

Netcure SOC Project

Realization Report

Baisangur Dudayev
Bachelor Electronics-ICT - Cloud & Cybersecurity

Table of Contents

1. INTRODUCTION	3
2. ANALYSIS – TOOLS AND PLATFORMS USED	4
2.1.1. Security Operations Center	4
2.1.2. Security Information and Event Management (SIEM)	5
2.1.3. Guardian360	5
2.1.4. CyberAlarm	5
2.1.5. ELK Stack (Elasticsearch, Logstash, Kibana) with Grafana	6
2.1.6. Security Incident Response Platform (SIRP)	7
2.1.7. Vagrant	7
2.1.8. Ansible	7
2.1.9. Docker	8
2.1.10. Nginx	8
2.1.11. n8n	9
2.1.12. OpenSSL	9
2.1.13. WSL (Windows Subsystem for Linux) + Visual Studio Code	10
2.1.14. Python	10
2.1.15. Ruby	11
2.1.16. APIs (Application Programming Interfaces)	11
2.1.17. Jinja	11
2.1.18. VPN (Virtual Private Network)	12
2.1.19. SCP (Secure Copy Protocol)	12
3. REALISATIONS	13
3.1 Basic SIEM Configuration	13
3.2 Visualisation & Reporting	19
3.3 Automation Integration	23
3.4 TheHive Integration	25
4. CONCLUSION	27
4.1. Recommendations for future improvements	27
REFERENCE LIST	28
ATTACHEMENTS	29

1. Introduction

This document presents the realisation report of the internship project carried out at **Netcure**, a cybersecurity firm specialising in tailored solutions to protect businesses against cyber threats. **While the internship project was carried out collaboratively by a team of interns, this document specifically highlights my own contributions, technical decisions, and responsibilities within the broader group effort.**

Within **Netcure's SOC environment**, there was a clear need to consolidate **alerts and notifications** originating from different security platforms. The existing situation made it difficult to gain a unified view of threats, follow up on incidents efficiently, and produce consistent reporting. The goal of the internship was to address this challenge by **centralising these alerts**, presenting them in a clear and structured way, and generating the necessary **standardised reports** for ongoing monitoring and analysis.

Based on the **project plan**, the primary objective was to build a **complete SOC from the ground up**. This involved designing, implementing, and optimising a fully functional **Security Information and Event Management (SIEM) solution** with capabilities for **log aggregation, correlation, visualisation, automated reporting, and integration with a Security Incident Response Platform (SIRP)**. The solution needed to consolidate a maximum amount of relevant information from various sources, **automate** as much of the analysis process as possible, and ensure that **incident follow-up** was efficient and well-documented.

The implementation aimed to **improve operational efficiency, reduce manual workloads**, and provide **enhanced visibility** across security events, ensuring faster and more accurate **detection, analysis, and response**. The project required close collaboration with the **Cyberdefense team** to determine the most effective way to integrate data sources, design dashboards, and automate reporting workflows.

This report covers:

- **An analysis** of the selected platforms, tools, and the reasons for their selection.
- **A detailed account** of the realisation phase, including technical decisions and implementations.
- **A conclusion** reflecting on the outcomes, evaluation against objectives, and recommendations for future improvements.

2. Analysis – Tools and Platforms Used

The creation of the **Security Operations Center (SOC)** from the ground up required careful selection of **proven, secure, and well-integrated platforms and tools**. These components had to work together to **collect, process, store, and analyse** large volumes of security data while also supporting **automation** and **structured incident response**.

The **initial phase** of the project involved **evaluating possible tools and technologies** to meet the SOC's requirements. While some choices were predetermined by earlier project work, there was still significant room for technical decisions, particularly in selecting solutions that balanced **performance, scalability, and ease of integration**.

When selecting technologies, the most important decision-making criteria were:

- **Proven reliability** in real-world security operations.
- **Strong community or vendor support** for long-term maintainability.
- **Cost-effectiveness**, preferably through open-source licensing.
- **Ability to integrate** with other SOC components without major compatibility issues.
- **Suitability for scaling** as the SOC expands in the future.

The following tools and platforms are presented with:

1. **What it is** – A clear explanation for both technical and non-technical readers.
 2. **Why chosen** – The reasoning behind the decision, with the **Weighted Ranking Method** applied where meaningful alternatives existed.
 3. **Where/how used** – Concrete examples of its role in the SOC during the internship.
-

2.1.1. Security Operations Center

What it is:

Not a single tool, but a structured environment where people, processes, and technology work together to monitor, detect, and respond to cybersecurity threats in real time.

Why chosen:

The SOC framework was adopted because Netcure needed a centralised, coordinated environment to bring together all security monitoring, incident response, and reporting activities.

Where/how used:

The SOC was built from the ground up during the internship, incorporating the SIEM, SIRP, automation, and reporting components into a single integrated workflow.

2.1.2. Security Information and Event Management (SIEM)

What it is:

A centralised system designed to collect security data from multiple sources, store it, and provide search, analysis, and alerting capabilities.

Why chosen:

A custom SIEM was developed instead of purchasing a commercial solution to give Netcure full control over architecture, configuration, and costs, while meeting all operational requirements.

Where/how used:

Built using the ELK Stack and Grafana, the SIEM became the heart of the SOC, aggregating logs from Guardian360, Cyberalarm, and WatchGuard, normalising them via Logstash, and visualising them through Grafana dashboards.

2.1.3. Guardian360

What it is:

Guardian360 is a **cloud-based security platform** that provides continuous **vulnerability scanning** (multiple scanners for internet-facing assets), **intrusion detection** via *Hacker Alert* canary/sensoring, and **automated reporting** with risk insights and trends. It's delivered as a modular, ISO-certified service and is designed for MSPs/critical infrastructure to give 360° visibility over exposed risks.

Why chosen:

It was one of Netcure's core platforms, already in use with multiple clients, and provided rich API access for retrieving scan results, alerts, and contextual data.

Where/how used:

I wrote extensive Python scripts to integrate Guardian360 into the SIEM. These scripts retrieved assets, alerts, probe data, schedules, and risk trends through API calls, which were then normalised and forwarded to Elasticsearch for storage and analysis.

2.1.4. CyberAlarm

What it is:

SecureMe2 **CyberAlarm** is a **network intrusion detection system (NIDS)** that analyses **network traffic behaviour** to identify suspicious activity. It uses **probes** to observe traffic, provides alarming/notifications, and offers operational features like per-customer notification settings, CSV export, and probe filtering in the alarm view—aimed at helping organisations monitor and act on threats seen on their networks.

Why chosen:

It complemented Guardian360 by supplying raw security logs and alert data, which could be filtered and correlated inside the SIEM.

Where/how used:

I developed scripts to collect logs from CyberAlarm using the API, including handling Elasticsearch's *search_after* function for continuous log collection. I also built a workflow for probe retrieval to monitor whether client probes were online, offline, or recently disconnected.

