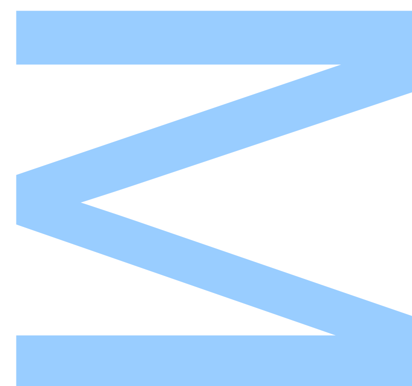
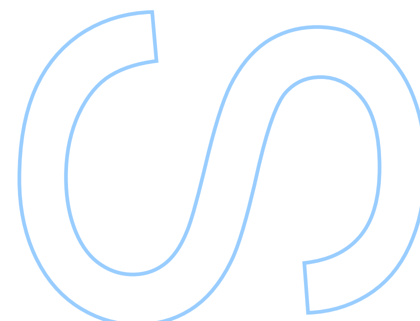
Briareos - A Modular Framework for Elastic Intrusion Detection and Prevention

André Martins Carrilho Costa Baptista
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2017

Orientador
Rolando da Silva Martins, Faculdade de Ciências da Universidade do Porto

Coorientador
Luís Filipe Coelho Antunes, Faculdade de Ciências da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas
pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/________

Dedicado a

César, Dique e Mariana

Abstract

Current intrusion detection systems are mainly based on signature detection running on top of highly optimized parallel engines. While other approaches exist, there is no unified intrusion detection architecture that is able to support them under a unified framework. Given the highly dynamical nature of networked attacks, there is a growing shift to a multi-disciplinary approach as a way to uncover novel algorithms, systems and techniques, including but not limited to, symbolic execution, machine learning and graph mining.

We propose a new intrusion detection and prevention approach capable of preventing unknown attacks by correlating network traffic with the operating system behavior. We implemented these concepts in a module extensible framework called *Briareos*. The *Briareos* Host Component is composed of pipelines for traffic processing, whose nodes contain modules. It supports multiple processing modes: inline, parallel and distributed. We have implemented a distributed system capable of performing heavy tasks, which can be a plus for detecting unknown attack vectors. The distributed system is composed of multiple clusters of workers and a broker that receive tasks using a Least Recently Used (LRU) queue. We also created a worker manager to automatically start or stop worker instances according to workload metrics.

Each processing mode of the *Briareos* Distrbuted System has advantages and disadvantages in terms of transfer rate. The distributed mode is the fastest mode in both low-processing rate and high-processing rate scenarios. The inline mode was the slowest mode, since it offers prevention and not just detection. The *Briareos* Distributed System was capable of adapting to workload changes. We achieved a superlinear speedup while comparing the performance with an elastic number of workers and a fixed number of workers. We also have built modules for unknown attack detection and experimentally tested a subset of vulnerable binaries with real-world vulnerabilities from Capture The Flag (CTF) security competitions. We detected and prevented 100% of the

exploitation attempts with a 95% confidence interval of [80%-100%]. *Briareos* is a disruptive framework that changes the way intrusion detection and prevention is currently performed. Users are given full control over the inspection processing flow.

Resumo

Os sistemas de detecção de intrusões actuais baseiam-se principalmente em técnicas de detecção através de assinaturas. Estes sistemas usam motores de processamento paralelos extremamente optimizados. Embora existam outras abordagens, não existe actualmente uma arquitectura de detecção de intrusões que seja capaz de suportar estes sistemas numa *framework* unificada. Dada a natureza dinâmica de ataques realizados ao nível da rede, existe uma necessidade crescente de usar uma abordagem multi-disciplinar como ponto de partida para descobrir novos algoritmos, sistemas e técnicas, includindo execução simbólica, *machine learning* e *graph mining*.

Propomos uma nova abordagem para detecção e prevensão de intrusões, capaz de prevenir ataques desconhecidos através da correlação entre o tráfego de rede e o comportamento do sistema operativo. Implementámos estes conceitos numa *framework* extensível chamada *Briareos*. O *Briareos Host Component* é composto por *pipelines* de processamento de tráfego, cujos nós contêm módulos. Esta *framework* suporta vários modos de processamento: *inline*, paralelo e distribuído. Implementámos um sistema distribuído para tarefas que necessitem de processamento elevado, o que é uma vantagem em termos de detecção de vectores ataques desconhecidos. O sistema distribuído é composto por vários *clusters* com múltiplos workers e um *broker* que recebe tarefas através de uma LRU *queue*. Criámos também um *worker manager* que permite lançar ou remover automaticamente instâncias de *workers* de acordo com determinadas métricas.

Cada modo de processamento do *Briareos Distributed System* possui vantagens e desvantagens em termos de taxa de transferência. O modo distribuído foi o mais rápido em dois cenários de testes diferentes: processamento reduzido e elevado. O modo *inline* foi o mais lento, o que era previsível porque oference a capacidade prevenção e não apenas detecção. O sistema distribuído desenvolvido foi capaz de se adaptar automaticamente a mudanças no *workload*. Através da introdução desta funcionalidade elástica, conseguimos obter um *speedup* super-linear,

comparandamente a um cenário em que o número de instâncias é fixo. Desenvolvemos módulos para detecção de ataques desconhecidos e testámos experimentalmente um sub-conjunto de programas vulneráveis de competições CTF, com vulnerabilidades presentes actualmente em software. Detectámos e prevenimos com sucesso 100% das tentativas de *exploitation* com um intervalo de confiança de 95% de [80% - 100%]. O *Briareos* revelou ser uma *framework* inovadora que muda a forma como a detecção e prevenção de instrusões é realizada actualmente. Os utilizadores têm controlo total sobre o fluxo de processamente realizado.

Acknowledgments

I would like to thank my thesis advisors in the first place, Rolando Martins and Luís Antunes, for their work, help and suggestions. A very special thanks to Luís Azevedo Maia, my unofficial co-advisor, for guidance and recommendations. I would like to express my gratitude to all my friends and colleagues, including my CTF team, xSTF, for all the support.

