THOMAS MORE

# Cegeka Internship

Realization document

Jarne Willems
Student Bachelor Applied Computer Science

2024 - 2025

# Table of Contents

# 1. Introduction

In February 2025, I began my internship at Cegeka, **an international, family-owned IT solutions and services provider headquartered in Hasselt, Belgium.** Cegeka offers a wide range of services, including cloud computing, cybersecurity, software development, and more.

I completed my internship within the **SOC (Security Operations Center)** department, publicly known as C-SOR²C (Cyber Security Operations and Response Centre) now rebranded as the Cegeka Modern SOC. Specifically, I was part of the Detection Engineering team.

**The goal of this team is to optimize existing detection logic and develop new detections**, working closely with other SOC teams such as SOC Analysts and SOC Engineers.

In this document, I will outline the **thought process, findings, and realizations** I made during my internship. For more detailed information about the internship assignment itself, I refer to my 'Plan of Action' document, found in my graduation portfolio. Here I explain the assignment's objectives and the steps I took to complete it.

I divided my internship into four main phases. For each phase, I will describe in detail what it entailed and share my findings and reflections.

The document begins with the **analysis phase**, where I explain the steps I took, starting from the preparation period before the internship officially started and leading up to **the realization phase**.

In the realization section, I describe the final results of my work. This part is shorter than the analysis section, as the main focus of my internship was on analysis.

Finally, I conclude with a reflection on my overall internship experience, summarizing the key lessons learned.

# 2. Analysis

In this chapter, I will describe the analysis I conducted as part of my internship assignment. Since the majority of my work revolved around analysis, this chapter will be quite extensive. I will walk through my thought process, the tools I used (and why), and conclude with some weighted rankings.

## 2.1. Initial research before the internship

Before starting my internship, it was already clear that I would be working with two SIEM solutions: **Microsoft Sentinel and Splunk**. At that point, I had no prior experience with either tool, so I took the initiative to do some research in advance. Given that my main tasks would involve analyzing, writing, and modifying detection logic, I focused my research on the query languages used by both platforms, **Splunk's SPL (Search Processing Language) and Microsoft Sentinel's KQL (Kusto Query Language**). While both are designed for querying large datasets, they have distinct characteristics:

| **Feature** | **Splunk – SPL** | **Sentinel – KQL** |
|---|---|---|
| **Type** | Proprietary language developed by Splunk | Microsoft's language used across Azure (Log Analytics, Application Insights, etc.) |
| **Syntax Style** | Unix-style pipeline-based commands | SQL-like declarative syntax |
| **Ease of Use** | More flexible but less intuitive | Easier for users familiar with SQL; better readability |
| **Learning Curve** | Medium to high (especially for complex queries) | Moderate (more intuitive for analysts with SQL background) |

From this comparison, I understood that learning both query languages would be crucial, as knowledge in one doesn't easily transfer to the other.

With that in mind, I began learning the languages. However, I encountered an early challenge: **while Splunk offers more flexibility** in terms of access (being available both on-premises and in the cloud), **Microsoft Sentinel is strictly cloud-based** and requires an active Azure subscription.

To learn SPL, I chose a method that had worked well for me in the past: hands-on practice through TryHackMe labs. These exercises allowed me to interact with real Splunk environments, which gave me practical insights into the structure of detection queries and commonly used commands experience that proved invaluable when my internship began.

For Microsoft Sentinel, the approach was different. My internship mentor, Lander, recommended the Microsoft Learn platform, **specifically the "Ninja Training" modules.** These were well-structured and rich in theoretical content, making them a solid foundation for understanding KQL and the workings of Sentinel.

However, applying this theory proved challenging. Since Microsoft Sentinel only runs in a cloud environment, and I had limited Azure credits through my student account (which I also used for school projects), I was unable to perform many of the practical exercises. Given the high costs associated with running Sentinel, I had to rely primarily on the theoretical materials provided in the training modules.

Despite this limitation, I was able to gain a foundational understanding of both platforms and their respective query languages, which definitely gave me a head start once the internship commenced.

## 2.2. Start of the internship

Once my internship began, the objective of my assignment became clear very quickly. **Within the first two weeks**, I had already created a comprehensive plan of action (which can be found in my graduation portfolio). I highly recommend reviewing that document as it outlines in detail the goals of my internship assignment and the approach I took to accomplish them.

The core objective of my assignment was to help Cegeka achieve consistent and equivalent detection coverage across both Splunk and Microsoft Sentinel. Cegeka uses both SIEM platforms, depending on client preference some customers prefer Splunk, while the majority opt for Sentinel.

However, the detection rules in place weren’t unified across the platforms. Some rules existed only in Sentinel, others only in Splunk. In certain cases, there were rules with similar purposes present in both platforms, intended to detect the same type of threat or event.

But prior to my internship, it was unclear:

- **Which detection rules were exclusive to either Splunk or Sentinel?**
- **Which rules overlapped between both platforms?**
- **In cases of overlap, how similar or different were the detection logics?**
- **Did one rule take a broader or more specific approach than its counterpart?**

These unanswered questions formed the basis of my assignment and the starting point of my analysis.

It quickly became evident that this assignment could be broken down into several logical phases, each building upon the previous to ensure a structured and thorough approach:

**Phase One: Inventory of all Detection Rules**
In the first phase, the goal was to identify which rules were present in both platforms, and which were exclusive to either environment. This laid the groundwork for the comparative analysis in the following phases.

**Phase Two & Three: Rule Comparison and Harmonization**
In the second phase, I focused on comparing the overlapping rules between the two environments. I designed a structured approach to systematically identify and highlight differences in the detection logic

The third phase built upon this by creating a standard procedure to align these rules. The aim was to ensure both platforms would offer the same level of detection coverage, closing any gaps and resolving inconsistencies.

**Phase Four: Documentation**
The final phase was dedicated to documentation. This served a dual purpose: contributing to my bachelor's thesis and providing clear internal documentation for Cegeka. By documenting the entire process, the team now has a solid reference point for any future projects involving detection rule analysis and alignment.

| Date | Phase | Description |
|---|---|---|
| 17-Mar | **Phase 1** | Inventory Of all Detection Rules |
| 24-Mar | **Phase 1** | Inventory Of all Detection Rules |
| 31-Mar | **Phase 1** | Inventory Of all Detection Rules |
| 07-Apr | **Phase 2** | Rule Comparison |
| 14-Apr | **Phase 2** | Rule Comparison |
| 21-Apr | **Phase 3** | Rule Harmonization |
| 28-Apr | **Phase 3** | Rule Harmonization |
| 05-May | **Phase 3** | Rule Harmonization |
| 12-May | **Phase 4** | Documentation |
| 19-May | **Phase 4** | Documentation |
| 26-May | **Phase 4** | Documentation |

## 2.3. Phase One: Inventory of all detection rules

After creating a thorough plan of action, it was time to initiate Phase One of my assignment: creating an inventory of the detection rules and categorizing them by exclusivity and overlap.

Cegeka has a use-case library that includes all detection rules from every customer environment (including deactivated ones). This library was provided as a Power BI file, which could be exported to Excel, a solid starting point for my analysis.

Immediately, several questions arose:

- **The file is massive, containing 424 lines of rules. What is the most effective way to perform the analysis?**
- **What criteria should I use to determine if rules are the same? Name, description, or something else?**
- **If I find overlapping rules, how can I ensure they are truly identical?**

There were two main approaches I considered for this analysis: writing an automation script (using Python, for example) or manually identifying overlaps using Excel's built-in filtering features.

| Approach | Time Spent (20%) | Effectiveness (40%) | Reusability (15%) | Scalability (15%) | Ease of Implementation (10%) | Total Score |
|---|---|---|---|---|---|---|
| **Manual Matching** | **4** | **5** | **3** | **1** | **5** | 3.90 |
| **Automated Matching** | **5** | **2** | **4** | **4** | **3** | 3.30 |

I based this **Weighted Ranking Method (WRM)** primarily on Time Spent and Effectiveness. To get a clearer picture of which approach would work best, I first analyzed some rules in the document and noticed that many overlapping rules had completely different names or used different phrasing to describe the same logic.

**An automation script could work using fuzzy matching**, but only if the rule names were somewhat similar which, in most cases, they weren't. This would mean that even after spending time writing and running a script, I'd still have to manually check for missed overlaps. At that point, it would be more effective to go through the document manually from the start.

So, **I chose the manual approach**. It turned out to be effective, as I could clearly mark overlapping rules and continue my search. What I discovered (and this also answered my third question) is that the only reliable way to confirm whether rules are truly the same is by comparing their detection logic, which is the goal of Phase Two, not Phase One.