2.1.5. ELK Stack (Elasticsearch, Logstash, Kibana) with Grafana

What it is:

- **Elasticsearch** – A high-performance search and analytics engine optimised for very large datasets. It stores security logs in a way that allows lightning-fast searches across millions of entries. This was crucial for analysts to quickly investigate incidents, such as finding all logs from a specific IP address within a defined timeframe.
- **Logstash** – A data pipeline tool that receives raw logs from various sources, processes and cleans the data, then sends it to Elasticsearch for storage.
- **Kibana** – A visualisation layer that enables analysts to explore Elasticsearch data through dashboards, charts, and interactive searches.
- **Grafana** – An additional visualisation tool that can integrate with Elasticsearch and other sources, offering advanced, customisable dashboards.

Why chosen:

Weighted Ranking Method comparing ELK Stack with Grafana to Splunk.

Criterion	Weight	ELK+Grafana Score	Splunk Score	ELK Weighted	Splunk Weighted
Cost	25	5	1	125	25
Community support	20	5	5	100	100
Integration flexibility	20	5	4	100	80
Visualisation capabilities	15	4	5	60	75
Documentation	10	5	5	50	50
Scalability	10	5	5	50	50
Total	100			485	380

(Zivanov, 2023; Tan, 2023; Li, 2022)

Conclusion: ELK Stack with Grafana ranked highest due to its cost-effectiveness, scalability, and strong integration capabilities.

Where/how used:

Formed the backbone of the SOC's SIEM. Deployed on the production server to ingest logs from Guardian360, Cyberalarm, and WatchGuard. Logstash processed and normalised logs before Elasticsearch indexed them. Kibana and Grafana visualised both real-time and historical data for security analysts.

2.1.6. Security Incident Response Platform (SIRP)

What it is:

A platform for managing, tracking, and documenting security incidents. It allows analysts to assign tasks, record investigation details, and ensure consistent workflows. In this project, TheHive was selected as the SIRP.

Why chosen:

TheHive offered strong integration with Elasticsearch, was open-source, and had a lightweight deployment footprint compared to commercial options like ServiceNow.

Where/how used:

Deployed alongside Elasticsearch and Nginx, set up so it was ready to create cases from alerts and be used by the Cyberdefense team for incident tracking and resolution.

2.1.7. Vagrant

What it is:

Vagrant is a tool for building and managing reproducible virtual machine environments for development and testing.

Why chosen:

It provided a safe, isolated environment to replicate the SOC setup locally, so deployments could be tested before being applied to the production server.

Where/how used:

I used Vagrant to provision local Linux VMs that mimicked the SOC server. These VMs were later configured with Ansible playbooks, ensuring testing was safe and consistent before deploying to production.

2.1.8. Ansible

What it is:

An automation tool following the Infrastructure as Code (IaC) approach. Instead of manually configuring servers and services, Ansible uses scripts (“playbooks”) to set up entire environments automatically.

Why chosen:

Weighted Ranking Method comparing Ansible to Puppet.

Criterion	Weight	Ansible Score	Puppet Score	Ansible Weighted	Puppet Weighted
Learning curve	25	5	3	125	75
Community support	20	5	4	100	80
Integration with Linux	20	5	4	100	80

Criterion	Weight	Ansible Score	Puppet Score	Ansible Weighted	Puppet Weighted
Documentation	15	5	4	75	60
Cross-platform use	10	4	4	40	40
Performance	10	4	4	40	40
Total	100			480	375

(Odazie, 2025; RedHat, 2022)

Conclusion: Ansible ranked highest due to its ease of learning, strong Linux integration, and powerful automation capabilities.

Where/how used:

Created and tested playbooks locally using WSL, then deployed them to configure Elasticsearch, Kibana, Docker containers, and Nginx in the production SOC environment.

2.1.9. Docker

What it is:

A platform that packages applications and their dependencies into isolated “containers,” ensuring they run identically in any environment.

Why chosen:

Docker was chosen because it allowed services to be isolated from the host environment without installing them directly on the virtual machine. Containers offered a lightweight alternative to running full virtual machines and ensured consistent deployments across environments. Its large community and extensive documentation made it a practical choice for this project.

Where/how used:

Used to containerise services such as Nginx and automation tools, ensuring consistent deployments between development and production.

2.1.10. Nginx

What it is:

Nginx is a high-performance web server, meaning it can deliver web pages, files, and other content to users over the internet or a network. In addition to serving content directly, it can also act as a **reverse proxy**, which means it sits between clients and backend services, forwarding requests to the correct service and then returning the responses to the clients. This setup can improve performance, add security,

Why chosen:

Chosen for its stability, speed, and proven ability to handle secure reverse proxy configurations for SOC services.

Where/how used:

Configured as a reverse proxy to securely route HTTPS traffic to internal SOC tools, including the SIRP and n8n

2.1.11. n8n

What it is:

n8n is a self-hostable, open-source workflow automation platform with a visual editor (“nodes” and “workflows”) that integrates with 400+ services/APIs (n8n, 2025). It’s designed for technical teams who want the flexibility of code with the speed of no-code

Why chosen:

It provided a flexible, visual way to automate SOC notification workflows, integrate email handling, and trigger VocalNotify calls. Its open-source nature made it highly customisable.

Where/how used:

Deployed in Docker alongside Nginx, I built workflows in n8n that processed security alerts, checked conditions such as log severity and frequency, and triggered notifications via Outlook and VocalNotify.

What it is:

VocalNotify is an email-to-voice alerting service: sending an email (subject/body) triggers an automated text-to-speech phone call to one or more recipients, with support for escalation lists and call status feedback.

Why chosen:

It provided a simple, reliable way to turn SOC alerts into immediate phone calls without building a telephony stack, and could be driven directly from n8n by sending an email to a dedicated mailbox.

Where/how used:

Configured as the **final escalation step** in automation: when thresholds/severity rules were met in n8n, an email was sent to the VocalNotify address to trigger a call to the on-duty contact or escalation list.

2.1.12. OpenSSL

What it is:

OpenSSL is a widely used open-source toolkit for implementing cryptographic functions and managing digital certificates. It can generate private keys, create certificate signing requests (CSRs), and issue self-signed certificates. These certificates are used to enable secure connections (HTTPS) between systems by encrypting data in transit and verifying the identity of the communicating parties. OpenSSL also provides tools for encrypting/decrypting files, verifying certificate validity, and converting certificate formats,

making it essential for setting up secure communication channels in networks and web services.

Why chosen:
Needed to create self-signed certificates for secure internal SOC communications without relying on external certificate authorities.

Where/how used:
Generated certificates to enable HTTPS connections between SOC services, including Nginx and TheHive.

2.1.13. WSL (Windows Subsystem for Linux) + Visual Studio Code

What it is:
WSL allows Linux software to run directly on Windows without a separate virtual machine. Visual Studio Code is a cross-platform code editor with strong extension support.

Why chosen:
Enabled development and testing of Linux-only tools like Ansible directly on Windows laptops.

Where/how used:
Configured WSL on development machines for local testing of automation playbooks, with Visual Studio Code as the primary editor.