I am also very grateful to my family, especially my parents, for the unconditional encouragement and motivation. Thank you, Patrícia, for your amazing company and support through all these years.

Contents

Abstract 5

Resumo 7

Acknowledgments 9

List of Tables 15

List of Figures 18

Listings 20

Acronyms 21

1 Introduction 1

2 State of The Art 3

2.1 Intrusion Detection Systems 3

2.1.1 Network-based Intrusion Detection Systems 3

2.1.1.1 Snort 4

2.1.1.2 Suricata 5

2.1.1.3 Bro IDS 7

2.1.2 Host-based Intrusion Detection Systems 9

2.1.2.1 OSSEC 9

2.1.2.2 Tripwire Open Source 11

2.1.3 Host-based vs Network-based Intrusion Detection Systems 12

2.2 Distributed Systems 13

2.2.1 Message-oriented Middleware 14

2.2.2 ZeroMQ 14

2.3 Exploit Detection 16

2.3.1 Memory Corruption 17

2.3.1.1 Background 17

2.3.1.2 Memory Corruption Mitigations 18

2.3.1.3 Modern Exploitation Techniques 24

3 Architecture **27**

3.1 Overview 27

3.2 Host Component 28

3.2.1 Loading Phase 28

3.2.1.1 Processing Engine 29

3.2.1.2 Pipelines 30

3.2.1.3 Network Interception 31

3.2.2 Processing Phase 31

3.2.3 Modules 33

3.3 Intelligence Sharing 34

3.4 Distributed System 34

4 Implementation **39**

4.1 Project Overview 39

4.1.1 Instructions 39

4.2 Host Component 43

4.2.1 Processing Engine 44

4.2.2 Network Interceptor 48

4.2.3 Modules 50

4.3 Distributed System 51

4.3.1 ZClient 52

4.3.2 ZBroker 53

4.3.3 ZCluster 57

4.3.4 ZWorker 60

5 Results **63**

5.1 Preliminary Results 63

5.2 Performance Analysis 66

5.2.1 Setup 66

5.2.2 Host Component 67

5.2.2.1 Low Processing Rate 67

5.2.2.2 High Processing Rate 68

5.2.3 Distributed System 69

5.3 Exploit Detection Results 72

6 Conclusion **75**

6.1 Future Work 76

References **77**

List of Tables

3.1 Inline Processing vs Parallel Processing. 29

5.1 Workers per Cluster used for Distributed Offload Performance Analysis. 70

5.2 Exploit Detection Results. 73

List of Figures

2.1 Snort Architecture. 5

2.2 Suricata Architecture. 6

2.3 Bro Architecture. 8

2.4 OSSEC Architecture. 10

2.5 Tripwire Flowchart. 11

2.6 Distributed Systems Overview. 13

2.7 ZeroMQ: Basic Message Patterns. 15

2.8 Buffer Overflow in the Stack. 18

2.9 Use After Free. 18

2.10 Data Execution Prevention. 19

2.11 Stack Canary. 20

2.12 Memory Space Mapping with ASLR Disabled. 21

2.13 Memory Space Mapping with ASLR Enabled. 21

2.14 Checksec. 24

3.1 Briareos Architecture. 28

3.2 Processing Modes. 29

3.3 Example: Input Pipeline of a Web Server. 32
3.4 Intelligence Sharing. 34
3.5 Distributed Offloading Architecture. 35

4.1 Class Diagram. 40
4.2 Running the ZBroker. 42
4.3 Running the ZCluster. 42
4.4 Running the BHC. 42
4.5 Host Component Class Diagram. 43
4.6 Processing Engine Class Diagram. 45
4.7 ZClient Class Diagram. 52
4.8 ZBroker Class Diagram. 54
4.9 ZCluster Class Diagram. 57
4.10 ZWorker Class Diagram. 61

5.1 Setup for BHC Performance Analysis. 67
5.2 Web Server Transfer Rate - Low Processing Rate. 68
5.3 Web Server Transfer Rate - High Processing Rate. 69
5.4 Distributed System Elasticity. 71
5.5 Distributed Offload Performance (Fixed Workers). 71

