a known threat or technique, gather some representative log lines (from a lab environment, public malware sandboxes, or SIEM history) that the rule *should* catch. Run your Sigma rule against this data to verify it indeed matches (and only at the right parts). Similarly, identify logs that are similar but benign, and confirm the rule does **not** match those. How do you

run a Sigma rule against data? If you have a SIEM or log management system, one way is to convert the Sigma rule to that platform's query and execute it on a dataset. The Sigma project provides tools to aid in this: for example, the **Sigma CLI** (sigma command) built on the *pySigma* library can translate Sigma rules into queries for many backends. PySigma is an official Python library for parsing and converting Sigma rules – essentially a successor to the older sigmac tool. You can install pySigma via pip and use it in scripts or use the CLI interface (which is a separate package, often called sigma-cli). For instance, if you want to test how your rule would run on Elasticsearch, you can do: sigma convert -t es-ql -r my_rule.yml (using the appropriate target flag and Sigma config). This yields the Elasticsearch Query Language query that corresponds to your rule, which you could then run against your Elastic indices containing test logs. Doing this allows you to catch errors early – perhaps the query returns no results on data where you expected a hit (meaning your rule logic might be too strict or field mappings are off), or it returns too many results (meaning the logic is too broad). Iteratively refine the Sigma rule, then reconvert and retest until the results align with expectations.

For a more automated approach, you can write unit tests for Sigma rules. Some organizations maintain a repository of Sigma rules and pair each rule with sample log events in JSON. They then use a simple script or CI pipeline to ensure that for each rule, the sample malicious events trigger a detection and the sample benign events do not. While Sigma itself doesn't include an execution engine to directly feed logs into rules, you can leverage your SIEM's query capabilities or use conversion plus a minimal search tool. In a pinch, you can even convert Sigma to a regex (some backends or tools can translate a Sigma rule to a generic regex which you can run on log lines). The key point is to **never deploy a rule blind** – always test it. The Sigma maintainers emphasize this as well: *"Test your rule before you commit them (we often see broken conditions)"*. A broken condition could be a logical mistake (like a typo in the condition combining selections), or simply a rule that doesn't match any data due to a mis-specified field. Testing catches these issues.

Using pySigma for Validation and Debugging: Beyond functional testing with log data, the **pySigma** toolkit provides a rule validation framework to catch errors and inconsistencies in rules. PySigma can validate the rule's syntax against the Sigma schema and even perform some semantic checks. In 2024, contributors introduced a JSON Schema for Sigma rules, which allows automated tools to verify that your rule YAML has all required fields and valid values. For example, it can catch if you used an invalid logsource category or if your detection section is malformed. The community even built a GitHub Action for Sigma rule validation – if you store your rules in a Git repository, you can have this action automatically check each new rule or change against the schema and known best practices. Embracing such automation is highly recommended for QA. It ensures things like your rule IDs are unique, the YAML is parsable, and fields like level, tags, falsepositives follow the expected formats. PySigma's validation module goes further by implementing checks for common pitfalls. For instance, one validator warns if you used a wildcard * where a Sigma modifier might be safer, or if your rule is overly broad without any condition to limit its scope (which could indicate a likely false-positive generator).

When a rule isn't working as intended, debugging it often involves **going back to the data and logic**. Check that the field names in your Sigma rule match those in your actual logs – mismatches are a frequent culprit (maybe your Windows Event uses TargetUserName but your log ingestion renames it to user.name; in Sigma you'd need to either use the processed name or apply a field mapping pipeline). PySigma supports custom *processing pipelines* to handle field name translation, which you can use during conversion to adjust your rule to your environment's

schema. If a rule is triggering too often (false positives), inspect some of the events it caught. Often this reveals a pattern that you can filter out with an additional condition, as discussed in section 5.1. If a rule never triggers, try removing one condition at a time to see if it starts matching (perhaps one of the conditions was too restrictive). Using Sigma's **modular detection** structure helps here: keep your detection logic organized in multiple selection and filter blocks, so you can more easily toggle pieces on/off during testing. For example, if you suspect the issue is with the command-line pattern, isolate that part and test just that selection on your data.

Another important aspect of QA is **peer review**. Treat Sigma rules like code: have a colleague review the logic. They might spot logic errors or suggest edge cases you didn't consider. Additionally, follow the community conventions for rule writing. SigmaHQ has published rule writing guidelines and conventions (covering naming, formatting, tagging, etc.). Adhering to these not only makes your rules cleaner, but in some cases avoids mistakes. For instance, one convention is to use lowercase for field names and values in the YAML (since Sigma is case-insensitive for matches by default) – deviating from that might not break a rule, but it could lead to confusion or conversion issues on certain backends.

Quality Assurance Checklist: To ensure a Sigma rule is production-ready, you can use a checklist like the following:

- **Correctness:** Does the rule detect the intended behavior? (Test with known malicious and known benign samples.) Does it avoid obvious false positives? (Did we include necessary exclusions/tuning?)
- **Clarity:** Is the rule logic as simple as it can be without losing meaning? If another engineer reads it, do the field names and condition make sense? (This is where good naming and comments help. Sigma allows comments in the YAML – use them to clarify non-obvious logic.)
- **Compliance with Schema:** Does the rule YAML pass schema validation? (All required fields present: title, id, description, author, date, etc., and formats correct.) Any schema linter or the Sigma validation tool should report zero errors.
- **Performance Considerations:** Review the rule for any expensive operations (regex, multiple wildcards). Are there opportunities to optimize as discussed in 5.1? Perhaps use endswith instead of a leading wildcard, or add an EventID to narrow the search. If the rule is intended for a high-volume log source, this step is crucial.
- **False Positive Assessment:** Check the falsepositives field in metadata – have you listed potential benign triggers? While this is optional, it's good practice for documentation. Moreover, if you listed false positive scenarios, double-check if the rule logic is **intentionally not** excluding them (maybe you chose to keep it broad but at least document the noise). Consider if any of those could be safely filtered out now via Sigma filters or conditions.
- **Tagging and Integration:** Ensure the rule is properly tagged (ATT&CK technique tags, log source taxonomy tags, etc.) so it integrates well with frameworks and can be found in searches. Also verify the rule's level and status (stable/experimental) are set appropriately given its maturity and reliability. A rule that's new or prone to false positives might start as status: test or experimental so it's clearly flagged.