2.1.14. Python

What it is:
A widely used programming language known for its readability, versatility, and strong support for scripting and automation tasks.

Why chosen:
Weighted Ranking Method comparing Python to Bash.

Criterion	Weight	Python Score	Bash Score	Python Weighted	Bash Weighted
Ecosystem & libraries	20	5	2	100	40
API integration	15	5	2	75	30
Data processing	15	5	2	75	30
Maintainability	15	5	3	75	45
Cross-platform	10	4	3	40	30
Automation support	10	4	4	40	40
Learning curve	10	4	3	40	30
Performance	5	3	4	15	20
Total	100			460	265

(Mahmood, 2023; Alamutu, 2023; Gokani, 2024)

Conclusion: Python outperformed Bash for SOC tasks requiring data parsing, API calls, and report generation.

Where/how used:

Wrote scripts for log retrieval via APIs, data processing, and automated report generation for Guardian360 scans.

2.1.15. Ruby

What it is:

Ruby is a programming language. In this project, it was used specifically within Logstash filter plugins to parse, clean, and transform incoming log data before forwarding it to Elasticsearch.

Why chosen:

Logstash relies on Ruby for custom filtering, and it allowed us to fine-tune log fields so that only relevant data was stored.

Where/how used:

I worked with Ruby code inside Logstash pipelines to normalise and filter logs retrieved from Guardian360 and CyberAlarm, ensuring data consistency before indexing in Elasticsearch.

2.1.16. APIs (Application Programming Interfaces)

What it is:

APIs are defined interfaces that let software applications communicate and share data in a controlled way. They expose specific functions or data without revealing the internal workings of a system, making it easier to integrate different tools, automate tasks, and exchange information reliably.

Why chosen:

Provided a standardised and secure way to retrieve logs and vulnerability data from external platforms like Guardian360, Cyberalarm, and WatchGuard.

Where/how used:

Python scripts used APIs to pull fresh data on a schedule, filter it, and send it into the SIEM without manual intervention.

2.1.17. Jinja

What it is:

Jinja is a templating engine used to create dynamic text files, reports, or web pages by replacing placeholders in a template with actual data at runtime. It supports control structures like loops and conditionals, allowing templates to adapt their output based on

the provided data. This makes it useful for generating consistent, formatted content such as configuration files, HTML pages, or reports from variable input.

Why chosen:

Allowed automated generation of HTML-based reports with consistent formatting.

Where/how used:

Used in Python scripts to generate vulnerability reports, combining scan data with standardised layouts and branding.

2.1.18. VPN (Virtual Private Network)

What it is:

A VPN creates an encrypted communication tunnel over the internet between a device and a remote network. This ensures that data cannot be easily intercepted or read by third parties. VPNs are often used to securely access private networks from outside locations, protect sensitive information on public Wi-Fi, and make remote systems appear as if they are on the same local network.

Why chosen:

Essential for accessing the SOC's internal resources securely from external networks.

Where/how used:

Used daily to connect development computers to the SOC network hosting the SIEM.

2.1.19. SCP (Secure Copy Protocol)

What it is:

A secure file transfer protocol for copying files between systems.

Why chosen:

Guaranteed secure file transfers of configuration files and logs between servers and workstations.

Where/how used:

Transferred generated reports and configuration backups from production servers to local machines for review.

Closure

The combination of these tools created a cohesive SOC ecosystem that met the internship's objectives: centralised log management, automation, visualisation, and incident response. The practical "where/how used" examples demonstrate that each tool was not only theoretically suitable but actively deployed to achieve the SOC's operational goals.

3. Realisations

3.1 Basic SIEM Configuration

The **SIEM environment** was deployed on the production server and prepared to collect and process security logs from **Guardian360** and **CyberAlarm** systems used by clients. Before starting the implementation, I carefully reviewed the **API documentation** of these platforms to understand the structure of the data, authentication requirements, and possible filtering options. Based on this, I wrote a series of **extensive Python scripts**, most of them targeting Guardian360 and some for CyberAlarm. Each script was responsible for making a **separate API call** to retrieve a specific type of log or dataset. This modular approach made it easier to maintain and extend the scripts over time.

Most log sources were configured to be retrieved every **five minutes**, while a smaller number of low-frequency sources were retrieved every **24 hours**. To achieve this, two higher-level **Python scripts** were created: one that executed all scripts requiring a five-minute interval, and another that executed the scripts with a 24-hour interval. These two scripts were then added to the server's **cron scheduler**, ensuring that logs were collected automatically and consistently according to the correct timing requirements.

The raw data retrieved by the scripts was stored locally before being processed further, where I ensured that unnecessary root fields were removed from the JSON log files so that only meaningful and relevant data was stored. **Logstash** was then configured to pick up these stored logs, parse and filter them using custom **Ruby code**, and normalise the fields. After this processing step, the cleaned and structured data was sent to **Elasticsearch** for storage and indexing. This pipeline ensured that logs were not only collected, but also standardised and made searchable in a uniform format that analysts could rely on.

During development, **Vagrant** was used to provision and manage local virtual machines that replicated the production environment. This gave the SOC team an isolated testing setup where deployments could be validated without risk. I contributed to developing **Ansible playbooks** that configured and adjusted these Vagrant-managed VMs, applying **Infrastructure-as-Code** principles. Because certain Ansible commands did not run correctly on native Windows, I installed and configured **Windows Subsystem for Linux (WSL)** on the project laptops. WSL acted as a lightweight Linux environment inside Windows, allowing us to run **Ansible playbooks** seamlessly. I also documented how to install WSL, integrate it with **Visual Studio Code**, and use it with Ansible, so that all interns could work with the same setup. Among the playbooks I worked on were those for setting up **Elasticsearch** and **Kibana**, ensuring they were deployed in a consistent and repeatable way.

As part of the SIEM setup, I also reviewed the **Elastic documentation** on key system configurations for the production environment (Elastic, 2025). I then automated the implementation of selected configurations using **Ansible**. These included:

- Configuring **system resource limits**
- Setting the **TCP retransmission timeout**

- Defining the **JNA temporary directory**

These optimisations ensured that the SIEM environment was robust and performed reliably under load.