That said, **automation is definitely still an option for the future**. If we can bring structure to the rule names, a script would be much more viable and in that case, likely the better option by far.

Since I chose the manual method, I was able to use multiple variables to effectively identify overlaps, such as the **rule name, description, and product category.**
The procedure I followed was as follows:

**Step One: Compare rule names**
This step yielded the most overlaps. The rule names usually contained enough context for me to make a confident decision.

**Step Two: Check the descriptions**
If I wasn't completely sure based on the names, I compared the descriptions of both rules to see if they referred to the same use case.

**Step Three: Analyze the detection logic**
If steps one and two didn't provide a clear answer, I examined the detection logic used in both rules. This step always gave me the confidence to determine whether the rules were truly the same or not.

**Now, if step three guarantees a definitive answer, why not start with that in the first place?**

The reason is simple: **steps one and two are significantly faster**. **Step three served more as a last resort**. Finding the detection logic required accessing either the Splunk or Sentinel environments, since this information was not included in the exported Excel sheet.

I'll go into more detail about this in Phase Two of my analysis, but for now, **most overlaps were easily identified in steps one and two**, with only a fraction of the time that would have been spent using step three alone.

The Excel sheet contained 424 lines of rules. The next step was to **identify which rules overlapped and mark them** so they could be easily found and analyzed within the document.

Overlaps typically consist of a pair of two rules, or in some cases, three or more. The most logical approach was to **tag these groups as single entities**, ideally within the existing Excel sheet to keep everything centralized.

**So, how could this be done in Excel?**

The solution was simple: I added an extra column called **‘Match Tag’**, which essentially means “these rules match and belong to the same group.” If two or more rules matched, I assigned them a shared tag. For example, two rules that detect password guessing on Windows would receive the match tag: **‘Windows – Password Guessing’**.

This method allows me to filter the sheet by the 'Match Tag' column and view all overlapping rules together, without interference from unrelated ones. It keeps the overview clean, organized, and easy to navigate.

## 2.4. Extra findings during Phase One

In this chapter, I’ll explain some additional findings I made during Phase One. I’ve chosen to keep these separate from the core assignment in order to maintain a clear overview and distinguish the official scope of my internship project from the extra initiatives I took on.

### 2.4.1. Optimized rule names

As mentioned earlier, the names of overlapping rules were often significantly different from one another. Here is an example:

- **Sentinel rule**: *Windows – Living Off The Land Binaries Executed From Non-Default Directory*
- **Splunk rule**: *Windows – LOLbins From Office Documents*

Even though these rules aim to detect similar behavior, their names vary greatly. For example, I happened to know that *LOLbins* is an abbreviation for *Living Off The Land Binaries*, but someone else might not, or might not recognize it right away. This can cause confusion, and confusion leads to inefficiency, ultimately wasting valuable time on something that could be easily avoided. Without a consistent naming structure, it becomes harder to identify related rules at a glance, which slows down the analysis process and increases the risk of missing overlaps.

To improve consistency and clarity, I added an extra column to the Excel sheet called **‘Name Optimized’**. As the name suggests, this column contains an optimized name for each rule within a given match tag.
In most cases, this meant assigning the same name to both Splunk and Sentinel, within a match tag. However, this wasn’t always possible.

Identifying overlapping rules isn’t always a black-and-white process, it often involves judgment calls about what should be considered “the same.” For instance, two rules

might share the same purpose, such as detecting **Living Off The Land** executions. One might detect executions from non-default directories, while the other focuses on executions triggered by Office documents. While the overall goal is the same, the specifics differ, so these rules cannot be treated as literally identical.

Reusing the example from above, the optimized names for those rules would be:

- *Windows – Living Off The Land Binaries – Executed From Non-Default Directories*
- *Windows – Living Off The Land Binaries – Executed From Office Documents*

These new names allows for consistent naming that still reflects the nuanced differences between rules. It improves clarity while maintaining consistency, a balance that's important when dealing with complex or similar rule sets.

Having consistent names like this also ties back to the automation aspect I discussed earlier. When the naming is consistent, automation becomes much more feasible. For example, rules tagged with *'Living Off The Land Binaries'* can easily be identified by a script or other automation tool, with the confidence that they refer to the same type of detection. This reduces ambiguity and increases the reliability of automated processes.

### 2.4.2. Optimized MITRE & ATT@CK Tactics and techniques

Each detection rule has MITRE ATT&CK tactics and techniques assigned to it. This helps tie each rule to specific attack vectors or paths that adversaries commonly use. For context, this information is also visible, per rule, in the exported Excel sheet.

While analyzing the rules I noticed some inconsistencies in how these tactics and techniques were applied. In some cases, they didn't seem to match the rule at all. Here's an example to illustrate my point:

**Detection rule name:**
Brute Force Password Guessing On VPN User

This rule was labeled with the tactic Initial Access and the technique T1078 (Valid Accounts). A more accurate classification would be T1110 (Brute Force) with the tactic Credential Access. If tactics and techniques are incorrectly assigned like this, it can cloud the overview and introduce inefficiencies, ultimately leading to time lost on something that could easily be avoided.

In the example above, my interpretation is based solely on the rule name and description. However, if the rule is actually detecting brute-force password guessing

using valid credentials, then assigning technique **T1078 (Valid Accounts)** would indeed be appropriate.

That said, this is just my perspective and to be fair, assigning MITRE tactics and techniques isn't always black and white. It's highly context-dependent, and there's often more than one correct answer.

I assign MITRE techniques to detection rules using the following methodology:

1. **Identify the 'why'** - this corresponds to the *tactic* in MITRE (the attacker's objective).

2. **Determine the 'how'** - these are the *techniques* or *sub-techniques* (the method used by the attacker).

For example, in the case of a rule detecting brute-force password guessing:

- The **why** is to gain access to credentials (Tactic: Credential Access)
- The **how** is by brute-forcing passwords (Technique: T1110, Brute Force)

Now, suppose the brute force attempt uses valid credentials belonging to legitimate accounts. Some might also assign **T1078 (Valid Accounts)**, which falls under the Initial Access tactic. While that's a reasonable interpretation, I consider it suboptimal for this reason:

Assigning **T1078 (Valid Accounts)** broadens the scope of what the rule is actually detecting. This rule is designed to detect credential access through brute force, not necessarily the successful initial access using valid accounts. There's no certainty the brute force leads to initial access, so I believe it's **more accurate** to limit the mapping to **Credential Access - T1110 (Brute Force)**.

The main reason I decided to optimize these classifications was to deepen my understanding of the MITRE & ATT@CK framework. It was only briefly mentioned during my studies, and I never had the chance to explore it in depth. Since this is an internship, learning is the main goal, and this seemed like a valuable opportunity to gain that experience.

Following the same methodology I used earlier, I created two additional columns in the Excel sheet: **"Techniques Optimized"** and **"Tactics Optimized."** In these columns, I added the MITRE techniques and tactics that, in my opinion, were a better fit for each rule. I applied this to all 424 rules in the sheet.

Although it took some time to complete, the process significantly deepened my understanding of the MITRE & ATT@CK framework. I'm glad I took the time to do it because the knowledge I gained is invaluable and will definitely benefit me in my future career as a cybersecurity professional.

## 2.5. Phase two: Comparing Of Detection Rules

After identifying all the overlapping rules, it was time to analyze their detection logic to find out exactly what made them different. As mentioned earlier, the only way to verify whether rules are truly the same is by comparing their underlying detection logic.

At the beginning of Phase Two, I asked myself a few key questions:

- **Where can I find the detection logic for these rules?**
- **What is the most efficient way to go about this?**
- **What criteria should I use for comparison?**
- **Since documentation is important, how will I document everything effectively?**

Let’s go through these one by one.

The detection logic of active rules could be found in either the test environments (*ACC* for Splunk, *Cynalco* for Sentinel) or directly within the customer environments.

**Splunk Methodology:**

1. Check the **Detection Engineering Team (DET)** use-case library.
2. Filter based on the rule name to identify which customer environment uses the rule.
3. Go to the relevant customer environment and search for the rule using a query.

**Sentinel Methodology:**

1. Log in to the **Azure portal** using my **Cynalco** account (used for the Sentinel test environment).
2. Navigate to **Sentinel** → **Analytics** and search for the rule there.