Performing QA on detection rules might seem time-consuming, but it pays off immensely. A buggy or noisy rule in a production SIEM can erode trust in your detections and consume analysts' time. On the other hand, a well-tested rule that fires rarely and only for genuine threats is worth its weight in gold. In industry practice, vendors and large orgs often have a **QA team or process for detection content** – for example, Sigma rules contributed to the public repository are reviewed for quality, and companies like AlphaSOC subject custom Sigma rules to a thorough QA before deploying to their analytics engine. Adopting a similar mindset, even informally, will improve your detections. Use tools like pySigma and Sigma CLI to automate what you can (parsing, validating, converting), and use human insight for the nuanced parts (logic review, threat modeling).

Lastly, treat rule maintenance as an ongoing task. Threat techniques evolve, and so must your Sigma rules. Re-test rules periodically (especially after significant SIEM updates or changes in log formats). Keep an eye on Sigma project updates for new features – e.g. new modifiers or rule types – that could simplify or enhance your existing detections. By continuously testing and refining, you ensure your Sigma rules remain robust. **In essence, think of Sigma rule development as a cycle: design → implement → test → tune → repeat.** With solid testing and QA practices, this cycle will consistently produce high-quality detection content that you can rely on when the next incident strikes.

6. Real-World Detection Scenarios

6.1 Detecting Malicious PowerShell

Adversaries frequently abuse PowerShell to execute malicious code, often using obfuscation techniques to evade detection. One common tactic is passing a **Base64-encoded command** to the PowerShell -EncodedCommand (or -enc) switch. This allows attackers to hide scripts in an unreadable form on the command line. For example, an attacker might run PowerShell with flags to bypass execution policies, hide the window, and supply an encoded payload: powershell.exe -NoProfile -WindowStyle Hidden -EncodedCommand <Base64Payload>. Such usage corresponds to MITRE ATT&CK technique *T1059.001 – PowerShell*, often combined with *T1027 – Obfuscated Files or Information*. Detecting these patterns is critical, since legitimate administrators **rarely use** encoded commands or hidden PowerShell windows in daily operations.

Key Indicators of Malicious PowerShell: Attackers' PowerShell commands tend to exhibit certain patterns that defenders can hunt for:

- **Encoded or Compressed Commands:** Presence of the -EncodedCommand switch (e.g. -enc, -e) or use of System.Convert::FromBase64String to decode payloads in scripts.
- **Download and Execute Patterns:** Keywords like IEX (Invoke-Expression) downloading from URLs (e.g. .DownloadString or DownloadFile calls) which indicate **Living-off-the-Land** download of malware.
- **Execution Policy/AMSI Bypass:** Flags such as -ExecutionPolicy Bypass or code that disables AMSI (e.g. setting amsiInitFailed via .NET reflection) to evade antivirus.
- **Suspicious .NET APIs or Strings:** Use of .NET classes for low-level actions (e.g. System.Management.Automation.AmsiUtils or memory allocation APIs) or string concatenation tricks (like 'Invo'+'ke-WebRequest') to mask malicious intent.

Using **Sigma**, we can detect these PowerShell abuse patterns across Windows event logs (such as *Process Creation* events or PowerShell Script Block logs). Sigma rules are written in YAML and allow us to specify conditions on event fields. For instance, a Sigma rule to detect any use of an encoded PowerShell command might focus on process creation logs where the image is PowerShell and the command line contains variations of the encoded command switch. An example Sigma rule is shown below:

```
title: PowerShell -EncodedCommand Switch Detected
id: <GUID>
description: Detect PowerShell execution with an encoded command
(possible obfuscation)
tags: [attack.execution, attack.t1059.001, attack.defense_evasion,
attack.t1027]
logsource:
  category: process_creation
  product: windows
detection:
  selection:
    Image|endswith: '\powershell.exe'
```

```
    CommandLine|contains:
      - '-e '
      - '-enc '
      - '-EncodedCommand'
  condition: selection
falsepositives:
  - Rare (administrative scripts with encoded commands)
level: medium
```

This rule will trigger when **powershell.exe** is launched with any typical -EncodedCommand flag (the Sigma detection lists -e, -enc, etc., to cover shorthand forms). In practice, any Base64-encoded payload in the command line is a strong sign of obfuscation that merits investigation. Sigma's flexibility allows adding further filters to reduce false positives – for example, also checking if -WindowStyle Hidden or -NoProfile appears (since attackers often combine these). In fact, an open Sigma rule from the repository looks not only for the encoded flag but also for the string " hidden " in the command line and specific PowerShell image names. By using multiple conditions, we can ensure the detection is robust (e.g. the rule "Malicious Base64 Encoded PowerShell Keywords" requires that the process is PowerShell, the command includes the word *hidden*, and contains certain Base64 patterns indicative of malicious content).

It's important to leverage **PowerShell Script Block Logging** (Event ID 4104) where available, as it records de-obfuscated code at execution time. For instance, if an attacker uses encoded PowerShell to delete Volume Shadow Copies (a ransomware tactic), the script block log will show the actual code (e.g. Get-WmiObject Win32_Shadowcopy | ForEach-Object { $_.Delete(); }) even if the command line was encoded. Sigma rules can be written against these logs (with logsource set to Windows PowerShell logs) to catch suspicious script content like decoding routines, invocation of WebClient downloads, or calls to risky APIs. In summary, by combining **command-line keyword detection** (e.g. -EncodedCommand, IEX, DownloadString) with script block analysis, Sigma rules provide a powerful way to detect malicious PowerShell usage in real time.