For Guardian360 in particular, I implemented a wide range of **API calls** to retrieve security-related data. I developed more scripts than the ones listed below, but these are the ones I had properly documented and could retrieve again when writing this report. They also represent the majority and most important calls for our project:

- Get a list of assets
- Get a list of companies
- Get a list of hacker alert appliances
- Get a list of hacker alert exclusions
- Get a list of hacker alerts
- Get operations on issues
- Get a list of probes
- Get a list of schedules
- Get a list of scanner exclusions
- Get a list of scanner platforms
- Get a live asset counter
- Get all unique issues
- Get historical trends by risk
- Get risks per detection date
- Get risks per scanner
- Get risks per scan object

Figure 1 below shows a section of the Guardian360 'hacker alert' script opened in Visual Studio Code. On the left, the full script list is visible, confirming the wide range of Guardian360 API integrations that were developed. This visual evidence supports the list of API calls described above.

```

1  import requests
2  import logging
3  import json
4
5  # Define the API key and token file path (close to each other for easy copy-pasting)
6  api_key = "REDACTED"
7  token_file_path = "REDACTED" # Change this to your token file path
8
9  # Base URL and endpoint
10 base_url = "REDACTED"
11 endpoint = "REDACTED" # Endpoint to retrieve a list of hacker alerts
12
13 # File Paths for Logging and Output
14 error_log_file = "REDACTED"
15 output_file = "REDACTED" # Output file name
16
17 # Setup logging configuration to log errors to the specified file
18 logging.basicConfig(
19     level=logging.ERROR, # Log errors only
20     format='%(asctime)s - %(levelname)s - %(message)s',
21     handlers=[logging.FileHandler(error_log_file)] # Log errors to GD360_error.log
22 )
23
24 # Function to read the API token from the file
25 def read_api_token(file_path):
26     try:
27         with open(file_path, 'r') as file:
28             token = file.read().strip()
29             logging.info(f"API token successfully read from {file_path}.")
30             return token
31     except Exception as e:
32         logging.error(f"Error reading API token from {file_path}: {e}")
33         raise
34
35 # Read the token from the file
36 api_token = read_api_token(token_file_path)
37
38 # Define headers, including the API key and token for authorization
39 headers = {
40     "Authorization": f"Bearer {api_token}",
41     "Api-Key": api_key,
42     "Content-Type": "application/json"
43 }
44
45 # Function to retrieve the list of hacker alerts
46 def retrieve_hacker_alerts():
47     try:
48         # Define parameters for pagination and filter by acknowledged status
49         params = {
50             "rows": 100, # Number of results per page
51             "page": 3, # Page number (can adjust as needed)
52             "columns[acknowledged]": 0 # 0 for unacknowledged hacker alerts, 1 for acknowledged
53         }
54
55         # Send the GET request to retrieve the list of hacker alerts
56         logging.info(f"Sending GET request to {base_url}{endpoint} with parameters: {params}")
57         response = requests.get(f"{base_url}{endpoint}", headers=headers, params=params)
58
59         # Check if the response is successful (status code 200)

```

Figure 1 - Guardian360 hacker alert script in Visual Studio Code, with the sidebar showing the full set of API scripts developed for Guardian360 integration.

By covering such a broad spectrum of endpoints, the scripts were able to collect not only raw log data, but also contextual and historical information about vulnerabilities, risks, and scanning activity across different clients. This allowed the SIEM to build a much more

complete picture of the security posture of each company connected to the platform.

For Guardian360, I also wrote additional scripts to retrieve **probes from subscribed customers**, which expanded the range of data sources ingested into the SIEM. To make the collected data more suitable for **Logstash**, I ensured that **unnecessary root fields** were removed from the JSON log files, so that only meaningful and relevant data was stored.

For **CyberAlarm**, I developed two main scripts:

- **Script to retrieve logs** – This was the most complex script, as it needed to implement the **search_after functionality**. This required storing the value returned by **Elasticsearch** separately and then retrieving it when the script was executed again. By doing so, log collection could resume from the exact point where it had last stopped, preventing both data duplication and gaps in collection. Debugging and testing this feature was challenging, but I ultimately succeeded in making the script **dynamically continue log retrieval** from the last collected entry. The modifications also ensured that only **level 1–3 logs** from subscribed clients were collected, and that the correct file permissions were automatically assigned to the newly created **log.json** file.
- **Script to retrieve probes** – This script was designed to collect **probes from customers subscribed to Netcure**, extending the amount of useful data that could be ingested into the SIEM. With this data, it became possible to check whether probes were **online, offline, or had recently gone offline**, giving analysts better visibility into the operational status of client infrastructure.

In addition, I implemented **error handling** across all my scripts so that whenever a problem occurred, it was written to dedicated **error log files**. To validate correctness, I tested my scripts in several ways. This included running them under normal conditions, but also with **larger timeframes** in the API calls and then comparing the results with the output from the standard scripts. I also ran the scripts **multiple times for the same timeframe** to verify that the results were consistent and that no duplicate or missing logs occurred.

While setting up the production environment, I also identified and solved an infrastructure issue. The production VM experienced connectivity problems due to a **DNS misconfiguration**. I diagnosed the cause and resolved it by changing the DNS entry in **/etc/resolv.conf** from the localhost IP to **Google's public DNS server**. This change restored proper outbound connectivity and ensured reliable script execution.

Finally, I developed a **Python-based solution for managing log storage in Elasticsearch**. This was achieved through a **custom Python script that interfaced with Elasticsearch's REST API**. The script was scheduled to run **once per week** and automatically deleted outdated documents once index size or document age thresholds were reached, always starting with the **oldest indices first**.

Figure 2 below shows a filtered view of Elasticsearch indices in Kibana. The document counts of 2000, 2001, or 0 demonstrate that the automated deletion script was actively trimming indices according to the configured threshold, confirming that it was functioning as intended.

Elasticsearch indices

[Default settings](#)
[Create a new index](#)

⚠ Enterprise Search has not been configured

The Elastic web crawler is not available without Enterprise Search.

[Review setup guide](#)

Available indices

☒ Show hidden indices
 ☒ Only show crawler indices

Index name	Index health	Docs count	Ingestion name	Ingestion method	Ingestion status	Actions
	● yellow	2001		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	22		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	2000		API	Connected	
	● yellow	0		API	Connected	
	● yellow	2001		API	Connected	

[<](#)
[1](#)
[2](#)
[>](#)

Figure 2 - Elasticsearch indices in Kibana (indices redacted), showing document counts of 2000, 2001, or 0, confirming the automated log deletion script was functioning as intended.

Together with Tosin, I co-decided on a limit of **2,000 documents per index**, which struck a balance between keeping enough recent data available for analysts and ensuring system performance. At the time of implementation, we had around **340 indices**, because the logs were deliberately structured in a very detailed way to make it easier for the dashboard team to pick and choose exactly the indices they needed for visualizations. Due to this level of

granularity, most indices grew slowly — for many, it would take at least **a month and a half** before reaching 2,000 documents. The script itself was also designed to be flexible: adjusting the threshold (for example, from 2,000 to 5,000) or adding new indices could be done with only a minor change.

This log deletion policy was essential for several reasons:

- **Performance** – Elasticsearch performance degrades if indices grow too large, slowing down queries and dashboards.
- **Storage management** – Logs accumulate quickly, and without deletion, disk space would eventually be exhausted.
- **Relevance of data** – Daily SOC work relies mostly on recent logs. Older logs were still backed up, but did not need to stay in Elasticsearch.

The logs were already being **backed up during their creation**, so deletion never meant permanent data loss. Instead, it created a clear split between “**hot**” **data for fast analysis** and “**cold**” **data for long-term retention**. Keeping months of hot-ready logs would not only waste resources, it would also slow down searches and dashboards with large volumes of outdated data that analysts rarely need in daily operations. For rare cases where long-term data is required, the **backup system** remained available, aligning with common SOC practice of separating hot and cold storage.