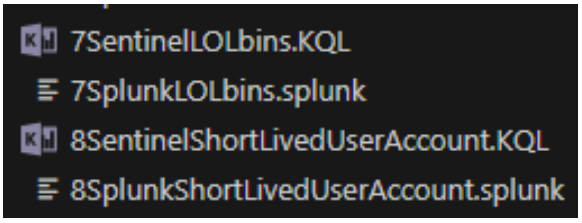
Since Splunk’s test environment is currently not up to date, I had to rely on the customer environments for retrieving the detection logic in those cases.

Because manually looking up each detection rule was time-consuming, I decided to store all the detection rules in a centralized location for future reference. I saved them as separate files in OneDrive, which allowed me to easily access and manage them through Visual Studio Code.

By organizing the rules as individual .KQL (Kusto Query Language) and .splunk (Splunk Search Processing Language) files within a single folder, I was able to quickly open and edit them in Visual Studio Code, creating a convenient and centralized repository that eliminated the need to repeat search steps each time.

One challenge I encountered was that Windows organizes files alphabetically or numerically by default. This meant that if a rule started with an “A” or a number (for example, “1”), those files would appear at the top of the list, while others would be pushed to the bottom. As a result, the detection rules became scattered throughout the folder, making it harder to maintain a clear overview.

To solve this, I created a simple naming convention where I assigned a number to each match tag, followed by the relevant technology.




This system allowed overlapping rules to be easily identified by their assigned number, significantly improving the organization of the folder and enhancing my workflow.

Now that I had the detection logic, the next step was to **compare the overlapping rules** from Sentinel and Splunk. But that raised another question: How can I document these comparisons?

The solution I came up with was to use **difference tables**. These are tables that group rules under a specific match tag and list the relevant comparison criteria. I created one table per match tag and placed them neatly in a Word document.

At this stage, I dove into the analysis. I began experimenting with different criteria for each table, assigning them based on what made the most sense per case.

It wasn’t standardized yet, I was still figuring out which aspects of the rules were worth comparing, but it helped me get familiar with the syntax and detection logic used in both Splunk and Sentinel. An example of the first draft of difference tables can be found below:

| Feature | Sentinel | Splunk |
|---|---|---|
| **Time Filtering** | ago(1h) / ago(1d) | earliest=-24h@h latest=now |
| **Field Extraction** | extract(@regex, 1, example) | rex or automatic extraction |
| **Aggregation** | join kind=inner(...) on example | Subsearch [search ...] |
| **Renaming** | project-rename NewName = OldName | rename oldname as newname |
| **Whitelisting** | None | None |

I quickly realized this unstructured approach introduced several major issues:

**1. Difficult to draw conclusions**
It became hard to recognize trends or patterns across the tables when each used different criteria.

**2. Lack of reproducibility**
If someone else (or even I) wanted to repeat the comparison in the future, it would be unclear why certain criteria were chosen. This makes the analysis unreliable.

**3. Future automation becomes impossible**
Without standardized comparison fields, automation would be infeasible. Every script or model would need custom logic per match tag, which is inefficient and error-prone.

In hindsight, this ties back to the earlier issue of inconsistent rule naming. Especially for automation, standardized naming and comparison criteria are essential. What I didn't realize at the start was that I was beginning to create a **procedure**, more specifically**, a procedure for aligning detection rules written in two different query languages.**

This realization marked a big step forward in my internship assignment. I came to understand that it wasn't just about 'getting the job done', it was about **creating a blueprint** that others could follow, learn from, and refine in the future.

With that in mind, I started thinking about **universal criteria** I could use for my difference tables, ones that would apply to all rules, not just a specific subset. To do this effectively, I had to take a step back and look at the rules from a higher-level perspective. A few key questions came to mind:

- **How are these rules actually built?**
- **Can I identify consistent patterns within them?**
- **What criteria are relevant for *all* detection rules?**

**Context: Reverse Engineering a Real-World Issue**

At the same time, I had just completed a task unrelated to my main assignment, the explanation of which can be found in the **Use-case validation in Splunk** document, found in the **Attachments** section of this report.

In short: a few detection rules in a customer environment failed to trigger during a security audit. My job was to investigate why and to figure this out, I had to **reverse engineer** the detection rules, break them down step by step to understand where the failure occurred.

This experience turned out to be incredibly valuable. It gave me deeper insight into the internal structure of detection rules and allowed me to recognize **common patterns** in how they're built.

After that assignment, I revisited my earlier questions with a more informed perspective. Identifying relevant comparison criteria became much easier because I now had a clearer understanding of the building blocks that make up detection rules.

Based on my deeper understanding of detection rules, I defined a set of universal criteria to use in my difference tables. These criteria allow for consistent, objective comparison across all matched rules, regardless of whether they originate from Splunk or Sentinel. An example of the second draft is shown below:

| Feature | Sentinel | Splunk |
|---|---|---|
| **Data Source** | Syslog | Index=”os” |
| **Threshold** | N/A | N/A |
| **Whitelisting** | N/A | N/A |
| **Counting Method** | Matches useradd events by username | Counts events by host |
| **Event Filtering** | ProcessName ==”useradd” | Searches for process “useradd” |
| **Output** | TimeGenerated, User,Computer, | Hostname, user |
| **Detection Scope** | Looks for all account creation events per user | Looks for all account creation events per host |

The criteria are as follows:

1. **Data Source**
   The origin of the logs or events used in the detection. This could be a specific product (e.g., Azure AD, Windows Security Logs) or service. Knowing the data source is essential to understanding the scope and context of the rule.

2. **Threshold**
   Some rules include a threshold to determine when an alert should be triggered. A threshold that is too low may cause alert fatigue due to excessive noise, while one that is too high might miss actual threats. This is a key parameter to evaluate.

3. **Whitelisting**
   The exclusion of trusted users, IPs, or systems to reduce false positives. This is something I noticed was often missing in Sentinel rules. A lack of consistent whitelisting can lead to unnecessary alerts and inefficient investigations.

4. **Counting Method**
   How the rule aggregates or counts events. For example, does it count all “secret view” events for a user or just the distinct secrets viewed? This plays a big role in how aggressive or conservative the rule is.

5. **Event Filtering**
   The filters applied within the rule logic. For example, a brute force detection rule might only include events where the authentication result equals "failure". These filters define the precision and relevance of the detection.

6. **Output**
   This is what the analyst sees when the rule is triggered. The output should provide all relevant context, but not be overloaded with unnecessary data. A clean, concise output helps analysts act quickly and accurately.

7. **Detection Scope**
   A brief summary of what the rule is actually detecting, and just as importantly, what it *does not*. This helps define the rule’s limitations and makes its purpose clearer during comparison.

With these new criteria in place, I went back and rebuilt the difference tables. It was a lot of work and not always easy, but it forced me to deeply understand how each rule functioned. That level of insight turned out to be incredibly useful, especially later on when it came time to align and refine the rules.

But even after completing phase one and two, something still felt missing.

There were plenty of conclusions I had drawn, and I knew I was no longer just completing a checklist, I was developing a procedure. And now, with phase three (implementation) on the horizon, I realized I didn’t yet have a solid plan of action.

Yes, I had the difference tables. Yes, I knew name standardization was necessary. But the questions still stood:

- **What exactly needs to change for each rule?**
- **And just as important, how am I going to implement those changes?**

It was time to turn insights into actions and I needed a framework to do this systematically.

## 2.6. Phase Three: Research for improvement

Having made an inventory of the overlapping rules and compared them, it was time to dive deeper and figure out exactly what needed to change to improve them, and whether any new rules needed to be added.

To approach this effectively, I needed a framework that would help me systematically identify what changes were required for each rule. Since this had never been done before within Cegeka, the key was to develop a clear, repeatable procedure, a blueprint that others (and even my future self) could follow, adjust, and build upon as needed.

The most efficient way to do this was by creating an Excel sheet. I laid out a detailed plan of action that consisted of multiple steps, each with a checklist to keep track of progress. This plan can be found in my graduation portfolio.

It consists of the following steps:

### 2.6.1. Step One: Name change

Like mentioned before, I see potential to automate the alignment of the rules in the future. However to automate you need a consistent way of doing things. Changing some of the existing rule names is step one.

I created fields that show the current name and the optimized name that needs to be implemented in both the Splunk and Sentinel environments. This way it's easily visible what the end product will be and why this is necessary.

### 2.6.2. Step Two: Alignment

In step two, I integrated the content of my difference tables. However, instead of comparing and documenting the differences as I did before, I used them to draw concrete conclusions about what exactly needed to change.

Each field in the difference tables was mirrored as a column in the Excel sheet. For every rule, I wrote down specific actions, what needed to be added, changed, or improved.