6.2 Lateral Movement & Ransomware Kill-Chain

In this subsection, we explore how Sigma rules can detect adversary behaviors in two critical phases: **lateral movement** within a network and the **ransomware kill-chain** (the sequence of actions ransomware actors perform from gaining access to ultimately encrypting data). These activities correspond to multiple MITRE ATT&CK tactics – primarily *Lateral Movement*, *Credential Access*, and *Impact* (especially data encryption and destruction). We will examine real-world techniques like PsExec and RDP for lateral movement, registry dumping of credentials (SAM hive), and ransomware behaviors such as disabling backups (shadow copies) and executing file encryption tools. Each technique can be translated into Sigma detection logic with practical YAML rule examples.

Detecting Lateral Movement Techniques: Lateral movement refers to an attacker's methods of spreading to other systems after an initial breach. Common techniques include using stolen credentials to access remote services or executing tools on remote hosts. For example, **Remote Desktop Protocol (RDP)** access (MITRE *T1021.001 – Remote Services: RDP*) is frequently used by threat groups (from APTs to ransomware crews) to jump between machines. Similarly, tools like **PsExec** (a Sysinternals utility) enable execution of commands on remote Windows systems using SMB and service creation, and are widely abused by adversaries.

Notably, ransomware operators such as Ryuk, DarkSide, and others have leveraged PsExec to deploy ransomware across multiple computers. Another crucial step before or during lateral movement is **credential dumping** – for instance, dumping the Windows Security Account Manager (SAM) database to obtain password hashes (ATT&CK *T1003.002*). Armed with credentials, attackers can then use legitimate protocols (SMB, RDP, WMI, etc.) to move laterally with less chance of detection.

Sigma rules can detect these activities by looking for the tell-tale signs in logs:

- **PsExec Usage via Service Creation:** When PsExec is executed against a target machine, it creates a service named *PSEXESVC* on that host. This can be caught either by process creation events (the appearance of a PSEXESVC.exe process) or by Windows event logs indicating a new service installation. A Sigma rule focusing on Windows Event Log for system services might specify Event ID **7045** ("A service was installed") with the service name **PSEXESVC**, or Event ID 7036 (service state change) for PSEXESVC. For example, consider this Sigma detection logic derived from an open rule:

```
logsource:
  product: windows
  service: system    # System event log (services)
detection:
  service_installation:
    EventID: 7045
    ServiceName: 'PSEXESVC'
    ImagePath|endswith: '\PSEXESVC.exe'
  service_execution:
    EventID: 7036
    ServiceName: 'PSEXESVC'
  condition: service_installation or service_execution
level: high
```

This rule will alert on any event where the PSEXESVC service is installed or started on a host, which strongly indicates **PsExec-based remote execution**. In a real incident, such an alert suggests that a remote admin tool was used – analysts should then check which system initiated the PsExec and whether it was authorized. (A legitimate administrator might use PsExec, but it's uncommon in most environments and usually not from random endpoints.)

- **RDP Logon Abuse:** While interactive RDP usage can be legitimate, unusual patterns like a surge of RDP login attempts or an administrative account logging in via RDP from a non-standard system may indicate lateral movement. A Sigma rule to detect suspicious RDP usage could target Windows Security Event ID **4624** (logon successful) filtered by LogonType = 10 (which denotes a remote interactive logon, i.e. RDP) and possibly constrain by source IP or domain. For example, one could write conditions such as: EventID 4624 with LogonType 10 and **AccountName** = "Administrator" (or other high-privilege accounts) logging in from an unusual client address. This requires tuning to avoid false alarms (administrators do use RDP), but combined with known bad source IPs or odd hours, it can highlight brute-force or unauthorized RDP sessions. (In Sigma, logsource would be Security audit logs; detection might include TargetUserName: Administrator and LogonType: 10, etc., with appropriate condition).

- **SAM Registry Hive Dumping:** Capturing the SAM database via registry commands is a noisy but effective way to obtain credentials. The Windows reg.exe utility can save registry hives to disk. A known DarkSide ransomware technique is running reg.exe save HKLM\SAM sam.save to dump the SAM hive. Sigma detection for this involves looking at process command-line parameters. Specifically, if **reg.exe** is executed with arguments containing "HKLM\SAM" and a save (or export) operation, it's highly suspicious. Below is an example Sigma rule logic to detect SAM hive dumps:

```
logsource:
  category: process_creation
  product: windows
detection:
  selection_1:
    Image|endswith: '\reg.exe'
    CommandLine|contains:
      - 'save'
      - 'export'
  selection_2:
    CommandLine|contains:
      - 'hklm'
      - 'hkey_local_machine'
  selection_3:
    CommandLine|endswith:
      - '\sam'
      - '\system'
      - '\security'
  condition: selection_1 and selection_2 and selection_3
level: high
```

In this rule, all three conditions must be met: The process is reg.exe with a command line containing the keywords *save* or *export*, referencing the hive path (HKLM) and ending in one of the hive filenames (SAM, SYSTEM, or SECURITY). Legitimate uses of reg save for those hives are exceedingly rare, so this rule has a low false-positive rate. It directly addresses MITRE technique *T1003.002 (SAM)* and alerts security teams to possible credential dumping before the attacker can use those creds for lateral movement.

Detecting Ransomware Kill-Chain Behaviors: Once attackers have moved through the environment and possibly obtained higher privileges, ransomware operators execute a series of actions to **maximize damage** before encrypting files. This "kill-chain" often includes: disabling or deleting backups, shutting down security processes, and finally starting the encryption of data. Sigma rules can play a vital role in spotting these steps, ideally early enough to intervene.

A very common ransomware technique is *Inhibiting System Recovery* (MITRE *T1490*). This involves destroying system recovery artifacts like Volume Shadow Copies, backups, or restore points, so that the victim cannot easily recover their files. Several methods exist to do this on Windows, and ransomware groups use a variety:

- **Deleting Volume Shadow Copies:** This is typically done via vssadmin.exe or via WMI. For instance, the one-liner vssadmin.exe delete shadows /all /quiet will silently purge all shadow backups. Ransomware like **Ranzy Locker** and others use this, as do many older strains. Another way is using the native tool **DiskShadow** (with a script or direct

command delete shadows /all). Some ransomware families (e.g. Nefilim) even use **WMIC**: wmic shadowcopy delete /nointeractive achieves the same result via WMI. And as noted earlier, ransomware can leverage PowerShell to do this through WMI calls (NetWalker is known to run a PowerShell command to find and delete shadow copies via the Win32_ShadowCopy class). All these variations aim to wipe out Volume Shadow Copies (snapshot backups) so that even administrator recovery is hampered.