The solution was tested extensively in **dry-run mode** using sample indices before being deployed in production. Once live, it successfully prevented performance degradation, reduced disk usage, and kept queries fast and efficient — while still ensuring that all historical logs remained securely available through the backup system.

3.2 Visualisation & Reporting

The reporting component of the SOC focused on transforming raw Guardian360 scan data into clear, professional reports that could be used both internally and by clients. These reports were generated in two formats: **interactive HTML reports** for on-screen review and **archive-friendly PDF versions** for distribution and long-term storage.

The reports were based on **two CSV files** provided by Guardian360: the main CSV containing vulnerability scan data, and a second mapping file used to correctly link scan objects to their corresponding **scanner platforms**. The main CSV did not explicitly specify which platform a probe belonged to, but it contained the **scan object field**. Since this field was also present in the mapping file, I could establish the correct connections between platforms and their logs. This mechanism ensured that results were accurately assigned to the right **scanner platform**, which significantly improved the precision of reporting.

I automated the entire reporting workflow by writing a **Python script** that processed the CSV files and populated a custom **HTML template**. The template defined the overall report layout, while **JavaScript** added interactivity and **CSS** handled styling. By combining these technologies, I was able to generate dynamic, visually clear reports that included:

- A **styled cover page** with company branding
- An automatically generated **table of contents**
- **Page numbering** for easier navigation
- Populated **issue descriptions, solutions, technical data, and severity levels**
- **Charts** such as donut graphs and stacked bar charts to visualize risk levels and issue distribution

Figure 3 below shows the automatically generated report cover page. This example is a **redacted version**, where sensitive company details and technical data have been replaced or blacked out. A full redacted report is available for download in my internship portfolio for further reference.



Figure 3 - Automatically generated cover page of the Guardian360 report, including company branding and standardized layout.

In addition to the HTML version, I integrated functionality to automatically generate a **PDF version**. The challenge was ensuring that the PDF retained the look and structure of the HTML report. This required several iterations to fix issues such as **misaligned charts, broken tables, and incorrect page numbering**. Eventually, I achieved a system where the PDF mirrored the HTML version with consistent styling, proper bookmarks, corrected page breaks, and reliable formatting.

The reporting system also included initial **multilingual support**. Static text, such as the report introduction, was predefined in Dutch, French, and English. For dynamic content — including issue descriptions, solutions, and technical details — I integrated the **DeepL API** to handle translations automatically (DeepL, n.d.). This allowed reports to be generated in multiple languages depending on client needs. However, my coach raised concerns that technical IT terminology might not always be translated correctly into Dutch and French, which could cause confusion or inaccuracies in client-facing reports. For this reason, the functionality was later scrapped, although the codebase remained fully functional and could be reactivated in the future.

Throughout development, I regularly **tested the reports** by comparing them against raw Guardian360 logs and reviewing them in both HTML and PDF formats. This ensured that the data was accurate, that formatting was preserved, and that charts correctly reflected key metrics. Feedback from **Lasse** and other colleagues led to further refinements, including improving the cover page design, restructuring the table of contents, and adjusting the layout of tables and charts for readability.

As the system evolved, reports were first generated at the **company level**, then refined to the **scanner platform level** to provide a more meaningful view of infrastructure. I later extended this functionality to allow reports to be generated **per probe**, giving even more flexibility depending on the level of detail required by analysts or clients.

Overall, the reporting system transformed raw CSV vulnerability scan data into **standardized, professional reports**. By automating cover pages, tables of contents, issue listings, and visualizations, it saved significant manual effort while providing clear insights into vulnerabilities and risks. The ability to export reports in both HTML and PDF ensured that they could be used interactively in the SOC as well as distributed externally to clients in a polished, archive-ready format.

To illustrate the scope of the implementation, Figure 4 shows the end of the HTML report template, which reached approximately 1100 lines of code, while Figure 5 shows the end of the Python script that populated the template, which consisted of around 700 lines. The screenshots demonstrate the scale and complexity of the reporting system that was developed during the internship.

```

guardian360 > new > backups > 23-04 16:53 issue page numbers
2    <html lang="en">
580  <body style="margin: 0; padding: 0;font-f
622    <div style="width: 1024px; margin: 0p
990    <script>
1066      function insertPageNumbersForTOC(
1079        // Sort offsets just in case
1080        breakOffsets.sort((a, b) => a
1081
1082        links.forEach(link => {
1083          const targetId = link.get
1084          const targetEl = document
1085
1086          if (targetEl) {
1087            const rect = targetEl.get
1088            const scrollY = window.sc
1089            const yOffset = rect.top
1090
1091            // Count how many breaks
1092            const pageIndex = breakOf
1093            const pageNumber = pageIn
1094
1095            const pageNumberSpan = do
1096            pageNumberSpan.textContent
1097            pageNumberSpan.style.floa
1098            pageNumberSpan.style.marg
1099
1100            pageNumberSpan.style.col
1101            pageNumberSpan.style.font
1102
1103            link.appendChild(pageNumb
1104          }
1105        });
1106      }
1107
1108      window.addEventListener('load
1109
1110
1111
1112    </script>
1113
1114  </body>
1115 </html>
1117