Listings

2.1 Publisher-subscriber: subscriber.py 15

2.2 Publisher-subscriber: publisher.py 16

2.3 Buffer Overflow Example. 23

2.4 Assembly Code: FORTIFY_SOURCE Disabled. 23

2.5 Assembly Code: FORTIFY_SOURCE Enabled. 23

4.1 Script: install-requirements.sh 41

4.2 Script: build-zworker-docker-image.sh 41

4.3 How To Run Briareos. 41

4.4 Example: BHC Configuration File. 43

4.5 Example: Pipeline Configuration File. 44

4.6 Connecting Functions. 46

4.7 Function: Pipeline.run() 47

4.8 Pipeline Configuration File. 48

4.9 Iptables Rule Matching the Previous Pipeline. 48

4.10 Function: Interceptor.configure_iptables() 49

4.11 Function: Interceptor.handler() 49

4.12 Module Example. 50

4.13 ZeroMQ: ZClient. 52

4.14 Example: ZBroker Configuration File. 53

4.15 ZeroMQ: ZBroker. 54

4.16 ZeroMQ: Worker Manager. 55

4.17 Example: ZCluster Configuration File. 57

4.18 ZeroMQ: ZCluster. 58

4.19 Usage calculation with the Docker Stats API. 59

4.20 Example: ZWorker Configuration File. 61

4.21 ZeroMQ: ZWorker. 62

5.1 Simple Web Application Firewall Configuration File. 63

5.2 Module: `app_data_filter.py` 64

5.3 Module: `simple_http_parser.py` 64

5.4 Module: `generic_url_exploit_detector.py` 65

5.5 Testing Pipeline for SQL and XSS. 66

Acronyms

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

ASLR Address Space Layout Randomization

BDS Briareos Distributed System

BFS Breadth-first Search

BFS Breadth-first Search

BHC Briareos Host Component

BIP Briareos Input Pipeline

BMS Briareos Manager Server

BOP Briareos Output Pipeline

CE Community Edition

CFI Control Flow Integrity

CPU Central Processing Unit

CTF Capture The Flag

CVE Common Vulnerabilities and Exposures

DAQ Data Acquisition Library

DEP Data Execution Prevention

DNS Domain Name System

DOS Denial Of Service

DPI Deep Packet Inspection

ELF Executable and Linkable Format

FTP File Transfer Protocol

GCC GNU Compiler Collection

GOT Global Offset Table

HIDS Host-based Intrusion Detection System

HTTP Hypertext Transfer Protocol

HTTPS Hyper Text Transfer Protocol Secure

ICSI International Computer Science Institute

IDPS Intrusion Detection and Prevention System

IDS Intrusion Detection System

IPC Inter-process Communication

IPS Intrusion Prevention System

IRC Internet Relay Chat

JOP Jump-oriented Programming

LRU Least Recently Used

MOM Message-oriented Middleware

NIDS Network-based Intrusion Detection System

NOP No-operation

NSM Network security monitor

OISF Open Information Security Foundation

ORB Object Request Broker

OS Operating System

OSI Open Systems Interconnection

PCAP Packet Capture

PGM Pragmatic General Multicast

PID Process Identification Number

PIE Position-independent executable

PLT Procedure Linkage Table

RAM Random-access Memory

RAP Reuse Attack Protector

RCE Remote Code Execution

ROP Return-oriented Programming

RPC Remote Procedure Call

SIEM Security Information and Event Management

SMTP Simple Mail Transfer Protocol

SQL Structured Query Language

SSH Secure Shell

SSL Secure Sockets Layer

TCP Transmission Control Protocol

UAF Use After Free

XML Extensible Markup Language

XSS Cross Site Scripting

Chapter 1

Introduction

Briareos is a modular framework capable of performing intrusion detection and prevention. It supports both inline and parallel processing modes, for intrusion prevention and detection, respectively. *Briareos* makes possible the detection of both known and unknown attack vectors, since it provides a framework capable of correlating network traffic with the behavior of the operating system. It is capable of creating new rules based on an unknown malicious input that reaches a certain server, by inspecting the correspondent output packets and also based on the correlation of the operating system behavior with that same input. The level of security is adjustable with the objective of giving system administrators trade-off control between performance and the level of security intended for a certain host. It offers users the possibility of building new modules and creating processing pipelines, allowing much more complex procedures than traditional rules. Users can define the processing flow and use any function in order to perform complex tasks, such as machine learning. Heavy tasks, such as packet mining, should be performed in parallel mode. Modules can analyze the traffic in many different ways, such as inspecting the behavior of a given service in order to classify a packet as malicious. These modules allow networks and systems to be increasingly secure over time since they are able to detect unknown attacks using a module extensible framework to prevent complex intrusions.

Although some intrusion detection systems can use multiple rules and scripting languages to detect complex attacks, there is no way to effectively prevent unknown attacks. Also, users are not able to dissect packets in an easy way and they are not able to control how the packets are processed. Many current solutions do not guarantee the privacy of communications inside

networks since traffic cannot be encrypted while performing Deep Packet Inspection (DPI).

Briareos is a modular extensible framework built for Linux using the Python programming language aiming to give users total control of the processing flow that is performed over network traffic. Users can extend Python modules with C and C++ for more performance. The traffic is processed in pipelines that can be easily built and imported. The security level of *Briareos* is adjustable in order to protect both critical and non-critical systems. Encrypted traffic present in some protocols, such as Hyper Text Transfer Protocol Secure (HTTPS), can be decoded by the host itself and then inspected without exposing sensitive information in the network. The ability to give inspection capabilities to the host, which is the primary target, is a plus for detecting unknown attacks and thus generating new detection patterns automatically. *Briareos* includes the *Briareos* Distributed System for heavy tasks, such as intensive data mining over network traffic. Hosts send tasks to a broker, which is therefore responsible for the fair distribution of the workload between workers across one or more clusters. The broker includes a worker manager capable of starting or removing worker instances according to workload metrics, which allows the *Briareos* Distributed System to be elastic.

Chapter 2

State of The Art

2.1 Intrusion Detection Systems

An Intrusion Detection System (IDS) is capable of protecting a network or a computer system by monitoring traffic and interactions at different levels of abstraction. The main goal is to detect malicious activities in the system that should be protected, using a layer of defense that monitors and protects the network and the hosts. An Intrusion Detection and Prevention System (IDPS) has the ability to block these attacks before they reach the destination, while an IDS, by definition, just detects malicious patterns and logs incidents. They both differ from a simple firewall, which usually just inspects the packet headers, but not the payloads, and enforces policies according to given ports, addresses and protocols.

There are two main types of intrusion detection systems: network-based and host-based. Network-based intrusion detection systems are widely used for protecting networks by inspecting and filtering traffic. A host-based IDS solution can offer complementary protection at the operating system-level by monitoring it is resources and collecting events.

2.1.1 Network-based Intrusion Detection Systems

A Network-based Intrusion Detection System (NIDS) monitors and logs all the packets inside a network, searching for known events. These systems are capable of disassembling packets, parsing headers and payload data and identifying protocols correctly without taking the source and

destination port into account. Most of them are able to block intrusions by identifying signatures in the traffic and then rejecting or allowing packets according to certain rules.

In this section, we cover three well-known systems of this type: Snort, Suricata and Bro. Those are excellent solutions for detecting intrusions and complex attacks in networks.

2.1.1.1 Snort

Snort is a powerful, lightweight, cross-platform and open source IDPS that was created in 1998 by Martin Roesch and is now maintained by Sourcefire [37] that was founded by him and acquired by Cisco [40] in 2013. Sourcefire is one of the leaders in this market today. In addition to other solutions, like malware detection appliances, Sourcefire sells rule subscriptions and tools, to improve the security offered by Snort and to make its use an easier task, respectively [2].

Snort is capable of both detecting and preventing intrusions by implementing a rule-driven language which can be used to combine protocol detection, signature-based inspection and anomaly-based audit methods. Like other IDPS solutions, it performs real-time traffic analysis and logging in networks in order to prevent and detect intrusions. However, it can be configured to perform only certain tasks of one or more modes of operation. The three main modes are sniffing, packet logging, and network intrusion detection [48]. While in detection mode, the traffic passes through a set of rules which can trigger actions.