We can detect shadow copy deletion by monitoring processes that invoke these commands. A Sigma rule can take a **multi-selection approach** to cover all the above methods. For example, one could write a single rule with separate selections for each utility (vssadmin, diskshadow, wmic, PowerShell) and look for relevant keywords in the command line. An open Sigma rule (simplified here) for detecting **T1490 – Inhibit System Recovery** is as follows:

```
logsource:
  category: process_creation
  product: windows
detection:
  # Selection 1: Vssadmin/Diskshadow/WMIC/PowerShell deletion
  shadow_delete:
    Image|endswith:
      - '\vssadmin.exe'
      - '\diskshadow.exe'
      - '\wmic.exe'
      - '\powershell.exe'
    CommandLine|contains|all:
      - 'delete'
      - 'shadow'
  # Selection 2: Vssadmin resizing trick (Hakbit ransomware)
  shadow_resize:
    Image|endswith: '\vssadmin.exe'
    CommandLine|contains|all:
      - 'resize'
      - 'shadowstorage'
  # Selection 3: WBAdmin deletion of backups
  backup_delete:
    Image|endswith: '\wbadmin.exe'
    CommandLine|contains: 'delete'
  # Selection 4: Bcdedit commands disabling recovery
  no_recovery_bcd:
    Image|endswith: '\bcdedit.exe'
    CommandLine|contains: 'set'
    CommandLine|contains|all:
      - 'recoveryenabled'
      - 'No'
    CommandLine|contains|all:
      - 'bootstatuspolicy'
      - 'ignoreallfailures'
  condition: 1 of shadow_delete, shadow_resize, backup_delete,
no_recovery_bcd
level: high
```

This single Sigma rule checks for multiple related behaviors. It will fire if *any one* of the conditions is met (logic 1 of selection*): whether it's a vssadmin/diskshadow/PowerShell deletion of shadows, a vssadmin *resize* trick (which forces deletion by shrinking storage), usage of wbadmin.exe delete to remove backups, or bcdedit being used to disable recovery boot options. By casting a wide net, defenders can catch different ransomware families' techniques with a unified rule. Of course, each sub-technique individually is also a strong indicator – even a simpler Sigma rule looking just for vssadmin.exe with delete shadows would detect many ransomware attacks in progress. In fact, Sigma has specific rules for PowerShell-based shadow copy removal (detecting the use of Get-WmiObject … Win32_ShadowCopy together with a .Delete() call), since numerous ransomware families implement that approach. The key is that **any attempt to delete or tamper with backups and shadow copies on endpoints is rarely benign**, so these Sigma detections should be tuned with a high severity.

- **File Encryption Activity:** Detecting the actual encryption of files by ransomware is challenging, but there are indirect signs. Many ransomware binaries when launched will try to **disable security software** and system logs (Defense Evasion). For example, WastedLocker was observed using commands like taskkill /F /IM sfc.exe (to stop Windows integrity check) and MpCmdRun.exe -RemoveDefinitions -All (to disable Windows Defender definitions). Another ransomware added its process path to Defender's exclusion list via PowerShell (Add-MpPreference -ExclusionPath …). Sigma rules can catch these actions by monitoring process command lines or registry changes. In the SigmaHQ repository, for instance, there are rules to detect **registry modifications** that disable Windows Defender real-time protections by setting specific registry values to 1 (e.g. DisableRealtimeMonitoring). Those rules look at events like Sysmon ID 13 or Security ID 4657 (registry value set) and match the targeted keys under HKLM\Software\Policies\Microsoft\Windows Defender\… being set to disable protections. By catching this, defenders can spot ransomware attempting to **impair defenses** (MITRE *T1562*) as part of its kill-chain.

As for the encryption itself (MITRE *T1486 – Data Encrypted for Impact*), direct detection via logs is hard because the ransomware may simply open and rewrite many files, which at the OS level can look like a normal process doing file I/O. If the ransomware uses a known filename pattern for encrypted files or drops a ransom note file (e.g., README.txt with certain keywords), those can be hunted via SIEM. Sigma rules in some cases utilize indicators like known ransom note text or the creation of files with extensions like .locked or .crypt – but these tend to be specific to a ransomware family. A more generic approach is to detect abnormal file access patterns (which might require behavioral analytics beyond simple Sigma logic). However, by detecting the earlier steps (disabling recovery, killing security, etc.), we often catch the ransomware **before or during encryption**. In practice, a layered detection approach is used: Sigma rules for the obvious pre-encryption actions (like **shadow copy deletion, backup removal, service stopping**), coupled with anomaly detection for mass file modifications. Each piece of the ransomware kill-chain offers an opportunity to raise an alert.

In summary, Sigma rules allow us to codify the **real-world TTPs** of lateral movement and ransomware. By referencing open Sigma examples (as shown above) and MITRE ATT&CK mappings, defenders can write clear and specific detections: from a remote PsExec service launching on a host, to a registry being dumped, to system recovery tools being abused. The YAML examples illustrate how multiple conditions (like command-line keywords, process names, and event IDs) are combined to precisely target malicious behavior while avoiding

noise. These building blocks feed into SIEM alerts that can stop an adversary's progression *mid-attack* – for example, detecting a shadow copy deletion command might give responders a crucial window to isolate a machine before ransomware deploys widely.

6.3 Multi-Platform Case Study (Sigma → Splunk / Elastic / Sentinel)

One of Sigma's greatest strengths is its **universality**: a single Sigma rule can be written once and then used to detect a threat across multiple platforms and SIEM products. In this case study, we illustrate how a unified Sigma rule is created and then deployed to three different detection platforms – Splunk, Elastic, and Microsoft Sentinel – without changing the core logic. This approach is invaluable for organizations with heterogeneous environments, ensuring consistency in detections across tools. Instead of writing and maintaining separate queries for each SIEM (SPL for Splunk, KQL for Sentinel, etc.), the security team maintains **one Sigma rule** as the source of truth.