```

Figure 4 - End of the HTML report template, showing total length of 1100 lines.

```

guardian360 > new > backups > 23-04 16:53 issue page numbers, to
663
664
665  output_filename = output_dir / f"
666
667  output_dir.mkdir(parents=True, ex
668
669  # Write the file
670  with open(output_filename, 'w', e
671    f.write(rendered_html)
672
673  # # Generate PDF version of the s
674  html_path_str = str(output_filena
675  pdf_path = output_filename.with_s
676  # Run async Playwright logic to g
677  asyncio.run(html_to_pdf(html_path
678
679  print(f"Saved HTML and PDF reports of
680
681
682  # -----
683  # Debug output (Company-level row counts)
684  # -----
685
686  scanobject_risk_counts = defaultdict(lambda:
687  print("\nCompany Row Counts:")
688  for company, count in company_counts.items():
689    print(f"  {company}: {count} rows")
690
691  print("\nReports generated.")
692  print("Brought to life by Baisangur Dudayev,
693
694
695  # <!-- <p class="client-title"><b>{{ transla
696  #
697    <p class="client-title"><b>{{ t
698  #
699    <p class="client-title"><b>Rappor
700  #
701    <p class="client-title"><b>scan

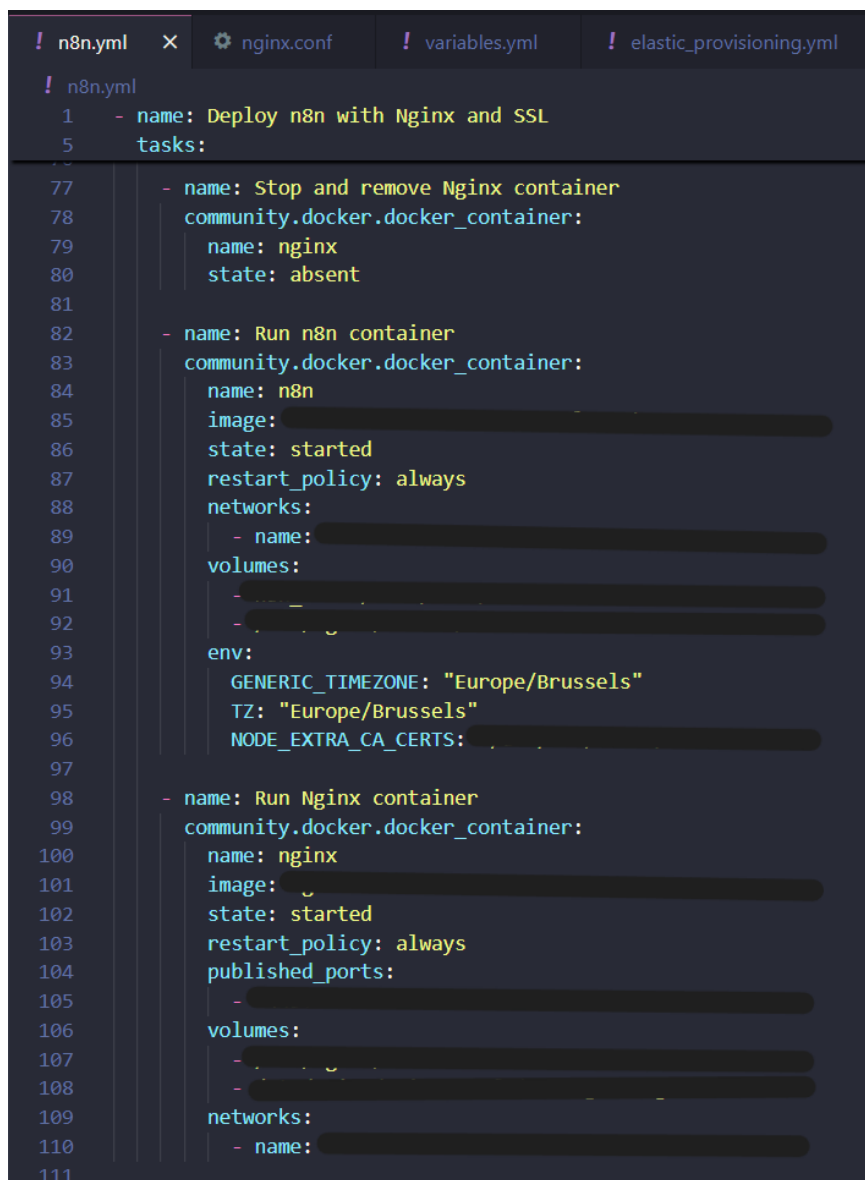
```

Figure 5 - End of the Python script that populated the report template, showing total length of 700 lines.

3.3 Automation Integration

n8n was deployed inside a **Docker container** together with **Nginx** configured as a reverse proxy (n8n, 2025; NGINX, 2025). To ensure secure communication, I implemented **HTTPS connections** using both **self-signed certificates** and certificates already in use by **Elasticsearch**. This guaranteed secure communication across services while avoiding reliance on external certificate authorities for internal traffic. To make the setup reproducible, I also created an **Infrastructure-as-Code (IaC) playbook** with **Ansible** that could deploy the full n8n and Nginx configuration directly to the live server after testing (Red Hat, 2025).

Figure 6 below shows an excerpt of the Ansible playbook I created for deploying n8n (sensitive details redacted). This illustrates the Infrastructure-as-Code approach used during the project, ensuring that the setup could be replicated consistently across environments.

The image shows a screenshot of an Ansible playbook file named 'n8n.yml'. The top of the editor shows tabs for 'n8n.yml', 'nginx.conf', 'variables.yml', and 'elastic_provisioning.yml'. The playbook content is as follows:

```
! n8n.yml
1  - name: Deploy n8n with Nginx and SSL
5    tasks:
77      - name: Stop and remove Nginx container
78        community.docker.docker_container:
79          name: nginx
80          state: absent
81
82      - name: Run n8n container
83        community.docker.docker_container:
84          name: n8n
85          image: [REDACTED]
86          state: started
87          restart_policy: always
88          networks:
89            - name: [REDACTED]
90          volumes:
91            - [REDACTED]
92            - [REDACTED]
93          env:
94            GENERIC_TIMEZONE: "Europe/Brussels"
95            TZ: "Europe/Brussels"
96            NODE_EXTRA_CA_CERTS: [REDACTED]
97
98      - name: Run Nginx container
99        community.docker.docker_container:
100          name: nginx
101          image: [REDACTED]
102          state: started
103          restart_policy: always
104          published_ports:
105            - [REDACTED]
106          volumes:
107            - [REDACTED]
108            - [REDACTED]
109          networks:
110            - name: [REDACTED]
111
```

Figure 6 - Excerpt from the Ansible playbook for n8n deployment (sensitive details redacted), demonstrating Infrastructure-as-Code setup.

Ensuring that n8n was up and running was crucial during this phase, as the work of all team members depended on it.

Once the environment was operational, I began developing **automation workflows** to handle security event notifications. The initial approach used a **Gmail mailbox** that n8n would check for incoming requests. Employees could **stop or restart VocalNotify calls linked to their names** by sending a message to this mailbox — for example, if someone was on vacation, they could pause calls until they returned. To keep the mailbox clean, I wrote a **Google Apps Script** that automatically deleted emails after they had been processed.

Later, based on feedback from management, this original Gmail-based workflow was **migrated to Outlook**. Employees who wanted to adjust their VocalNotify preferences now sent requests to a dedicated **Outlook mailbox** instead. After this, Tosin requested that the workflow be updated again to pull contact information from **Microsoft Lists**, which I successfully implemented. Later, the data source changed once more, and I reconfigured the workflow to retrieve names and escalation rules directly from an **Excel file stored in SharePoint**. These iterations ensured that the notification system matched Netcure's internal tools and processes as they evolved.

The most critical workflow I worked on centered on **CyberAlarm log monitoring**. The general design of this workflow had already been outlined on paper, but **me and the team co-decided on the details**, and I was the one who **implemented and coded it in n8n**. The workflow was built with flexibility in mind:

- **Company selection** – By default, all client companies were included, unless they were explicitly excluded. This ensured that new customers were automatically covered without additional configuration, while exceptions (e.g., clients not subscribed to certain SOC packages) could be excluded.
- **Level 2 alerts (moderate severity)** – For each included company, the workflow retrieved level 2 alerts from CyberAlarm. If the number of level 2 logs exceeded a defined threshold, an email was automatically generated and sent to the **VocalNotify mailbox** at Netcure. This email triggered VocalNotify to call the appropriate employee, ensuring that critical issues were escalated quickly. During development, I tested this successfully using another Netcure Outlook account, as the production VocalNotify email was not yet available to me.
- **Level 1 alerts (highest severity)** – Level 1 alerts were handled with additional care to avoid false alarms. When such an alert appeared, the workflow waited a predefined amount of time to give analysts an opportunity to review the logs manually. After the waiting period, the workflow re-checked the same logs to see if an analyst had already marked the event as **reviewed**. If the alert was flagged as reviewed, no further action was taken. If it was not marked, an email was sent to the VocalNotify mailbox to trigger a phone call.