The architecture of Snort contains four main components: sniffing, preprocessing, detection and alerting/logging. It uses Data Acquisition Library (DAQ) [47], an abstraction layer that replaces direct calls to *libpcap* functions, which is useful to run Snort in many different types of software interfaces and hardware. There are different types and modes of packet capture that can be performed with DAQ, such as passive and inline.

According to Roesch (1999), Snort is focused on performance, simplicity, and flexibility [31]. The configuration is loaded, rules are parsed and data structures are generated before initialization of the sniffer module. Roesch work also describes the packet decoder (preprocessing phase), the detection engine and logging/alerting system. The packet decoder operates from the data link layer through the transport layer and the application layer. Snort is very efficient since it preserves packet data pointers for the detection phase. In order to have an efficient detection

phase, Snort generates a two dimensional linked list from the rules, containing chain headers and multiple chain options for each chain header. A chain header contains information about the source and destination IP addresses and ports, while a chain of options contains information about the payload and flags. If there are multiple rules in use for a given protocol, this is a good optimization since it improves the detection performance because protocols mostly share the same ports. Rules can trigger actions, such as alert, log, drop and reject.

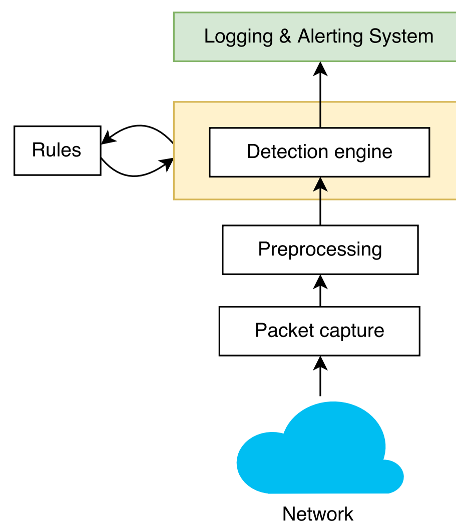


Figure 2.1: Snort Architecture.

Snort sells rule subscriptions for both personal and business use, including the ability to submit false positives and false negatives and receiving new rules immediately upon release. It is also important to mention that there are also free and commercial third party tools available to interact with Snort and other IDPS solutions, supporting automatic updates for rules.

2.1.1.2 Suricata

The first stable version of Suricata was released in 2010 and it was developed by the Open Information Security Foundation (OISF). It is a well-known open source IDPS [39]. In addition to real-time Network security monitor (NSM), intrusion detection and offline Packet Capture (PCAP) processing, it implements inline intrusion prevention, i.e., it is capable of both detecting and preventing known attacks using different mechanisms such as anomaly detection or signature-based

filtering and dropping the packets before they reach the destination.

Suricata inspects network packets according to certain rules and signature checking procedures but it also took a step beyond these simple approaches by supporting LUA scripting in order to detect complex intrusions by defining multiple rules [32]. Suricata also moved up in the Open Systems Interconnection (OSI) model, since it is capable of analyzing and interpreting application layer traffic. This new feature can be used to perform advanced Hypertext Transfer Protocol (HTTP) processing or even detect web server intrusions. It does not just log packets, it is possible to extract potentially malicious executables, Secure Sockets Layer (SSL) certificates, requests or queries, file signatures and other objects by automatically detecting protocols, regardless of the destination port [38]. Among other features, it supports flow tracking, both IPv4 and IPv6, GeoIP, IP reputation and Domain Name System (DNS) parsing [8].

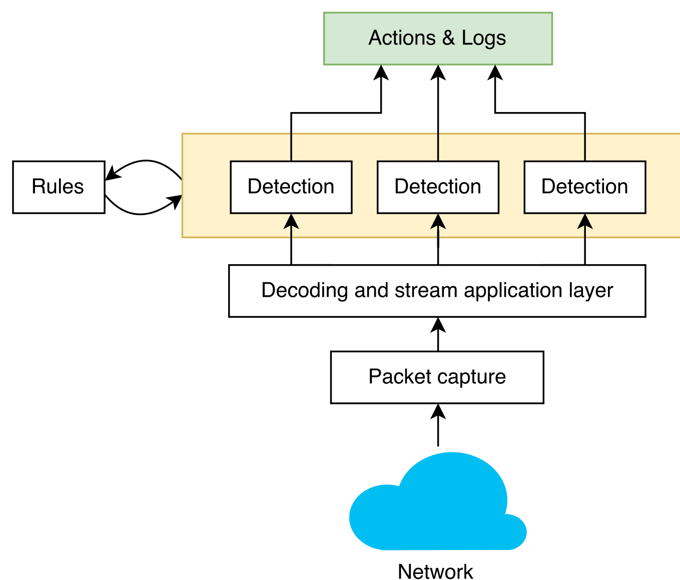


Figure 2.2: Suricata Architecture.

Suricata is also known for its efficiency and performance since it is a multithreaded solution and takes advantage of hardware acceleration [32]. According to White, Fitzsimmons & Matthews (2014), Suricata is better than Snort in terms of performance [50]. The performance of both systems was compared while scaling the number of Central Processing Unit (CPU) cores and varying configurations such as using multi-instance Snort. According to Fekolkin (2015), the

precision of the rules used by Suricata and Snort affects the rate of false negatives and false positives in terms of threat detection [6]. These rules can trigger actions, just like Snort, after the packet analysis phase. It is also important to mention that Suricata was mostly based on Snort, but besides being more modern, it took a step further into achieving a greater scalability, efficiency and precision.

Suricata can detect malware by analyzing signatures of executables or parts of binaries and then searches for known signatures, which is sometimes enough to detect common malware. However, it is a difficult task to detect new malware or new attacks, even for anti-viruses, which also use signature-based techniques among other methods. Regardless of this limitation, the possibility of saving malicious executables and other payloads present in the captured traffic for post-processing is also a plus while studying malware and what kind of attacks are being used against a company or organization.