Let's walk through the process with an example. Suppose we want to detect the use of PsExec for lateral movement (as discussed in section 6.2). We have already formulated a Sigma rule in YAML to catch the creation of the PSEXESVC service via Windows events. Now, consider an enterprise where Windows logs are flowing into Splunk (on-premises), Elastic Security, and Azure Sentinel simultaneously (perhaps different business units use different tools, or the enterprise is migrating between them). We will see how the **same Sigma rule** can be utilized in all three:

- **Writing the Sigma Rule:** We start by defining the Sigma rule abstractly, independent of any SIEM query language. For PsExec detection, the rule's logsource is Windows system events, and the detection section has the conditions for EventIDs 7045/7036 with ServiceName "PSEXESVC" (as shown earlier). We ensure that the field names we use are Sigma's standard field names (EventID, ServiceName, etc.) which are mapped to the appropriate fields in each target platform by Sigma's backend converters. The Sigma rule also includes metadata like title, description, tags (including MITRE technique tags), and a unique ID. At this point, the rule is **platform-agnostic** – it's essentially a human-readable and tool-parseable representation of the detection logic.
- **Converting to Splunk Query:** Using a Sigma-to-Splunk converter (which could be an open-source tool like sigmac or a SIEM integration), the YAML rule is transformed into a Splunk Search Processing Language (SPL) query. The converter knows that logsource: product: windows, service: system corresponds to searching in the Windows Event Log index (e.g., index=wineventlog SourceName=Service Control Manager in Splunk, since Event 7045 comes from the Service Control Manager). It also maps Sigma field EventID to the Splunk field (often called EventCode or simply a field in the event) and crafts the condition for ServiceName. The resulting SPL might look like:

  ```
  index=wineventlog EventCode=7045 Service_Name="PSEXESVC" OR
  (EventCode=7036 Service_Name="PSEXESVC")
  ```

  (We won't delve into exact syntax, but effectively the Sigma rule produces a query that Splunk can run to find those events.) The crucial part is that we did **not** write this query by hand – it was generated from the Sigma rule, ensuring that any update to the rule logic in YAML can be propagated to Splunk easily.

- **Converting to Elastic (ELK/Elastic Security):** Similarly, the Sigma rule can be converted to a query for Elastic Stack. Depending on the Elastic usage, this could be an Elasticsearch Query DSL, an EQL (Elastic Query Language) rule, or a KQL (Kibana Query Language) query for detection engine. The converter will map the same logsource to the Elastic indices (for instance, Winlogbeat indices that contain Windows event data). In Elastic Common Schema (ECS), Windows EventID might appear as event.code or winlog.event_id, and ServiceName could be something like winlog.event_data.ServiceName. The Sigma converter handles these differences. The resulting Elastic query will retrieve events where winlog.event_id:7045 and winlog.event_data.ServiceName:"PSEXESVC" (or the equivalent in the chosen query language). Again, no manual coding of this query is needed – the logic comes straight from the Sigma rule.

- **Converting to Azure Sentinel (Microsoft Sentinel):** Azure Sentinel uses Kusto Query Language (KQL) to query logs (often through the Log Analytics workspace). Windows event logs in Sentinel reside in a table typically named **SecurityEvent** or **Event**, with columns like *EventID*, *ServiceName*, etc. The Sigma rule, when converted, would yield a KQL query along the lines of:

  ```
  SecurityEvent
  | where EventID in (7045, 7036)
  | where ServiceName == "PSEXESVC"
  ```

  (plus appropriate filtering for the time range, etc., when operationalized). Once more, the key field names and values come from our Sigma rule. The converter ensures that, for instance, ServiceFileName|endswith '\PSEXESVC.exe' condition is either dropped or translated appropriately if that detail isn't available in Sentinel logs (since SecurityEvent 7045 actually provides the service file name in a field which might or might not be directly accessible – if it is, the converter would include it).

- **Deploying and Using the Rule:** With the queries for each platform generated, security teams can then integrate them into their respective SIEM detection engines. In Splunk, the query can be saved as a scheduled search or an alert. In Elastic Security, it can be loaded as a detection rule. In Sentinel, it becomes a custom analytic rule. The beauty is that **all three detections are logically identical** – they will alert on the same event conditions in their datasets. This ensures that no matter where the Windows logs reside, the malicious PsExec event is caught. It also simplifies maintenance: if we decide to broaden the rule (say, include detection for **PAExec**– a PsExec clone – which might use a service name like "PAExecSvc"), we simply edit the Sigma YAML (e.g. add "PAExec*" in the ServiceName list) and re-run the converters. Each SIEM's query gets updated with the new logic, keeping parity across tools.

This multi-platform applicability extends beyond just this example. Sigma is designed to be "**a straightforward manner... applicable to any type of log file**". In practice, this means a well-written Sigma rule for a given use case (malicious PowerShell, lateral movement, etc.) can be a single source that drives detection in numerous environments: from traditional SIEMs like Splunk/QRadar, to modern data lakes and cloud-native platforms. Many organizations use Sigma as a **unifying detection language** – analysts write rules in Sigma and use automation to deploy them to different monitoring systems. This avoids the scenario of having siloed detection

logic (one team writing Splunk queries, another writing Sentinel queries for the same threat). It also reduces errors, since the logic is defined once and consistently applied.

In summary, this case study demonstrates that a unified Sigma rule can cover a multi-platform deployment by abstracting the detection logic from any specific query language. We maintained focus on Sigma itself – the YAML rule capturing the essence of the threat – and relied on Sigma's tooling to handle translation to each backend. The result is **consistent, transparent, and efficient detection coverage**. A junior analyst can read the Sigma rule in plain language and understand what it looks for (e.g., "service installation of PSEXESVC"), and a senior engineer can trust that this logic is reliably implemented in Splunk, Elastic, and Sentinel without manually writing three separate rules. This approach underscores why Sigma is often called a *"universal detection rule language"* – it enables security teams to speak one language while listening on many platforms.