To reach these conclusions, I systematically compared the two matching rules from both environments. For each criterion, I assessed which rule handled it better. Here's a concrete example: if the Splunk rule filters on Event ID 1102 and the Sentinel rule uses Event ID 104, then in the *Event Filtering* column I would write that Event ID 104 should be added to the Splunk rule and Event ID 1102 to the Sentinel rule.

Another example: if a threshold was present in the Splunk rule but absent in the Sentinel rule, then I would note in the *Threshold* column that a threshold should be added to Sentinel.

In most cases, both the Sentinel and Splunk rules had their strengths. These strengths varied per rule and per criterion. For example, in one set of rules, Splunk might have a better *Output* and *Event Filtering*, while Sentinel has a better *Threshold* or *Counting Method*. In such cases, I documented exactly what needed to be added or modified in each environment to bring the two rules into alignment.

### 2.6.3. Step Three: Implementation and Checklist

In this step the changes get implemented. Following the criteria that are defined in step two. In the realization part of this document I will go over exactly how I applied these changes. For now, when aligning a rule to a specific criterion, this can be tracked using the checklist.

The checklist uses three simple levels: **OK**, **NOK**, or **N/A**.

### 2.6.4. Step Four: Review the changes

At the end of the task, a thorough review must be carried out to ensure that nothing has been overlooked and that no errors are present. This involves carefully checking each criterion to confirm that the changes have been successfully implemented. Additionally, it is important to **test the newly modified rules** to verify that they generate the expected events.

This testing phase ensures a smoother rollout to customer environments and helps minimize the number of errors after release.

### 2.6.5. Recommendations for new rules

When aligning the rules I noticed that there was potential for new rules to be added, four to be exact, the rules in question:

**Clearing Of Event Logs and Event Log Service Shutdown**
I grouped these two rules together because they are very similar, both are designed to detect tampering with event logs. The main difference lies in what exactly they monitor: one detects Event ID 1100 (Event Logging Service has shut down), and the other detects Event ID 1102 (Audit log was cleared).

For context, the "Windows Event Log" is a collective term for three specific log types: **WinEventLog:Security**, **WinEventLog:System**, and **WinEventLog:Application**. Event ID 1100 relates to the shutdown of the entire Windows Event Log service, while Event ID 1102 is triggered specifically when the **Security** event log is cleared.

Currently, only Sentinel has a rule that detects the clearing of **event logs (1102)**. I would have liked to see this rule represented in Splunk as well. On the other hand, Splunk has a rule for detecting the **shutdown of the event logging service (1100)**, which Sentinel does not currently cover.

Both rules serve a unique purpose. If logs are cleared without shutting down the service, the Event ID 1100 rule won't trigger. Conversely, if the service is shut down without clearing the logs, the Event ID 1102 rule won't trigger. This creates a gap in detection if only one of these events is being monitored.

Attackers could exploit this gap to stay under the radar, clearing logs or shutting down logging services without being detected, effectively hiding their tracks. Therefore, having both rules present in each SIEM platform adds an extra layer of defense and closes this blind spot.

**Living Off The Land Binaries**

There is currently one rule active in both Splunk and Microsoft Sentinel for detecting **Living Off The Land Binaries (LOLBins)**. However, these rules do not serve the same purpose. In Splunk, the rule is designed to detect **LOLBins being executed from Office documents**, while in Sentinel, it focuses on detecting **execution from any directory other than C:\Windows**.

Initially, I marked these as overlapping, but after further analysis, I concluded that the goals of these rules are fundamentally different. As such, aligning them would not be meaningful.

That said, this mismatch does create a **detection gap** between the two environments. To close this gap, the most effective approach would be to **add the Office document-based detection rule to Sentinel**, **and introduce a rule in Splunk that detects execution from non-default Windows directories.**

# 3. Realization

In this part of my realization document, I will go over what I have realized, since most of my work was analysis this part of the document will be shorter, nevertheless there are some things that I'd want to highlight.

## 3.1. Realization of phase one

As mentioned earlier in the analysis, Cegeka did not have a clear overview of how many detection rules belonged to each technology, nor how many were overlapping between the two. While creating an inventory was necessary, it had been consistently postponed due to other work taking priority.

However, understanding the current detection coverage is essential, not just internally, but also for delivering high-quality service to customers. Like any cybersecurity provider, Cegeka strives to offer the best possible protection, whether that's through Splunk or Microsoft Sentinel.

With my research, I was able to provide Cegeka with valuable insight into their current detection landscape across both platforms. These are the key findings:

| | Exclusive rules | Overlapping rules |
|---|---|---|
| **Microsoft Sentinel** | 75% | 6,35% |
| **Splunk** | 18,59% | 6,35% |

This data is highly valuable because it enables Cegeka to **identify detection gaps** between the two environments and take action to close them. By optimizing detection coverage in both Splunk and Sentinel, Cegeka can ensure **more consistent protection across all customers**, ultimately strengthening their overall cybersecurity service offering.

## 3.2. Realization of phase two and three

In phases two and three, I not only compared detection rules across Microsoft Sentinel and Splunk, but also developed a **structured methodology** to bring consistency and alignment between the two platforms. This involved defining **new rule names** and a **repeatable procedure** for analyzing and improving detection rules across environments.

This is valuable not just for internal use at Cegeka, but also for **customers**, and here's why:

- **Improved Visibility and Consistency**
  By introducing optimized naming, it becomes easier to track, manage, and understand detection rules across customer environments. This ensures faster identification of issues and more efficient response to threats.

- **Fewer Detection Gaps**
  The alignment procedure helps identify and eliminate blind spots by ensuring both SIEM platforms offer comparable levels of coverage. This is crucial for customers relying on accurate and comprehensive threat detection.

- **Higher Quality and More Reliable Detections**
  Comparing rules based on well-defined criteria, such as thresholds, whitelisting, and event filtering, leads to better-tuned detection logic. Customers benefit from **reduced false positives** and more relevant alerts, which improves trust in the system and reduces alert fatigue for SOC teams.

- **Scalability and Future Readiness**
  The standardized approach makes it easier to onboard new customers, introduce new rules, and even migrate between SIEM platforms. This flexibility translates to a **future-proof** detection strategy that adapts as the customer's needs evolve.

In short, this structured approach allows Cegeka to deliver **more reliable, efficient, and consistent security monitoring services**, improving the overall customer experience and strengthening Cegeka's value proposition as a cybersecurity provider.

## 3.3. Realization of phase three

The analysis was the main objective of my internship assignment. However, since I completed it well ahead of schedule, there was ample time left to actually start **implementing my findings**.

To do this, I applied the **rule alignment procedure** I had developed during phase three. This gave me a structured, step-by-step approach to make real improvements to the detection logic. Here's how I went about it:

**Step 1: Fill in the Excel Alignment Sheet**

I started by completing the Excel sheet I had created earlier. This provided a **clear and centralized overview** of the changes required for each rule. It served as my blueprint before diving into the engineering work.

**Step 2: Apply the new rule names**

I assigned new, consistent names to the rule pairs. This step ensured clarity and traceability, especially when these rules are reviewed or used by others in the future.

**Step 3: Define Rule-Specific Changes Based on Criteria**

For each pair of matching rules, I filled out the universal criteria fields in the Excel sheet (such as threshold, event filtering, output, etc.). Based on my comparison, I described exactly what needed to change.
For example:

“Change Sentinel threshold to match the value used in Splunk.”
“Add missing whitelist in Sentinel.”

This approach ensured that changes were **objective, reproducible, and based on logic** rather than assumptions.

**Step 4: Modify and Test the Detection Logic**

Next, I moved on to the actual engineering phase. I took the original queries and brought them into customer environments, where I modified and tested the detection logic according to the documented changes. Testing was essential to validate that the adjustments behaved as expected and still triggered correctly.

**Step 5: Deployment**

After successfully testing the rules, the next step was to deploy them to the customer environments. During this phase, I encountered a few challenges.
Some of the rules in the subset were very specific, meaning they were unlikely to trigger frequently. As a result, there were no available logs to test certain rules, simply because the conditions they detect had not yet occurred.
This wasn’t an issue during the earlier testing phase (step 4), where I could use certain workarounds:

- For example, I could replace a rare event ID with a more common one to generate enough log data for testing, especially when I only needed to validate the output format.

- Alternatively, I could broaden the search by temporarily removing or relaxing filters.

These methods helped verify that the rule syntax was correct, but they didn’t guarantee that the rule would function as intended in a real-world scenario without

those modifications. And since I only want to deploy rules I'm fully confident in, they had to be tested end-to-end, without any workarounds.