Figure 7 below shows the n8n workflow for CyberAlarm log monitoring. The visual workflow illustrates how Level 1 and Level 2 alerts were filtered, thresholds applied, and escalation handled through VocalNotify.

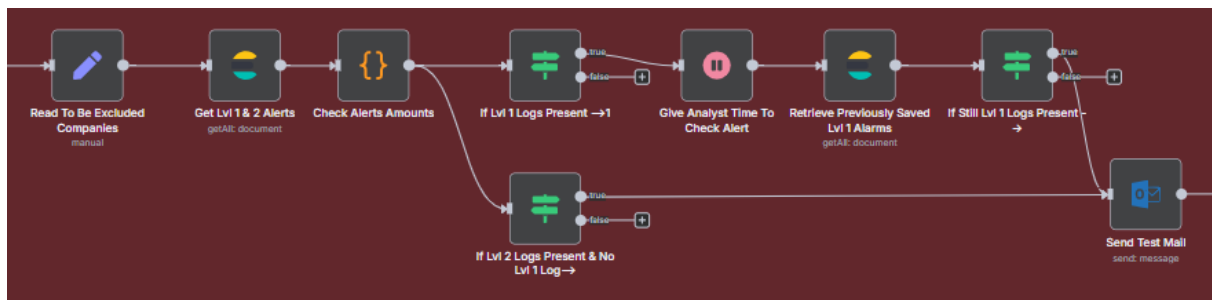


Figure 7 - n8n workflow for CyberAlarm log monitoring, showing the automated filtering of Level 1 and Level 2 alerts and the escalation to VocalNotify.

The wait period for Level 1 alerts was introduced deliberately, for several reasons:

- **Operational stability** – Preventing unnecessary disruption from immediate calls, especially during off-hours.
- **Consistency with SOC practice** – SOC teams often apply a short triage period before escalation to reduce alert fatigue.
- **Better prioritization** – Analysts can quickly check if the alert is part of a larger incident or if another event is more urgent.
- **Give analysts control** – Analysts can mark alerts as “reviewed” if they are already handling them, preventing redundant escalation.
- **Avoid duplicate escalation** – Ensures that multiple analysts aren’t unnecessarily mobilized for the same incident.
- **Reduce false positives** – Some Level 1 alerts may be triggered by unusual but harmless events; the wait period allows time to filter these out.

This system allowed Netcure to balance **speed and accuracy** in incident response: critical alerts were escalated quickly when needed, but analysts retained the ability to intervene and prevent unnecessary VocalNotify calls.

The combination of **Dockerized services, secured reverse proxy connections, and n8n automation workflows** created a reliable automation layer within the SOC. By connecting CyberAlarm to VocalNotify with flexible escalation rules, the SOC team was able to reduce manual workloads, shorten response times, and ensure that important alerts were always acted upon without overwhelming employees with false alarms.

3.4 TheHive Integration

To provide the SOC with a structured incident response process, I worked on the installation and integration of **TheHive**, an open-source Security Incident Response Platform (SIRP). Before deployment, I first studied TheHive in detail by completing online labs (such as **TryHackMe rooms**) and testing a **local installation on my laptop**. This allowed me to explore its features, understand its requirements, and evaluate how it could best fit into Netcure’s SOC environment.

For production, I created an **Ansible playbook** to automate TheHive installation, following the Infrastructure-as-Code approach used throughout the project. A challenge in this phase

was that TheHive required **Nginx and Elasticsearch** as dependencies. Since both services were already running in the SOC environment for other tools, I configured TheHive to reuse these existing containers instead of deploying new ones. This required updating the connection settings so TheHive could properly link to the **existing Elasticsearch cluster**, which it uses as its backend for storing and querying cases. In parallel, it was connected to the SOC's **Nginx service**, which acted as a reverse proxy to provide secure access over HTTPS in line with how the other SOC tools were exposed.

During testing, I initially ran into compatibility issues with the playbook, so I switched to a **manual CLI installation**, applying configuration tweaks until the service ran correctly. With support from the team, I was able to complete the setup and confirm that TheHive was running stably in the environment.

Figure 8 below shows the login page of TheHive after successful installation. Although simple, this screenshot confirms that TheHive was up and running, accessible through the SOC's environment, and integrated with the existing Elasticsearch and Nginx services. Sensitive information has been redacted.

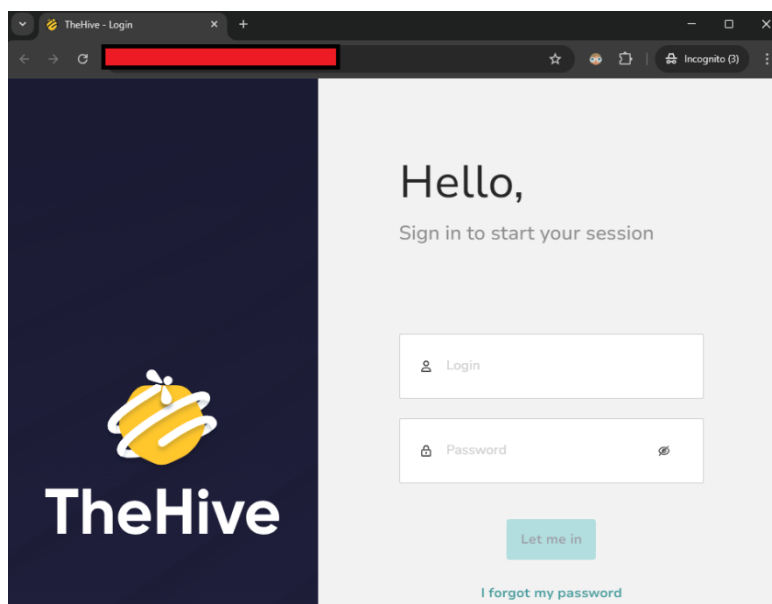


Figure 8 - TheHive login page after successful installation, confirming that the service was accessible and integrated into the SOC

With this integration, the SOC gained a platform for **centralized incident management**. TheHive enabled analysts to open, assign, and track cases directly linked to security alerts, ensuring that investigations were documented in a structured way and making collaboration between team members easier.

Completing TheHive's integration was an important **final milestone of my internship**, as it represented the last major deliverable of the SOC project. It closed the loop between log collection, automation, and incident response, providing Netcure with a more complete and professional SOC environment.

4. Conclusion

This internship resulted in the creation of a functioning **Security Operations Center (SOC) environment** for Netcure, designed to centralise log collection, automate key workflows, and strengthen incident response capabilities.