Suricata has been included in some distributions such as OPNsense and Security Onion. It can be easily integrated with management tools or event processing technologies such as Elasticsearch, Logstash and Kibana developed by Elastic [1].

In short, Suricata is a very reliable and high performance IDS, Intrusion Prevention System (IPS) and NSM. It is easily scalable through multi-threading.

2.1.1.3 Bro IDS

Bro is an open source framework for network analysis and security monitoring, which can be used to build a powerful NIDS for UNIX systems. It was created by Vern Paxson in 1995 and it has been developed by researchers and students of the International Computer Science Institute (ICSI) [27]. It is a passive traffic analyzer which uses DPI to detect intrusions in the network traffic. It is also capable of performing tasks not related to security, such as traffic baselining and performance tests. There are many built-in functionalities: Bro parses application data, extracts files from Transmission Control Protocol (TCP) streams, identifies outdated versions of software, logs every event in the network in a well-structured format, which can be externally analyzed by other software and stored in databases, and much more. However, users can write their own scripts in order to achieve complex tasks, using a domain-specific, Turing-complete scripting language with an event-based programming model. [27].

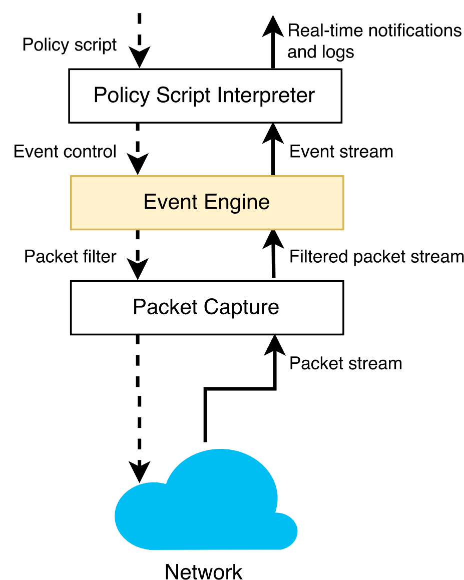


Figure 2.3: Bro Architecture.

Unlike Snort and Suricata, which are capable of performing intrusion prevention, Bro does not work in inline mode [19]. It is a policy based IDS that generates logs and also an excellent solution for intelligence gathering. It uses *libpcap* in order to filter the captured packets at the kernel level and reduce the workload by selecting packets needed by the current policy [36].

Bro has two main components: the event engine and the policy script interpreter. Packets are processed by the event engine, which performs processing task, such as state management and protocol parsing, and generates a stream of events which are passed to the policy layer [36]. The policy script processes these events with the scripts supplied by users. Users can define event handlers in these scripts and therefore perform certain actions, working with a high-level abstraction of the packets. These actions can even launch user-supplied scripts or external programs in order to trigger an active response to an attack [27].

In terms of scalability, Bro supports clustering for large-scale deployments and is also capable of performing both offline and real-time analysis. It supports many application layer protocol analysis, such as DNS, File Transfer Protocol (FTP), HTTP, Internet Relay Chat (IRC), Simple Mail Transfer Protocol (SMTP), Secure Shell (SSH) and SSL, and also non-application layer

analysis, which include built-in analyzers that can detect port scanning techniques. It also supports IPv6 and tunnel detection and further tunnel traffic analysis using decapsulation techniques. Bro supports alternative backends, such as Elasticsearch, Logstash, and Kibana by Elastic [1], since all connections, sessions, and application level data are written to a large set of log files [35], which can be useful to normalize and analyze Bro logs. Signature detection techniques are also support and Snort rules can also be easily imported.

Bro is a very reliable IDS solution and has a very extensive built-in support to analyze standard protocols. It is not multithreaded but supports cluster deployment solutions to achieve more performance.

2.1.2 Host-based Intrusion Detection Systems

A Host-based Intrusion Detection System (HIDS) monitors events inside the host that should be protected, by gathering logs and inspecting resources. They can operate at the operating-system-level and application-level. For both classes, there are three main approaches to analyze the collected data: misuse-based, anomaly-based and specification-based [49]. Misuse-based techniques consist in analyzing the stream of collected data and matching certain activities against descriptions or signatures of known attacks. Anomaly-based detection relies on models of the normal behavior of users and applications, which is efficient to detect unknown attacks although the number of false positives is high, as expected. Specification-based approaches are based on static analysis of applications, system call models and interceptions in order to classify them as malicious or not, according to certain specifications.

In short, HIDSes collect data from system calls, modifications in the file system and login events. The system log and known application logs are also useful to identify suspicious activities in the host.

In this section, we cover two well-known systems of this type: OSSEC and Tripwire Open Source.

2.1.2.1 OSSEC

OSSEC is an open source HIDS, created by Daniel B. Cid and now owned by Trend Micro. Like other HIDSes, OSSEC is capable of detecting intrusions in the host that is being monitored

by performing log analysis, file and registry integrity checking, time-based alerting and *rootkit* detection, among other features [43]. It is a cross-platform solution, but its manager runs only in UNIX systems. OSSEC can also analyze *syslog* events from many different firewalls and routers.

The architecture of OSSEC is composed of two main components: manager and agents. The manager is a server that offers centralized management and it is responsible for receiving, processing and correlating the information collected by the agents. It stores databases, logs, events and system auditing entries. Every rule, decoder and major configuration option is stored in this central component. The agents are installed in all the systems that should be protected. An agent performs real-time log collecting and does not affect the system performance. It is also important to mention that the communications between the manager and the agents are encrypted.

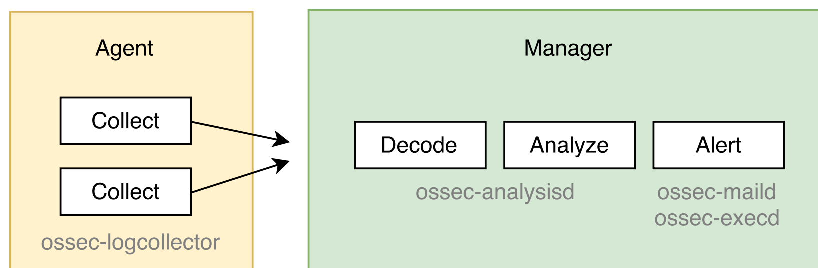


Figure 2.4: OSSEC Architecture.