Chapter 7: Future Evolution and Community Collaboration

In this final section, we turn our attention to the ongoing evolution of the Sigma project and the vital role of the community in its growth. Sigma's journey doesn't end with mastering the basics or integrating rules into a SIEM; in fact, that's just the beginning. The strength of Sigma lies in its adaptability and the collective effort of security professionals worldwide. We will explore how Sigma continues to expand with new back-ends, modifiers, and rule types, how you can contribute to the SigmaHQ community, and what resources and learning paths can help you deepen your expertise. Throughout, the emphasis is on community engagement, continuous learning, and giving back to the shared body of detection knowledge.

7.1 Upcoming Back-Ends, Modifiers, and Rule Types

Sigma is a living project – a framework designed to grow and adapt as the security landscape changes. One area of constant growth is the **development of new back-ends**. Back-ends are the translation modules that convert Sigma's generic detections into platform-specific queries for various SIEMs and analytic tools. As new log management platforms, cloud services, and endpoint solutions emerge, the Sigma community responds by creating back-ends to support them. The introduction of **pySigma**, a reengineered Sigma converter, has made this process more flexible: back-ends and processing pipelines are now maintained separately from the core, allowing vendors and community members to develop and update their own converters independently or in collaboration with SigmaHQ. This means Sigma can swiftly accommodate emerging technologies – if a new cloud SIEM or an innovative analytics tool gains popularity, one can expect a Sigma back-end for it to appear so that detection rules remain truly universal. The message is clear: Sigma's compatibility horizon is always expanding.

Equally important is the evolution of Sigma's rule language itself. Over time, Sigma's maintainers have introduced new **modifiers** and rule constructs to express more sophisticated detection logic. *Modifiers* in Sigma are like built-in functions or operators that refine how a match is performed (for example, matching substrings, regex patterns, byte sequences, or numeric comparisons). Introduced in 2019, Sigma modifiers gave detection engineers the power to perform more complex operations on log data within rules. These operations include regular expressions, Boolean logic combinations, CIDR notation for IP ranges, and many others – capabilities that align Sigma with what most SIEM query languages support. The set of available modifiers has grown to cover common needs (such as case-insensitive matching or base64 decoding of strings), and it will continue to grow as new use-cases emerge. We can anticipate that future modifiers will be added to handle specialized patterns or data transformations, ensuring that Sigma rules can adapt to novel attacker behaviors or logging quirks. The goal is that no matter how an attacker tries to hide their activity in logs – be it through encoding, obfuscation, or unconventional data formats – Sigma's language will eventually have the constructs to detect it.

Another significant leap in Sigma's evolution is the introduction of new **rule types** and advanced rule capabilities. Initially, Sigma rules focused on single events or simple patterns in isolation. However, modern attack detection often requires linking multiple events or applying contextual logic. In response, Sigma has extended its format with *Meta rules*, which enable advanced techniques like **correlation rules** and **filter rules**. **Correlation rules** allow analysts to define

detections that span multiple events – for example, detecting a sequence or count of related events that together indicate a threat pattern, rather than a one-off log entry. Sigma correlation rules make it possible to express these multi-event relationships within the Sigma format, enabling more sophisticated and targeted detections by combining and analyzing relationships between events. Similarly, **filter rules** help in tuning detections by filtering out known noise or common false positives, referencing other rules and systematically excluding benign events. The addition of correlations and filters to Sigma is a prime example of how the rule taxonomy is growing to meet real-world needs. Going forward, the Sigma community is likely to continue exploring new rule types or schema improvements – perhaps enabling easier anomaly detection logic, stateful patterns, or integration with other frameworks – all while keeping the core philosophy of Sigma: a universal, shareable detection language. Importantly, any new feature in Sigma is designed with broad applicability in mind. By staying general and vendor-neutral, Sigma's evolution ensures that your detection logic can keep pace with advancements in both attack techniques and defense tools without locking you into a single platform.

For junior and mid-level security professionals, the takeaway is that Sigma will not stand still. To remain effective, you should keep an eye on Sigma's updates (be it new back-end support, language features, or rule formats) and be prepared to incorporate them into your detection practice. Embracing the evolving capabilities of Sigma means your arsenal of detection techniques will remain sharp and up-to-date. In essence, Sigma's ongoing enhancements – new back-ends connecting to an ever-wider array of systems, new modifiers making the language more expressive, and new rule types capturing complex attack patterns – are all about one thing: **staying ahead of adversaries by continuously improving how we describe and share detections**.

7.2 Contributing to SigmaHQ

One of the greatest strengths of Sigma is not just its technology, but its **community**. Sigma was born from the security community's need for a common detection language, and it has thrived because of contributions from analysts and engineers around the world. The official SigmaHQ rule repository is a testament to this collective effort – it hosts thousands of detection rules (over 3,000 at the time of writing) covering a vast range of threats, all shared freely to benefit defenders everywhere. These rules aren't created by any single vendor or small team; they are written and curated by a broad community of practitioners, from seasoned threat hunters to passionate newcomers. Every rule that is submitted goes through peer review by other detection engineers, which means the repository's content is continuously refined and vetted by experts. This collaborative model helps maintain a high standard of quality and reliability. In fact, SigmaHQ openly attributes its success to community contributions: the project "would've never reached this height without the help of the hundreds of contributors" who have lent their time and expertise.

How can you, as a security professional, become a part of this ecosystem? There are many ways to contribute to SigmaHQ and the broader Sigma community, even if you consider yourself junior or mid-level. The most direct way is by **writing Sigma rules** and sharing them. Whenever you create a new detection for a threat in your environment – perhaps you developed a Sigma rule to catch a novel phishing technique or a lateral movement trick – consider contributing it back. By submitting your rule to the SigmaHQ repository, you allow others to benefit from your insight, and you'll often receive feedback to improve the rule (which in turn improves your own skills). Writing a good Sigma rule for community use involves adhering to the common

conventions and ensuring the rule is as general (environment-agnostic) as possible. SigmaHQ provides guidelines and a contribution checklist to help authors align with the expected format and quality, but the process is welcoming – even first-time contributors are encouraged, as long as they are willing to learn and iterate.