Through the development of **extensive Python scripts**, security data from **Guardian360** and **CyberAlarm** was integrated into the SOC's **SIEM**. These scripts ensured consistent retrieval of logs and contextual data, applied filtering rules, and prepared the output for ingestion into **Logstash**. With the addition of **Ruby filters**, the data was cleaned and normalised before being stored in **Elasticsearch**. To guarantee long-term performance, I also implemented a weekly **log deletion script** that trimmed indices beyond set thresholds while maintaining secure backups for historical reference.

Automation formed a central part of the project. Using **n8n** deployed in Docker with **Nginx** as a reverse proxy, I created workflows to handle alert notifications, integrate with **Outlook mailboxes**, and trigger **VocalNotify calls** when escalation thresholds were met. These automations reduced manual workload, ensured timely responses, and gave analysts control to pause or restart notifications when needed.

The **reporting system** transformed Guardian360 vulnerability scan data into professional **HTML and PDF reports**, automatically generating cover pages, tables of contents, issue descriptions, and charts. This standardised client communication and reduced the manual effort normally required to prepare detailed reports.

Finally, the deployment of **TheHive** introduced a structured **incident response platform** integrated with Elasticsearch and secured through Nginx. This provided a centralised case management system where investigations could be tracked and documented consistently, ensuring a more efficient and auditable workflow for the Cyberdefense team.

Overall, the project achieved its key objectives:

- Delivering a **custom SIEM** tailored to Netcure's needs.
- Implementing **automation workflows** that balanced speed and accuracy in incident escalation.
- Providing **reporting tools** for both internal use and client-facing communication.
- Deploying a **SIRP (TheHive)** for structured incident management.

The work also underscored the importance of **collaboration** with fellow interns and guidance from Tosin, whose coaching was essential in overcoming technical and organisational challenges.

4.1. Recommendations for future improvements

- **Dashboard optimisation:** Refining Kibana and Grafana dashboards for greater usability and analyst efficiency.
- **Workflow resilience:** Enhancing n8n automations with more robust error handling and escalation logic.
- **Scaling Elasticsearch:** Monitoring system growth to adjust thresholds and maintain performance as the client base expands.

REFERENCE LIST

- Alamutu, H. (2023, April 8). *Bash vs Python Scripting: A Simple Practical Guide - DEV Community*. Opgehaald van dev.to: <https://dev.to/husseinalamutu/bash-vs-python-scripting-a-simple-practical-guide-16in>
- DeepL. (sd). *Your first API request - DeepL Documentation*. Opgehaald van developers.deepl.com: <https://developers.deepl.com/docs/getting-started/your-first-api-request>
- Elastic. (2025). *Important system configuration | Elastic Docs*. Opgeroepen op 2025, van [www.elastic.co: https://www.elastic.co/docs/deploy-manage/deploy/self-managed/important-system-configuration](https://www.elastic.co/docs/deploy-manage/deploy/self-managed/important-system-configuration)
- Gokani, M. (2024, October 11). *Bash vs Python for automating repetitive tasks | LinkedIn*. Opgehaald van [www.linkedin.com: https://www.linkedin.com/pulse/bash-vs-python-automating-repetitive-tasks-mirav-gokani-andze/](https://www.linkedin.com/pulse/bash-vs-python-automating-repetitive-tasks-mirav-gokani-andze/)
- Li, L. (2022, November 18). *Dashboard Design: Visualization Choices and Configurations | Splunk*. Opgehaald van [www.splunk.com: https://www.splunk.com/en_us/blog/tips-and-tricks/dashboard-design-visualization-choices-and-configurations-part-1.html](https://www.splunk.com/en_us/blog/tips-and-tricks/dashboard-design-visualization-choices-and-configurations-part-1.html)
- Mahmood, S. (2023, June 19). *Bash Scripting vs Python: Choosing the Right Language for Automation*. Opgehaald van [nextdoorsec.com: https://nextdoorsec.com/bash-scripting-vs-python/](https://nextdoorsec.com/bash-scripting-vs-python/)
- n8n. (2025). *Docker | n8n Docs*. Opgeroepen op 2025, van [docs.n8n.io: https://docs.n8n.io/hosting/installation/docker/](https://docs.n8n.io/hosting/installation/docker/)
- n8n. (2025). *n8n-io/n8n: Fair-code workflow automation platform with native AI capabilities. Combine visual building with custom code, self-host or cloud, 400+ integrations*. Opgeroepen op 2025, van [github.com: https://github.com/n8n-io/n8n](https://github.com/n8n-io/n8n)
- NGINX. (2025). *NGINX Reverse Proxy | NGINX Documentation*. Opgeroepen op 2025, van [docs.nginx.com: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/](https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/)
- Odazie, D. (2025, June 25). *Puppet vs Ansible: Key Differences Explained*. Opgehaald van [spacelift.io: https://spacelift.io/blog/puppet-vs-ansible](https://spacelift.io/blog/puppet-vs-ansible)
- Red Hat. (2025). *Ansible playbooks - Ansible Community Documentation*. Opgeroepen op 2025, van [docs.ansible.com: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html#playbook-syntax](https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html#playbook-syntax)
- RedHat. (2022, November 28). *Ansible vs Puppet: What you need to know*. Opgehaald van [www.redhat.com: https://www.redhat.com/en/topics/automation/ansible-vs-puppet](https://www.redhat.com/en/topics/automation/ansible-vs-puppet)
- Tan, M. (2023, July 14). *Top 10 Grafana features you need to know about*. Opgehaald van [grafana.com: https://grafana.com/blog/2023/07/14/celebrating-grafana-10-top-10-grafana-features-you-need-to-know-about/](https://grafana.com/blog/2023/07/14/celebrating-grafana-10-top-10-grafana-features-you-need-to-know-about/)
- Zivanov, S. (2023, June 29). *ELK Stack vs Splunk: Ultimate Comparison*. Opgehaald van [phoenixnap.com: https://phoenixnap.com/kb/elk-stack-vs-splunk](https://phoenixnap.com/kb/elk-stack-vs-splunk)

ATTACHEMENTS

Figure 1 - Guardian360 hacker alert script in Visual Studio Code, with the sidebar showing the full set of API scripts developed for Guardian360 integration. _____	15
Figure 2 - Elasticsearch indices in Kibana (indices redacted), showing document counts of 2000, 2001, or 0, confirming the automated log deletion script was functioning as intended. _____	17
Figure 3 - Automatically generated cover page of the Guardian360 report, including company branding and standardized layout. _____	20
Figure 4 - End of the HTML report template, showing total length of 1100 lines. _____	22
Figure 5 - End of the Python script that populated the report template, showing total length of 700 lines. _____	22
Figure 6 - Excerpt from the Ansible playbook for n8n deployment (sensitive details redacted), demonstrating Infrastructure-as-Code setup. _____	23
Figure 7 - n8n workflow for CyberAlarm log monitoring, showing the automated filtering of Level 1 and Level 2 alerts and the escalation to VocalNotify. _____	25
Figure 8 - TheHive login page after successful installation, confirming that the service was accessible and integrated into the SOC _____	26