Decoders should interpret the contents of a log entry by parsing, decoding and normalizing data according to its format [30]. The logs that match certain parameters of the decoder are forwarded to the rules phase for processing. Then, alerts are generated and actions are taken. Decoders and rules are written in the Extensible Markup Language (XML) format and it already contains default decoders that can parse well-known logs from sources such as web servers and SSH. Decoders and rules can easily be created in order to detect complex incidents.

The active response feature of OSSEC consists in performing actions in the agent or server in response to certain triggers, such as alerts or alert levels. OSSEC can send alerts via *syslog* to Logstash and it is easy to integrate it with Elasticsearch and Kibana, developed by Elastic [1], which is very useful to visually inspect dashboards and statistics. It can easily be enabled in OSSIM, a well-known open source Security Information and Event Management (SIEM).

2.1.2.2 Tripwire Open Source

The open source version of Tripwire is a HIDS. Specifically, it is a security and data integrity tool useful for monitoring and alerting on specific file changes on a range of systems [14]. It was created originally by a student, Gene Kim, and a professor, Dr. Eugene Spafford, back in 1992. The current code is based on a project launched in 2000 by Tripwire, Inc. Tripwire is available also as an enterprise version.

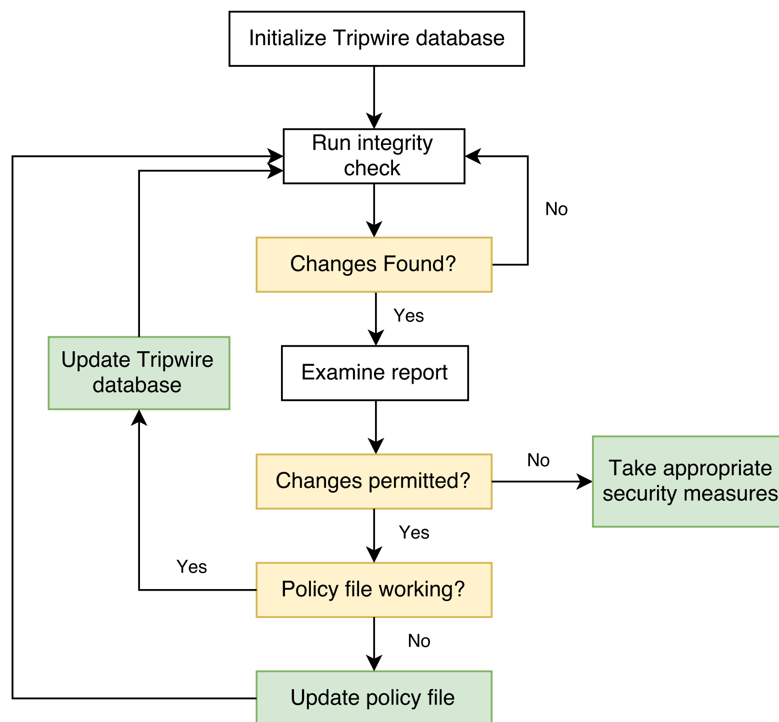


Figure 2.5: Tripwire Flowchart.

Tripwire runs integrity checks on the host and reports abnormal behavior to the user. It orders to compare two states, it scans for files and stores the respective hashes in a database [15] and then it looks systematically for certain file changes, both size and differences in the previously stored hashes. A potential intrusion is detected if any suspicious changes happen.

Tripwire Enterprise is better in terms of scalability, compliance and automation and is available for more platforms, such as Linux, Windows, Solaris and AIX. The Tripwire Enterprise Manager allows users to control a large number of Tripwire instances across multiple hosts. However, Tripwire Open Source shares many functionalities with the enterprise version [15], such as alerting

different users and groups based on the nature of the detected changes, assessing the level of seriousness of compromised file or directories, *syslog* reporting, among others.

Tripwire Open Source is a good solution for monitoring a small number of Linux servers but if centralized control, real-time alerts upon intrusion detection and advanced reporting features are needed, users need to use the enterprise version.

2.1.3 Host-based vs Network-based Intrusion Detection Systems

NIDSes are more portable, but network performance is always very important to the users. There is always a trade-off between security and usability, which has been overcome by multithreading and clustering. Using HIDSes is an approach that usually scales better [17] and tries to achieve the same goal at the host level.

In terms of online intrusion detection and detection, both solutions are good, but only HIDSes are capable of offline protection. A traditional HIDS does not need additional bandwidth to work since it operates at the host level. In terms of cross-platform compatibility, NIDSes are better than HIDSes since they can protect any host, regardless of its operating system. Both systems usually have good logging mechanisms, but only HIDSes can scan the host for file changes or registry scans, for example. An NIDS has a higher failure rate than a HIDS and it is a single point of failure [22].

One of the main advantages of using NIDSes is the possibility of issuing a verdict on a given packet according to a set of rules and triggering certain actions. Packet inspection and protocol analysis are also a plus since it is possible to discover complex attacks at the network level, especially in the application layer. The main advantage of HIDSes is the ability to inspect abnormal behavior inside the hosts, such as file changes and unauthorized access. The best solution is to use both NIDSes and HIDSes, since they use different approaches to achieve the same goal: protect a given computer system from attackers.

2.2 Distributed Systems

A distributed system is a collection of independent computers appears to its users as a single coherent system [41]. It is a system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing [3]. Distributed systems are connected inside a network and do not need to share memory since they communicate through messages in order to cooperate together and coordinate actions. These systems are a plus for intensive processing tasks because more performance can be achieved and more resources can be easily added.