I was able to do this for about half of the aligned rules. The remaining rules either lacked sufficient log data or, in the case of Splunk-specific rules, did not require changes, only the Sentinel variant was updated.

In the end, **9 out of the 23 aligned rules** were ready for deployment to customer environments. While this was fewer than I had hoped, it still represents a meaningful improvement in detection quality.

As for the rest of the aligned rules, the ones that couldn't be fully tested, the plan would be to trigger them in a controlled lab environment to generate the necessary log data. Once tested thoroughly, these rules could also be deployed.

However, due to time constraints within the internship period, I was unable to begin this lab testing process. Instead, I chose to focus on finalizing and deploying the rules that had already been fully tested and verified.

### 3.3.1. Aligned rules deployment process

As mentioned earlier, I chose to finalize and deploy only the rules that had been properly tested and were ready for implementation. The deployment process differed between the two environments.

**Deployment in Sentinel**

The deployment process for Sentinel was relatively straightforward and consisted of three key steps:

**Step One: Peer Review**

Before deployment, my optimized detection rules were reviewed by my colleague Jeroen, a detection engineer. Having another set of eyes on the rules was essential, as it helped catch issues I might have missed and offered valuable new insights.

This step proved its value when Jeroen identified one rule that would likely produce an excessive number of false positives. We then took the necessary steps to revise and improve that rule before moving forward.

**Step Two: Update the Detection Logic**
Next, I implemented the updated logic in Sentinel's lab environment, called **Cynalco**. Here, I replaced the existing rule names with the newly structured names and updated the logic with the optimized versions I had developed.

**Step Three: Submit an RFC (Request for Change)**

RFCs are a formal requirement for managing changes in an IT environment. They ensure transparency, both internally within Cegeka and externally to its customers.

To modify or update existing rules, an RFC must be created. Each RFC includes:

- **A summary of the changes made**
- **A list of affected rules**
- **The name of the author** (in this case, myself)

Since it was my first time creating an RFC, I received guidance from Jeroen, as this process is both critical and precise.

**Step Four: Deployment**

Once completed, the RFC is handed off to the SOC engineering team. In my case, I specified which rules needed to be updated and where the updated files could be found. The engineers then took over by:

1. Creating ARM templates from the updated rules,
2. Converting those templates into **Bicep files**, and
3. Adding them to the deployment pipeline, which ensures the rules are automatically distributed to customer environments.

**Deployment in Splunk**

The deployment in Splunk was a bit more challenging:

**Step 1: Peer review**

Just like with Sentinel, the first step was a peer review. The Splunk rules were reviewed by my colleague Jeroen, and once again an error was identified, one rule had too broad a scope, which would have caused it to trigger far too often.

This reinforced the value of peer reviews. Both the Sentinel and Splunk rules contained small errors or misconfigurations that I hadn't noticed. Without a second pair of eyes, these flawed rules would have been pushed into customer environments, potentially causing false positives or missed detections.

**Step 2: Change configuration packages**

The next step was to update the configuration packages in which the Splunk rules are stored. These rules are grouped by Splunk app (essentially by product, such as Azure), and the changes I made affected two separate apps. This meant I needed to update the configuration files for both.

These packages are normally edited directly in the Splunk GitLab environment. However, since I didn't have access to GitLab, my colleague Laurens (who guided me through the Splunk deployment process) provided the necessary files so I could make the changes locally. After completing the updates, I returned the files to Laurens, who then uploaded them to GitLab, replacing the old versions.

**Step 3: RFC creation**

While I was working on the configuration changes, Laurens created a new RFC (Request for Change) based on an internal template. This was slightly different from the Sentinel process, where I simply added my changes to a larger RFC containing updates from all detection engineers. In this case, the RFC was created specifically for the changes I had made.

After the RFC was created, I filled it out following the same methodology as in Sentinel, detailing what was changed, which rules were affected, and who authored the RFC.

**Step 4: Deployment**

Splunk's deployment process differs significantly from Sentinel's. Once the changes are made in GitLab, version control is applied. This means every change must be assigned a new version number. To do this, we use a custom script executed in the command line within the package folder.

Next, a Jenkins pipeline is used to prepare the deployment. Here, we specify key details such as:

- Which customer environments the changes should be deployed to,
- The corresponding RFC for the change.

This pipeline packages the updated files and places them into the **package registry**. Once complete, a second pipeline, managed by the Splunk Engineering team, takes over. This pipeline pulls the packages from the registry and distributes them to the customer environments.

**But, why is collaboration with Splunk Engineering necessary?**

Unlike Sentinel, which is fully cloud-native, all Splunk services at Cegeka currently run **on-premises**. This difference has a major impact:

- If **Sentinel** goes down, it's Microsoft's responsibility.
- If **Splunk** servers go down, it's an internal issue for the Splunk engineering team.

This means certain failures (like the Splunk Ansible server being unavailable) can delay the pipeline and require intervention from the Splunk engineering team. This is also why there are two separate pipelines:

- **SOC Pipeline:** Prepares packages for deployment.
- **Engineering Pipeline:** Deploys packages to customer environments.

In contrast, **Sentinel's full deployment flow is handled entirely by the SOC team**.

That said, the Splunk deployment workflow could benefit from further optimization. This might make for a great assignment for a future intern. If I had more time left in my internship, I would've definitely liked to explore this area myself.

# 4. Conclusion

Looking back on my internship at Cegeka, I can confidently say it has been a **valuable and enriching experience**. Over the course of three months, I had the opportunity to immerse myself in the world of detection engineering within a professional SOC environment.

By optimizing the internal workflow through my internship assignment, reverse-engineering detection rules to solve real-world customer issues, create and test new detections rules from the ground up, analyzing threat hunts like ESC1 and ESC2, and translating them into actionable detection rules in collaboration with CSIRT**, I significantly expanded both my technical skills and my practical understanding of cybersecurity**. I also actively participated in the testing and implementation phases, further reinforcing my knowledge.

One of the most important lessons from this experience was **gaining a clear understanding of how threat detection is built and fine-tuned in practice**, from initial threat research to rule creation, testing, and validation.

Being part of the collaboration between different SOC roles taught me **the critical importance of teamwork and clear communication** in a security operations environment.

Throughout the project, I developed a more structured approach to problem-solving and **learned to explain technical processes more clearly to different audiences**. From feedback of Lander (my internship supervisor) and Steven (SOC detection engineer team leader), I realized that my **communication was initially inconsistent**, a key area I worked hard to improve. I also learned that detection engineering is not just about technical accuracy, but also about ensuring that outputs are practical and usable for SOC analysts, ultimately strengthening the entire defense chain.

I am proud that I was able to contribute directly to Cegeka's security offering**, both by optimizing internal workflows and by creating new detection rules that help protect customers against emerging threats.** This experience has confirmed my passion for cybersecurity and has further motivated me to pursue a career in this field.

In short, my internship was a major learning experience, both professionally and personally, **and I am grateful for the opportunity to have been part of the Cegeka Modern SOC team.**

Finally, I would like to **sincerely thank the entire SOC team** for their support and willingness to help whenever needed. A special thank you goes to **Steven (Detection Engineering Team Leader), Lander (my internship mentor at Cegeka), and Alexander (my internship supervisor at Thomas More)** for their invaluable guidance, both technical and in soft skills, throughout the internship.

**Looking Ahead**

Looking ahead, I am eager to continue building on the skills and knowledge I gained during my time at Cegeka. I genuinely enjoyed my work every day, often working overtime not because I had to, but simply because I was passionate about what I was doing.

I realized that **working in a SOC environment suits me extremely well, and that is the path I plan to continue pursuing in the future**. Cegeka has expressed interest in having me join the SOC team permanently, but currently, there are no open positions available.

Reflecting on the experience, I've realized how much I enjoy hands-on detection engineering and the broader field of cybersecurity. Because of this, I'm now seriously **considering pursuing a master's degree in cybersecurity** after completing my bachelor's. I'm still weighing my options, **either continuing my studies or entering the workforce directly.** Whichever path I choose, this internship has confirmed that cybersecurity is the direction I want to take my career.

This internship has truly been a defining step in my career journey, and I am excited for what lies ahead.

# 5. Reference list

In this section, I have listed all spoken and written sources consulted, both during my internship and in the preparation of this realization document.

**Spoken Sources**

*My internship mentors and supervisor:*

**Steven Beullens**
Team Leader SOC Process Architecture and Detection Engineering | Cegeka Hasselt
(Provided feedback and guidance during my internship on technical and non-technical topics, especially communication and leadership.)

**Lander Cruysberghs**
SOC Detection Engineer | Cegeka Hasselt
(Offered feedback on both non-technical and technical aspects, particularly related to Detection Engineering. Lander also provided invaluable tips for my bachelor's thesis.)