Contributing to Sigma isn't limited to just adding new rules. You can also **improve existing rules** by reporting false positives or suggesting enhancements. If you run Sigma rules in your SOC and notice a rule is overly broad or not working as intended, you can engage with the maintainers (for example, by opening an issue on the GitHub repository) to discuss fixes. This kind of feedback is incredibly valuable: it helps fine-tune detections and ensures that rules stay relevant as software and attack patterns change. In many cases, you might propose a tweak or an additional condition that reduces noise in a rule – a small change that ends up benefiting every organization that uses that Sigma rule.

Beyond rules, more technically advanced practitioners might contribute to Sigma's **tooling and back-ends**. As described in the previous subsection, Sigma's architecture allows for new back-end modules or improvements to the Sigma codebase. If you have programming skills and use a log platform that isn't supported yet, you could write a converter (or "back-end") for it and share it with SigmaHQ. There is a history of community members doing exactly this – contributing back-ends for platforms they're familiar with. Likewise, improving documentation, writing translations, or adding examples are all forms of contribution. Even creating educational content (like blog posts, tutorials, or Sigma rule walkthroughs) and sharing those with the community helps spread knowledge and drive more adoption.

The key point is that **Sigma is a community-driven project**, and it thrives when professionals like you get involved. By contributing, you not only help others, but you also accelerate your own learning. You'll interact with experienced detection engineers, learn best practices, and stay on top of emerging threats (since writing or reviewing rules naturally keeps you informed about attacker behaviors). SigmaHQ provides multiple channels for community engagement – from the GitHub repository discussions to community chat rooms and forums (for example, Sigma has an active Discord and other groups). Don't hesitate to join these communities. Even asking questions or sharing how you used a Sigma rule in a creative way can spark discussions that improve the ecosystem.

Finally, contributing back to Sigma is a way of **giving back to the shared defense knowledge**. Remember that most likely many of the detections protecting your organization come from open Sigma rules that someone else contributed. By adding your own insights, you're paying it forward and ensuring that collective defenses stay strong. The culture of Sigma is one of openness and collaboration: the more people contribute, the more robust the catalog of detection rules becomes, and the harder it is for adversaries to hide in logs unseen. Whether you add a single rule or become a regular contributor, your participation matters. As SigmaHQ's maintainers often acknowledge, every bit of community effort counts in making Sigma the universal language for detection that it is today.

7.3 Further Reading, Labs, and Training Paths

Mastering Sigma and detection engineering is a journey, not a destination. As you finish this book, it's important to continue building on this foundation through ongoing learning and hands-on practice. Fortunately, there is a wealth of **resources, labs, and training paths** available –

formal and informal – to help you deepen your skills in using Sigma and developing effective detection logic.

Official Documentation and Knowledge Bases: A good starting point is always the official Sigma documentation and related knowledge bases. The Sigma project's website (SigmaHQ) maintains up-to-date documentation on the rule syntax, specification changes, and examples. Exploring the Sigma documentation beyond what we covered here can reinforce your understanding and introduce advanced topics (for instance, the Sigma docs detail the full list of modifiers, additional examples of correlation rules, and so on). The official repository's README and Wiki also contain useful references and links to talks or papers. It's wise to keep an eye on SigmaHQ's blog or announcement pages for news on new releases. For example, new rule package releases are periodically announced with highlights of what threats have been recently addressed by the community. Reading these updates can give you a sense of trending threat detection topics and how the Sigma language or repository is evolving in practice.

Community Publications and Research: The Sigma community and the broader security industry frequently publish articles and research that can enhance your detection engineering prowess. Many **blog posts, whitepapers, and conference presentations** focus on real-world detection challenges and how Sigma rules can be applied. For instance, you'll find write-ups by threat hunters describing how they translated an incident or threat report into Sigma rules, often sharing tips on rule tuning. Incident analysis reports (such as those on well-known DFIR blogs) often include Sigma rules as detection recommendations, which serve as excellent study material – they show how expert analysts think in terms of log patterns. Additionally, Sigma has been a topic at security conferences and webinars. Talks like *"Generic Signatures for Log Events"* by Sigma's creators and webcasts tying Sigma rules to frameworks like MITRE ATT&CK have been recorded and shared. These are valuable to watch or read because they provide insight into Sigma's design philosophy and advanced use cases. Engaging with such content keeps you up-to-date on best practices and creative techniques in writing rules.

Hands-On Labs and Exercises: Theory and reading are important, but there is no substitute for practice. To build confidence, seek out labs and simulated environments where you can apply Sigma in a safe, controlled setting. Many online platforms and community events offer **threat hunting and detection labs** that are perfect for this. For example, capture-the-flag (CTF) challenges or workshops focused on blue-team skills often include scenarios where you must analyze logs and find malicious activity – an ideal opportunity to write or use Sigma rules to detect the clues. Some events (hosted by security companies or community groups) provide participants with log datasets and ask them to create detections, which mirrors the process of developing Sigma rules. Even outside of structured events, you can practice on your own: set up a home lab with a SIEM (or use a cloud logging service), ingest some open-source attack data or audit logs, and try to craft Sigma rules to catch certain behaviors. This kind of hands-on experimentation will solidify your understanding of how Sigma rules translate to actual alerts. It will also teach you how to adjust a rule when faced with data noise or slight variations in log formats – a skill that every detection engineer needs in the field.

Structured Training and Certification Paths: As Sigma has grown in prominence, it has found its way into the curriculum of various cybersecurity training programs. If you learn best through structured courses, consider pursuing training in threat detection, security monitoring, or threat hunting that includes Sigma as a component. For instance, some well-known cybersecurity education providers offer courses on detection engineering or SIEM content development,

where Sigma is used as a teaching tool due to its universal applicability. These courses can provide a comprehensive path – starting from writing simple rules to integrating those rules into workflows, automating their deployment, and measuring their effectiveness. Additionally, professional certifications in threat hunting or security monitoring can be valuable; while they may not focus exclusively on Sigma, the concepts of writing effective detections and understanding log data that you've learned here will give you a strong advantage. When evaluating training options, look for those that emphasize practical detection exercises and mention modern, tool-agnostic approaches (which is exactly where Sigma shines).