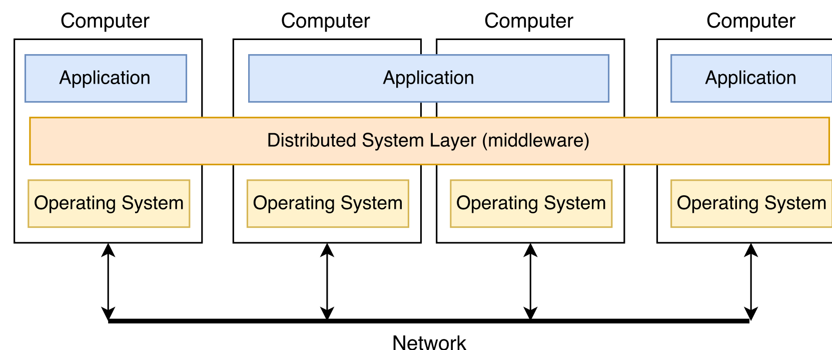


Figure 2.6: Distributed Systems Overview.

There are clear advantages while using distributed systems. In a non-distributed system, if there is a failure, the only option is to restart machines, the failure is total and we know that it occurred. In a distributed system, we should not know that a failure exists. Even if a given failure is partial, the system can be recovered since these systems implement strategies to handle failures. Distributed systems are scalable, easy to manage and are a plus in terms of processing and aim to improve the performance since it is possible to distribute tasks across a network with multiple systems. There are three main categories of middleware: Remote Procedure Call (RPC), Object Request Broker (ORB) and Message-oriented Middleware (MOM). Distributed applications use a common communication middleware to send messages and coordinate tasks. Middleware aims to provide common services and protocols that can be used by many different applications and operating systems.

2.2.1 Message-oriented Middleware

MOM is a software or hardware infrastructure, which allows distributed applications to easily send and receive messages. Software developers implement communications using a common Application Programming Interface (API) that supports multiple operating systems and network interfaces. MOM aims at high-level persistent asynchronous communication through the support of middleware-level queues [3] and ensures fault tolerance. Instances send each other messages, which are queued. Senders do not need to wait for an immediate reply. MOM systems ensure fault tolerance. There are 4 main actions in a MOM system [3]:

- PUT - Appends a message to a given queue
- GET - Block until the specified queue is nonempty and removes the first message
- POLL - Checks a given queue for new messages and removes the first without blocking
- NOTIFY - Install a handler to be called when a message is put into a given queue

Clients make a API to send messages to a given destination and can continue to do other operations since the communications are asynchronous, opposed to a request-response architecture. Message queue systems require a message broker, which takes care of application heterogeneity by transforming messages and acts as an application gateway. In this type of systems, the message received can be different from the original, since MOM systems can transform messages in order to match the requirements of the sender or the recipient [4]. Message queues are temporary storage when the destination is busy or unavailable.

The Advanced Message Queuing Protocol (AMQP) is a open standard application layer protocol designed for MOM by iMatix [10] back in 2004. This protocol takes care of message orientation, queuing, routing, reliability and security [25] and is a plus in terms of interoperability since it defines the protocol and formats used for communication.

2.2.2 ZeroMQ

ZeroMQ is a high-performance asynchronous messaging library, created by iMatix [10] in 2007, that provides a message queue. It supports message patterns like request-reply, publisher-subscriber,

push-pull (pipeline) and router-dealer. Unlike MOM, it can be used without a dedicated message broker. ZeroMQ is written in C++ but provides binding for many libraries, such as Python and Java. It can be used to easily implement distributed systems in the most common programming languages.

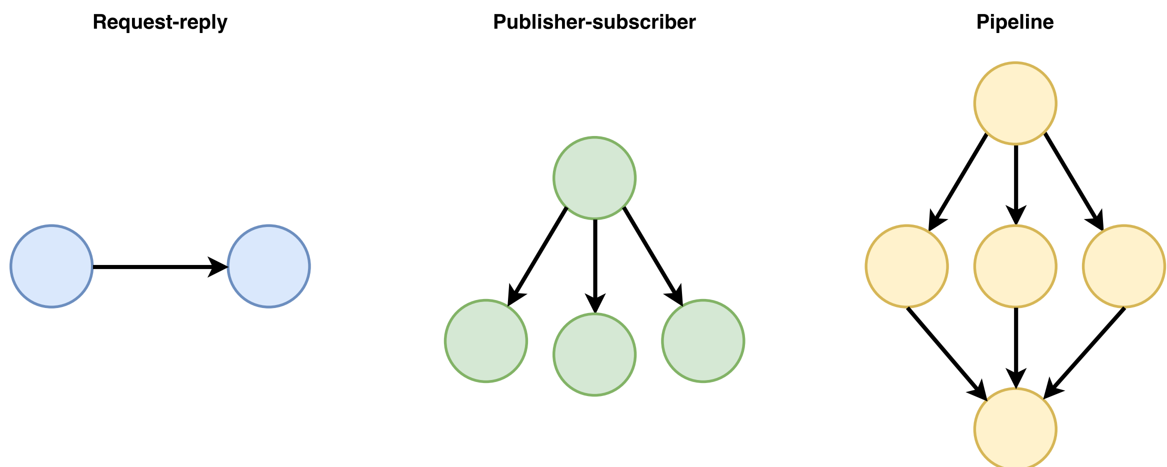


Figure 2.7: ZeroMQ: Basic Message Patterns.

ZeroMQ includes asynchronous sockets that support both unicast and multicast protocols, including *inproc*, Inter-process Communication (IPC), TCP and Pragmatic General Multicast (PGM). The following example was implemented in Python and implements a publisher-subscriber pattern [9]. The client subscribes a publisher running on localhost, port 5555 and specifies a filter for zip codes. The publisher generates random temperatures for 3 different zip codes. The client will receive temperatures only for the zip code `10001`, which is specified in the filter using `setsockopt`.

```
import zmq

context = zmq.Context()
socket = context.socket(zmq.SUB)

socket.connect("tcp://localhost:5555")

filter = "10001"
socket.setsockopt(zmq.SUBSCRIBE, filter)
```

```
10
11 while True:
12     print(socket.recv())
```

Listing 2.1: Publisher-subscriber: `subscriber.py`

```
1  import zmq
2  import random
3  import time
4
5  context = zmq.Context()
6  socket = context.socket(zmq.PUB)
7  socket.bind("tcp://*:5555")
8
9  zipcodes = [10001, 10002, 1003]
10
11 while True:
12     zipcode = zipcodes[random.randint(0, 2)]
13     socket.send("%d %d" % (zipcode, random.randint(1, 215)))
```

Listing 2.2: Publisher-subscriber: `publisher.py`

AMQP provides pre-packaged solutions to common problems whereas ZeroMQ provides tools that let users solve these problems easily in user-space [11]. ZeroMQ is a better solution compared to AMQP, with smarter queuing, fewer management costs, less complexity, and significantly better performance.