**Alexander Hensels**
Lecturer and Internship Supervisor | Thomas More Geel
(Supervised my bachelor's thesis and served as my main point of contact for all internship-related questions.)

*Members of the Cegeka SOC team:*

**Kris Bogaerts**
SOC Detection Engineer | Cegeka Hasselt
(Provided guidance with the Threat Hunts.)

**Maarten Louis Claes**
SOC Detection Engineer | Cegeka Hasselt
(Provided in-depth guidance for Microsoft Sentinel.)

**Maarten Van Berkel**
Security Architect | Cegeka Hasselt
(Provided guidance for Microsoft Sentinel and helped me setup my account for accessing the environments in Microsoft Sentinel.)

**Laurens Vermunt**
Security Engineer | Cegeka Hasselt
(General guidance for Splunk.)

**Jelle Stoffelen**
Security Engineer | Cegeka Hasselt
(Provided guidance for Splunk.)

**Jean-Luc Vliegen**
SOC Engineering | Cegeka Hasselt
(Provided guidance for SOC engineering related topics.)

**Niels Pirotte**
Team Leader SOC Engineering | Cegeka Hasselt
(Provided access to the Gitlab environment.)

**Warre Tielens**
Support Engineer | Cegeka Hasselt
(Provided access to the Gitlab environment.)

**Christos Katopis**
CSIRT Analyst | Cegeka Greece
(provided guidance for testing detection rules.)

**Robin Lenaerts**
Penetration Tester | Cegeka Hasselt
(Provided access to the penetration testing lab.)

**Arne Holemans**
SOC Analyst | Cegeka Hasselt
(Gave me knowledge about the SOC analyst role.)

**Rune Mannaerts**
SOC Engineer | Cegeka Hasselt
(Gave me knowledge about the SOC Analyst and SOC Engineering roles.)

**Leander Van Bael**
SOC Analyst | Cegeka Hasselt
(Gave me invaluable tips for my bachelor’s thesis.)

**Written Sources**


Microsoft (2025) – *Defender for Cloud Apps' transition from alerts to behaviors*
Retrieved from: Investigate behaviors with advanced hunting - Microsoft Defender for Cloud Apps | Microsoft Learn

Microsoft (2025) – *Threat intelligence in Microsoft Sentinel*
Retrieved from: Threat intelligence - Microsoft Sentinel | Microsoft Learn

Microsoft (2025) – *Understand security coverage by the MITRE ATT&CK ® framework*
Retrieved from: View MITRE coverage for your organization from Microsoft Sentinel | Microsoft Learn

Microsoft (2024) – *Tutorial: Detect Threats by using analytics rules in Microsoft Sentinel*
Retrieved from: View MITRE coverage for your organization from Microsoft Sentinel | Microsoft Learn

Microsoft (2024) – *Threat detection in Microsoft Sentinel*
Retrieved from: Threat detection in Microsoft Sentinel | Microsoft Learn

Microsoft (2024) – *Create a scheduled analytics rule from scratch*
Retrieved from: Create scheduled analytics rules in Microsoft Sentinel | Microsoft Learn

Microsoft (2024) – *What is Microsoft Sentinel*
Retrieved from: What is Microsoft Sentinel? | Microsoft Learn

Microsoft (2025) – *Kusto Query Language overview*
Retrieved from: Kusto Query Language (KQL) overview - Kusto | Microsoft Learn

Microsoft (2025) – *Tutorial: Learn common operators*
Retrieved from: Tutorial: Learn common Kusto Query Language operators - Kusto | Microsoft Learn

Microsoft (2024) – *Tutorial: Use aggregation functions*
Retrieved from: Tutorial: Use aggregation functions in Kusto Query Language - Kusto | Microsoft Learn

Microsoft (2024) – Best practices for Kusto Query Language queries
Retrieved from: Best practices for Kusto Query Language queries - Kusto | Microsoft Learn

Splunk (2024) – *Splunk Enterprise Overview*
Retrieved from: About Splunk Enterprise - Splunk Documentation

Splunk (2024) – *Search Tutorial*
Retrieved from: About the Search Tutorial - Splunk Documentation

Splunk (2021) – *Exploring the Search views*
Retrieved from: Exploring the Search views - Splunk Documentation

Splunk (2021) – *Specifying the time ranges*
Retrieved from: Specifying time ranges - Splunk Documentation

Splunk (2021) – *Basic searches and search results*
Retrieved from: Basic searches and search results - Splunk Documentation

Splunk (2023) – *Use fields to search*
Retrieved from: Use fields to search - Splunk Documentation

Splunk (2021) – *Use the search language*
Retrieved from: Use the search language - Splunk Documentation

Splunk (2022) – *Use a subsearch*
Retrieved from: Use a subsearch - Splunk Documentation

Splunk (2022) – *Enabling field lookups*
Retrieved from: Enabling field lookups - Splunk Documentation

Splunk (2023) – *Search with field lookups*
Retrieved from: Search with field lookups - Splunk Documentation

Splunk (2025) – *Splunk to Kusto cheat sheet*
Retrieved from: Splunk to Kusto map - Kusto | Microsoft Learn

IBM (2024) – *SPL to KQL converter: Bridging Splunk and QRadar gap*
Retrieved from: SPL to KQL converter: Bridging Splunk and QRadar gap - IBM Developer

Raul Carmona (2024) – *AD CS 101: Introduction to Active Directory Certificate Services & How to Detect and Mitigate ESC1 Attacks*
Retrieved from: AD CS 101: How to Detect and Mitigate ESC1 Attacks | BeyondTrust

Raj (2025) – *AD Certificate Exploitation: ESC2*
Retrieved from: AD CS ESC2 Certificate Exploitation: Techniques and Mitigation

# 6. Attachments

In this section, you will find two documents from separate extra assignments I completed for Cegeka. I created this documentation while working on those assignments, which included a use case validation in Splunk and threat hunts for ESC1 and ESC2.

## 6.1. Document A: Use-case validation in Splunk

In late 2024, a security audit was conducted at one of Cegeka's client companies. During this audit, it was discovered that certain detection rules in the Splunk environment did **not trigger** as expected. This raised concerns about whether the **underlying log sources** for these rules were still functioning correctly.

### Plan of Action

To address this issue, I was tasked with investigating the detection rules in question. An Excel sheet was provided that contained a list of **30 rules** needing review. All of these rules were active in the customer's Splunk environment.
The first step was to **locate each rule** in the environment. To do this efficiently, I used the following search query in Splunk:

*| rest splunk_server=* /servicesNS/-/-/saved/searches | search action.correlationsearch.label="*Windows - Scripts launched*" | table splunk_server search  action.correlationsearch.label description disabled cron_schedule is_scheduled*

This allowed me to find the base query tied to each detection rule and verify whether the expected log sources were still producing data.

To begin, I looked up the **correlation search label** of each rule, essentially, the rule's name, within the Splunk environment. Once I located the rule, I **copied the associated query** and pasted it into the search bar to check whether it still generated results.

The **testing approach was based on timeframes**:

1. I first ran the **full query** over a short time window (usually the past month).
2. If that didn't yield results, I tested the **base query with base filtering applied** within the same timeframe.
3. Still no results? Then I began applying **wildcards** to the filtering conditions of the base query to broaden the search.

These detection rules were thoroughly tested in the past, so the issue likely didn't lie in the query syntax itself. **In most cases, the root cause was a change in the client's IT environment or on their endpoints**, for example, an updated logging policy, system reconfiguration, or even a disabled log source.

That's why **the first step in identifying the problem** is always checking whether the **log sources are still active and producing events**.

A typical rule query might look like this:

```
index=wineventlog sourcetype="WinEventLog:Security" EventCode=4688 "scrons.exe"
| lookup asset_inventory.csv ComputerName OUTPUT Department Location Criticality
| table _time, ComputerName, Department, Location, Criticality
```

**Disclaimer:** This rule does not exist, it is a fictional example used purely to illustrate what a typical detection rule looks like in Splunk.

This example rule searches within the wineventlog index, where Windows stores all of its logs. More specifically, it targets the WinEventLog:Security sourcetype. From there, it filters for events with **EventCode 4688**, which corresponds to the creation of a new process on a system. In this case, it's looking for a specific process named **"scrons.exe"**.

After identifying such events, the query uses a **lookup table (asset_inventory.csv)** to enrich the data. It matches on the ComputerName field to determine additional context such as **department**, **location**, and **criticality level** of the machine involved.