Lifelong Learning Mindset: No matter which resources you choose, the most important thing is to maintain a mindset of continuous learning. Cyber threats are continuously evolving, and so are defensive techniques. By staying curious and engaged – reading the latest research, playing with new Sigma features, practicing in labs, and collaborating with peers – you ensure that your skills remain relevant. Becoming adept with Sigma also positions you at an interesting intersection of the security community; you'll be conversant in a "lingua franca" of threat detection that many professionals and organizations are adopting. This can open up opportunities to participate in wider community projects or even mentor others down the line.

In conclusion, this chapter (and the book) may be wrapping up, but your journey with Sigma is just opening into a new phase. **The evolving nature of Sigma and the ever-growing community around it mean that there will always be something new to learn, a rule to write or refine, and a chance to contribute.** By leveraging further readings, engaging in hands-on labs, and following a path of continuous training, you will not only improve your own capabilities but also help advance the state of the art in detection engineering. The Sigma project exemplifies how a community of practitioners can come together to share knowledge and tools for the common good. Now, as you step forward, you are encouraged to be an active part of this story – to innovate, to share, and to keep pushing the boundaries of what we can detect and defend against. Happy hunting, and welcome to the ongoing Sigma journey.

Bibliography

- https://graylog.org/post/the-ultimate-guide-to-sigma-rules/
- https://attack.mitre.org/techniques/T1021/001/
- https://blog.sigmahq.io/connecting-sigma-rule-sets-to-your-environment-with-processing-pipelines-4ee1bd577070
- https://blog.sigmahq.io/how-to-validate-sigma-rules-with-github-actions-for-improved-security-monitoring-3d23a23803ff
- https://blog.sigmahq.io/introducing-sigma-filters-204bd896273
- https://blog.sigmahq.io/introducing-sigma-specification-v2-0-25f81a926ff0
- https://blog.sigmahq.io/sigmahq-rules-release-highlights-r2024-03-26-50b086b2f540
- https://blog.sigmahq.io/sigmahq-rules-release-highlights-r2024-05-13-237ed77459bf
- https://blog.sigmahq.io/sigma-rule-repository-enhancements-new-folder-structure-rule-types-30adb70f5e10
- https://cardinalops.com/blog/splunk-and-other-siem-detections-for-follina/
- https://community.opentext.com/cybersec/threat-detect-response/f/discussions/105445/sigma-rules-guide-threat-hunting-for-esm-arcsight-command-center-and-logger
- https://cymulate.com/blog/cymulates-sigma-rules/
- https://detection.fyi/mbabinski/sigma-rules/2024_redcanary_threatdetectionreport/technique_powershell_encoded_command/
- https://detection.fyi/sigmahq/sigma/windows/file/file_event/file_event_win_lsass_default_dump_file_names/
- https://detection.fyi/sigmahq/sigma/windows/powershell/powershell_script/posh_ps_potential_invoke_mimikatz/
- https://detection.fyi/sigmahq/sigma/windows/process_creation/proc_creation_win_msra_process_injection/
- https://detection.fyi/sigmahq/sigma/windows/process_creation/proc_creation_win_powershell_base64_hidden_flag/
- https://detection.fyi/sigmahq/sigma/windows/process_creation/proc_creation_win_sysinternals_procdump_lsass/
- https://docs.alphasoc.com/detections_and_findings/sigma_custom/
- https://enescayvarli.medium.com/sigma-rules-to-splunk-from-detection-logic-to-real-time-alerts-c1c8900ca660
- https://github.com/SigmaHQ/pySigma
- https://github.com/SigmaHQ/sigma
- https://github.com/SigmaHQ/sigma/wiki/Rule-Creation-Guide/97c55ea762e775787c1f38a623a8ae12de94253b
- https://graylog.org/post/sigma-specification-2-0-what-you-need-to-know/
- https://marketplace.visualstudio.com/items?itemName=humpalum.sigma
- https://medium.com/@Architekt.exe/writing-your-first-sigma-rule-5ed783c87570
- https://medium.com/@balasubramanya.c/advanced-guide-to-sigma-rules-for-elastic-siem-web-server-log-analysis-3fd595c9e74d
- https://medium.com/@imanvanpersien/mitre-att-ck-framework-as-a-standard-for-developing-siem-use-cases-d7dc7db4e1ba

- https://micahbabinski.medium.com/creating-a-sigma-backend-for-fun-and-no-profit-ed16d20da142
- https://pypi.org/project/pySigma/
- https://readsecurity.medium.com/powerful-powershell-commands-to-monitor-for-attack-detections-2397b4f154d1
- https://securitybeztabu.pl/sigma-uniwersalny-jezyk-regul-detekcji-czesc-1/
- https://securitybeztabu.pl/sigma-uniwersalny-jezyk-regul-detekcji-czesc-2/
- https://sigmahq.io/docs/basics/conditions.html
- https://sigmahq.io/docs/basics/log-sources.html
- https://sigmahq.io/docs/basics/modifiers.html
- https://sigmahq.io/docs/basics/rules.html
- https://sigmahq.io/docs/digging-deeper/backends
- https://sigmahq.io/docs/guide/about
- https://sigmahq.io/docs/guide/getting-started.html
- https://sigmahq.io/docs/meta/
- https://socprime.com/blog/uncoder-ai-automates-mitre-attck-tagging-in-sigma-rules/
- https://www.logpoint.com/en/blog/how-to-use-logpoint-pysigma-backend-for-threat-detection/
- https://www.nextron-systems.com/2018/02/10/write-sigma-rules/
- https://www.picussecurity.com/resource/detection-and-prevention-in-the-late-phase-of-the-ransomware-attacks
- https://www.reddit.com/r/QRadar/comments/13179z9/ho_to_fix_field_does_not_exist_in_catalog_events/