I ran this rule over a **1-month timeframe**, meaning it searched for relevant events that occurred within the past month. However, no results were returned.

There could be several reasons for this:

- The **log source** might be non-functional or not forwarding data.
- There were **no instances** of "scrons.exe" running during that timeframe.
- The **lookup table** might be missing or broken.
- There might be a **syntax error** or a typo in the query.

For this assignment, my responsibility was only to check whether the **log source** was functional. Issues like broken lookups or bad syntax were outside the scope, although in some interesting cases, I took the extra step and fully **reverse-engineered** the rule to understand it better and identify potential problems.

For now, however, let's focus on the **main objective**: verifying whether the **log source is still functional**.

The **log source** refers to the origin of the data, in this case, wineventlog, and more specifically, WinEventLog:Security. In the example query, we apply **base filtering**, which is the initial layer of filtering done directly on the raw logs. For this query, the base filtering consists of EventCode=4688 and process name "scrons.exe".

In more complex rules, additional filtering may be applied after this base layer, but for the purposes of this documentation, we won't go into those more advanced filters.

Sometimes, running the query with just the base filtering is enough to return results. That depends on how specific the rule is. In this case, the rule is **very specific**, because it searches for a rare process name **"scrons.exe"** which is not commonly seen on typical endpoints, unlike more standard executables such as cmd.exe.

Since the goal is only to check whether the **log source** is actively sending data, we don't need to be that specific. Instead of filtering on "scrons.exe", we can **replace that part with a wildcard (*)** to catch **any** executable being logged. This broadens the search and helps us verify whether the log source is generating relevant events at all.

This approach allows us to **validate the functionality of the log source** without getting blocked by overly narrow filters.

The log source turned out to be functional. I repeated this verification process for every rule listed in the Excel sheet that failed to trigger during the audit.

After testing, I documented the results directly in the Excel sheet. Next to each rule's name, there's a tab labeled **"Base Query"** where I recorded the base query I used to test the log source. This made it easy to trace exactly how each rule was tested.

Adjacent to that is the **"Comment"** tab, where I summarized the outcome of the test. In most cases, I wrote **"Results"**, which indicates that the base query returned data and the log source was working correctly.

In cases where the **full query appeared to be working**, I wrote **"Results (full query appears operational)"**. If only part of the query seemed functional, I noted **"Results (near full query operational)"**. This helped distinguish between rules that were fully operational and those that might still require further inspection.

## Findings

Out of the 30 rules I reviewed, **two rules appeared to be fully operational**, their full queries returned results and didn't show any signs of malfunction.

As mentioned earlier in this document, some rules caught my interest during testing, prompting me to go beyond just checking the log source. One such case is described below.

**Case Example: Detection Rule with Regex Filtering**

While validating a rule, I noticed something unusual. According to Splunk, the rule had detected **30 matching events**, but **none of them were actually visible** in the search results.

This raised suspicion, so I investigated the rule further by examining its query. I noticed it used **very specific regular expressions (regex)** for filtering:

| *regex process = ":\/\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"*

| *regex process != "(.*(10\..*|192\.168\..*|169\.254\..*|127\..*|172\.1[6-9]\..*|172\.2[0-9]\..*|172\.3[0-1]\..*).*)"*

The second line (the exclusion filter) works as intended, it correctly excludes private IP ranges such as 192.168.x.x or 10.x.x.x.

However, the **first regex line**, which is supposed to **include any IP address pattern**, doesn't seem to function properly. After removing this specific filter and rerunning the search, the results became visible again.

## Results

The log sources for **all thirty rules** were thoroughly tested and confirmed to be ingesting logs correctly. Based on my research, the team was able to **conclude that there are no issues with log ingestion**.

To continue troubleshooting, the next step is to simulate or trigger the **specific events** that these detection rules are designed to catch. This can be done either by the **client themselves** or, if desired, with support from **Cegeka's CSIRT team**.

At this stage, the **use case has been temporarily closed**, pending further input from the client. This marks a successful conclusion of the current phase and demonstrates that I was able to **solve a real-world issue for a client**, contributing directly to operational effectiveness.

## 6.2. Document B: ESC1 and ESC2 threat hunts

At Cegeka, CSIRT analysts conduct regular weekly Threat Hunts to stay ahead of emerging threats. These hunts focus on identifying relevant attack techniques and behaviors that may affect customers or internal environments. The results of each hunt are compiled into written reports, which include technical analysis and proposed detection strategies. These are then passed on to the Detection Engineering team for further investigation and implementation.

I had the opportunity to review two such threat hunts, **ESC1** and **ESC2**, and translate their findings into practical detection rules that could be used in our SIEM environments.

# Important Terminology (for context)

Before diving into the documentation, I want to briefly explain some technical terms to ensure clarity. Since this topic is relatively specialized, it's useful to understand these concepts up front.

**What is Kerberos?**

Kerberos is a network authentication protocol designed to **securely verify the identities of users and services over potentially insecure networks** (such as a corporate LAN or even the internet). It uses **secret-key cryptography and a trusted third-party system** to authenticate clients without exposing their passwords during transmission.

Think of Kerberos like passport control at an airport. Instead of presenting your password at every checkpoint, you receive a ticket (called a *ticket-granting ticket*) that proves your identity. This ticket can then be used to access other services securely, without repeatedly sharing sensitive credentials.

**What is PKINIT?**

**PKINIT** stands for **Public Key Cryptography for Initial Authentication in Kerberos**.
It's an extension to the Kerberos authentication protocol that allows users to authenticate using **public key certificates** instead of just a password.

This means users can log in with a smart card or a digital certificate, a method considered more secure and harder to compromise than traditional password-based authentication.

**What is a TGT (Ticket Granting Ticket)?**

A TGT is a core component of the Kerberos protocol. It's essentially **proof that you've successfully authenticated**. Once you have a TGT, you can access other network resources without needing to re-enter your credentials each time.

Here's how it works:

1. **Login** – You authenticate (either via password or certificate).
2. **Receive TGT** – The Key Distribution Center (KDC) issues a TGT.
3. **Request Access** – You use your **TGT** to request Service Tickets for specific resources (e.g., file shares, email, applications).
4. **TGT Validity** – The TGT is encrypted and typically valid for a set time (commonly around 10 hours).
5. **Stored Locally** – Your device stores the TGT so you can seamlessly access multiple services.

**PKINIT** and **TGT** are used together **to enable certificate-based Kerberos authentication:**

- Instead of entering a password, your certificate is used to obtain the TGT.
- From there, the rest of the Kerberos process continues as usual, using the TGT to access services securely.

This approach enhances security by eliminating passwords from the initial authentication step, reducing the risk of credential theft.

**What is SAN (Subject Alternative Name)?**

**SAN**, or **Subject Alternative Name**, is an extension in an **X.509 digital certificate** that allows multiple identities to be associated with a single certificate. These identities can include domain names, IP addresses, email addresses, and more.

While the **Common Name (CN)** field represents the primary identity the certificate is meant to secure, SAN extends the certificate's usability by including **additional valid identifiers**. This is especially useful for securing multiple domains or services with a single certificate.

**What is PKI (Public Key Infrastructure)?**

**PKI**, or **Public Key Infrastructure**, is a framework that enables secure communication, **authentication**, and **data integrity** over untrusted networks (such as the internet).

It uses **cryptographic key pairs** (a public and a private key) to:

- Encrypt data

- Verify identities
- Ensure that data has not been tampered with

PKI is the foundation behind digital certificates, HTTPS, secure email, code signing, and more.

**What is AD CS (Active Directory Certificate Services)?**

**AD CS** is Microsoft's implementation of a **PKI** within a Windows environment. It provides a full suite of tools for managing digital certificates and identity security.

With AD CS, organizations can:

- **Issue** digital certificates to users, computers, and services
- **Manage** the certificate lifecycle (e.g., renewal, revocation)
- **Validate identities** securely within an Active Directory domain

AD CS enables organizations to create and manage their own **internal Certificate Authority (CA)**, a trusted entity responsible for issuing and verifying digital certificates. This is crucial for enabling encryption, secure communication, and digital signing inside enterprise networks.

**What is an Event ID?**

An **Event ID** (or **EventID**) is a unique numerical identifier assigned to a specific type of event logged by the **Windows Event Log** system. These IDs help administrators, analysts, and security tools quickly identify and filter events of interest within the massive amount of system activity that Windows records.

## Content of the Threat hunt:

The two threat hunts I reviewed focused on the **ESC1** and **ESC2** vulnerabilities, attack paths related to **misconfigurations in AD CS (Active Directory Certificate Services)**, as documented in detail by **SpecterOps** in their research on **PKI (Public Key Infrastructure) abuse**.

These vulnerabilities fall under the category of **ESC (Enterprise Security Control) abuse techniques**, which highlight how attackers can escalate privileges by taking advantage of insecure certificate template settings.

**Conditions for ESC1:**

ESC1 becomes exploitable when the following conditions are met:

- A **certificate template** allows **Client Authentication** (used for Kerberos logon).
- A **low-privileged user** has **Enroll rights** on that template.
- The template allows the use of **Subject Alternative Name (SAN)** or allows users to **specify a custom subject**.

**Risk:**

If these conditions are met, an attacker can:

- Request a certificate **impersonating any user**, such as a **Domain Admin**.
- Use the certificate for **PKINIT** (Kerberos authentication using certificates).
- Obtain a **TGT (Ticket-Granting Ticket)**.
- Escalate privileges and potentially gain **full control over the domain**.

**Tools Used for Exploitation:**

- *Certify.exe* – The tool used for this analysis
- *ForgeCert.exe* – Another tool used for crafting malicious certificates.

**Conditions for ESC2:**

ESC2 is exploitable when:

- A certificate template **includes issuance requirements**, such as **manager approval** or **authorized signatures**.
- However, the **issuing Certificate Authority (CA)** is **misconfigured** and does **not enforce** those requirements.

**Risk:**

Due to this misconfiguration, an attacker can:

- **Bypass the intended approval workflow** entirely.
- **Request and receive certificates** that should require additional manual validation or authorization.
- Use the acquired certificate for **PKINIT authentication**, potentially leading to **privilege escalation** if the certificate represents a privileged user account.

## What Did I Do With This Information?

After thoroughly reading the threat hunt documents, I quickly realized that they contained high-quality, practical information. The documentation clearly outlined the nature of the vulnerabilities (ESC1 and ESC2), how to simulate them in a controlled test environment, and the exact steps required to exploit them.

Armed with this knowledge, my next task was to translate the analysis into actionable detection rules. The CSIRT analyst that wrote the documentation for both threat hunts (Christos from our team in Greece) had already provided a solid foundation, a blueprint for the rules, which I used as a starting point. I built upon this by improving the rule output and formatting, so the information would be more useful and intuitive for our SOC analysts during investigations.

Once the rules were refined, I deployed them into our test environment. However, detection rules are only as good as their ability to actually trigger, so thorough testing was essential. I reached out to Christos to schedule a meeting, where we walked through the process of triggering the detections together.

This session was a great learning opportunity. I got to witness firsthand how simulated attacks are executed to validate detection logic. It gave me deeper insight into attacker behavior and the kinds of actions that can be observed and flagged, knowledge that is not only valuable as an intern but essential for any blue team member working in a SOC.

By the end of the initial testing session, **three out of six detection rules were functioning as intended**, while the remaining three failed to produce results. After further investigation and fine-tuning by Christos, **five of the six rules were confirmed to be fully operational**.

## Releasing the new detection rules

Our intention was to publish all six rules simultaneously. However, despite additional time for testing, we were ultimately unable to get the final rule to trigger.

It's worth noting that **two of the published rules are based on Event ID 4898**, which is related to changes in certification services but this Event ID **is not logged by default**. To generate these events, **auditing for certification services must be explicitly enabled** in the customer's environment.

Because of this, these two rules **come with prerequisites**. It's important that this requirement is clearly documented both **within the detection rule description itself** and **communicated to customers**. This ensures transparency and allows customers to make informed decisions on enabling the required audit settings if they want to benefit from these detections.

In addition, these two rules required some additional fine-tuning. Detection engineers often refine rules to minimize false positives and ensure that the output is as clear and actionable as possible for **SOC analysts**. Initially, **two out of the six rules** were showing raw event data as output which, in this context, was not valuable for the analysts. I took steps to **extract relevant fields** from the event data, transforming the output into a more readable and precise format.

## Results

After final review our team decided to only release two rules dedicated to Microsoft Defender, not Microsoft Sentinel. The lesson I learned from this is that sometimes in a corporate environment things do not go your way. I would have liked to see more of these rules released but unfortunately, due to circumstances, this was not possible.

The implementation of these new detection rules effectively closes the detection gap that previously existed in the context of AD CS abuse. With these rules in place, we've added an additional layer of security, ensuring that our customers are now better protected against threats that were previously undetected.

By introducing these new rules, I have demonstrated tangible value to both **Cegeka** and its customers, strengthening the overall security posture and providing more robust threat detection capabilities.

## 6.3. Document C: Sentinel use-case migration

In the final week of my internship, I had the opportunity to support my colleague Maarten, a fellow detection engineer, on a project for a Cegeka customer migrating from Splunk to Microsoft Sentinel. This migration required all existing detection rules in Splunk to be recreated and implemented in Sentinel.

Maarten gave me the chance to take part in this process, and I was eager to help. I got to choose which rules I wanted to work on, and naturally, I gravitated toward PowerShell-related rules, as that's an area I find really interesting.

# My plan of action

The whole process was relatively straightforward but still required attention to detail and a clear approach. Here's how I tackled it:

**Step 1: Understand the Detection Logic in Splunk**

The first step was to really dig into the existing rule in Splunk. What exactly was the rule detecting? What filters were applied? Was there any enrichment or whitelisting used?

Only after fully understanding what the rule did could I begin to recreate it accurately in Sentinel.

**Step 2: Translate the Logic to Sentinel**

Splunk and Sentinel have query languages that are similar but definitely not the same. This step involved figuring out how to recreate the detection logic in Sentinel's KQL language.

One big difference was the handling of log fields. For example, in Splunk, there's a predefined field for the parent process. In Sentinel, that field didn't exist in the same way. So I had to extract the parent process path from the metadata field (ParentImage), then use regex to isolate just the parent process name from the full path.

**Step 3: Add Whitelisting and Enrichment**

Both enrichment and whitelisting in Sentinel are handled using *watchlists*. Splunk uses *lookups*, which are pretty similar in how they work.

Enrichment was used to add extra context to the detection output, for example, additional info about a specific process. I'd typically do this by joining the rule's result with a watchlist using an inner join, based on something like the process name.

Whitelisting worked a bit differently. Let's say I wanted to whitelist certain users, I'd load a specific field like UserInitiator from the watchlist into a variable and compare it to the user in the rule. That logic would look like this:

| *where User !in (UserInitiator)*

After implementing the logic, it was time to test the detection rule to make sure it worked as expected.

## Generating Test data

Luckily, Cegeka has a great test setup: the Cynalco environment. It includes domain controllers, child domains, and a range of virtual machines, a whole virtual forest to play with and safely simulate detection scenarios.

Let me walk you through one specific example I tested: detecting the deletion of shadow copies via PowerShell.

**Shadow copies** are automatic file or folder backups created by Windows. They're useful for:

- **Restoring older versions of files**
- **File recovery**
- **System restore points**

Because they're so useful for recovery, attackers (especially those using ransomware) often try to delete them to prevent system recovery.

The detection rule I worked on looked for:

- **The parent process being powershell.exe**
- **A child process of either vssadmin.exe or wmic.exe**
- **And a command containing the words delete and shadow**

To trigger this rule safely, I needed to simulate the behavior without actually deleting shadow copies. So I used this test command:

*powershell.exe -Command "Start-Process vssadmin.exe -ArgumentList 'delete testing shadows /all /quiet'"*

I added the word **'testing'** to the command to make it invalid. It fails to execute, but it still launches vssadmin.exe and contains the keywords delete and shadow, so the detection rule still triggers. Perfect for safe testing.

I used this kind of workaround for testing most of the rules, being careful not to run anything destructive while still ensuring the rule would fire under the right conditions. A bit of creativity goes a long way here, especially when you're working with potentially dangerous commands.

**Results**

This assignment ended up being one of the highlights of my internship. Even though I only had a week left, I dove into the work and spent most of my time on it, not because I had to, but because I genuinely enjoyed it.

In the end, I successfully created **eight Sentinel detection rules** from scratch. Seven of those were tested using the simulation method I described. These rules are now ready to be deployed into the customer's environment, where they'll be monitored and fine-tuned over time, for example, by tightening up whitelisting.

What I enjoyed most was how this project brought everything I'd learned during the internship together. These rules didn't exist in Sentinel before. Now they do and **they'll improve detection coverage and overall protection for the customer.**

All of this was possible thanks to the trust I received from both my team lead and Maarten, who supported and guided me throughout. I'm really grateful for the opportunity. This project only strengthened my motivation to pursue a career in cybersecurity.