2.3 Exploit Detection

Unknown exploits are hard to detect by conventional means, such as IDS solutions. Current solutions for unknown attack detection and prevention follow both static and dynamic analysis of network traffic, looking for common payloads, unauthorized shells and *shellcode*. 0day exploits usually lead to lethal attacks because they take advantage of previously unknown vulnerabilities.

Some well-known IDSes, such as Snort and Suricata, support rules to detect the outcome of a successful exploit, such as reverse shells like meterpreter from Metasploit [29]. There is also a

plugin for Bro IDS that uses the Unicorn Engine to detect *shellcode* in network traffic [23] and emulates the instructions, looking for *syscalls* such as *execve*.

Some research works also covered and developed techniques to detect exploits. Polychronakis, M. (2009), developed some techniques to detect previously unknown code injection attacks through *shellcode* analysis. Static analysis of *shellcode* present in incoming packets cannot handle malicious code that employs advanced obfuscation methods [26]. In order to overcome this problem, an attack detection system was created, called *Nemu*, that uses a CPU emulator to perform dynamic analysis over valid instruction sequences. Some heuristics were developed that cover common *shellcode* types, such as self-decrypting and non-self-contained polymorphic *shellcode*, plain or metamorphic *shellcode* and memory-scanning *shellcode* [26]. This approach allows the detection of previously unknown attacks and is robust to evasion techniques like indirect jumps and self-modifying code.

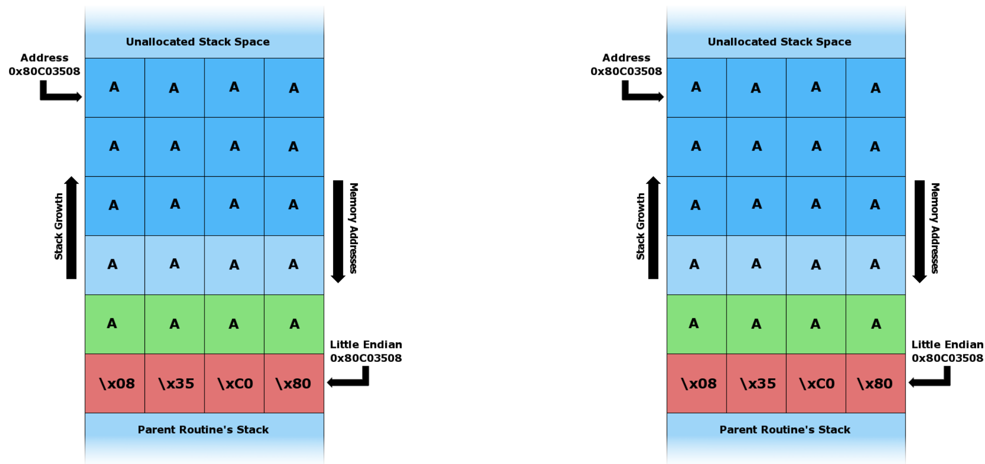
2.3.1 Memory Corruption

2.3.1.1 Background

Memory corruption occurs when a given location of memory is unintentionally accessed or modified, and therefore violating memory safety. Programming errors are the main cause of memory corruption vulnerabilities. A deep knowledge of low-level programming languages, such as C and C++, is required to avoid this type of errors. However, many Common Vulnerabilities and Exposures (CVE) have been issued due to this type of vulnerabilities and many of them are actually exploitable, giving attackers the ability of Remote Code Execution (RCE), privilege escalation or Denial Of Service (DOS), for example.

The most common vulnerability that leads to memory corruption is the buffer overflow, which happens when the adjacent memory of a buffer is overwritten and its contents are controlled by a given client or local user. This can happen in any segment of memory where the buffer is allocated, such as the stack and the heap. Buffers are allocated with a fixed size or variable size, but if an exploit can bypass the limit checks using other vulnerabilities, such as integer overflows, or if there no checks at all, memory corruption occurs. In a stack overflow, the saved return address of the current function and other variables can be overwritten in order to get control over

the execution flow of a given program.



(a) Intended Behavior. (b) Overwritten Memory.

Figure 2.8: Buffer Overflow in the Stack.

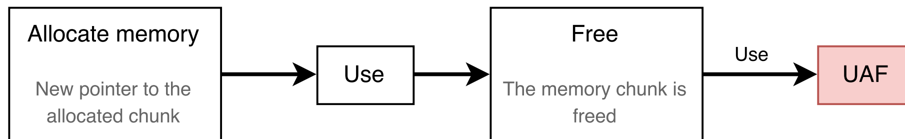


Figure 2.9: Use After Free.

Uninitialized memory and dangling pointers are also a common flaw that can lead to other vulnerabilities, such as Use After Free (UAF), which also can lead to successful exploitation, since exploits can control the heap and get control over the execution flow by predicting the heap state and then overwriting function pointers, for example. UAF is a very common vulnerability in Web Browsers. More advanced techniques also exist according to the memory allocator implementation.

2.3.1.2 Memory Corruption Mitigations

Many memory corruption mitigations have been introduced in various operating systems in order to make exploitation harder or even impossible in certain scenarios. These mitigations were introduced in many Linux distributions, but some also in other operating systems. Other mitigations that will not be described also exist, such as memory allocator hardening, which is

very useful to detect the corruption of the heap.

Data Execution Prevention

In the past, exploits could jump to *shellcode* if they were able to overflow a buffer in the stack. Basically, the idea was to overwrite the return address of the current function with a stack address in order to jump to *shellcode*, i.e., user-supplied assembly instructions, and execute arbitrary code that was not part of the original program.

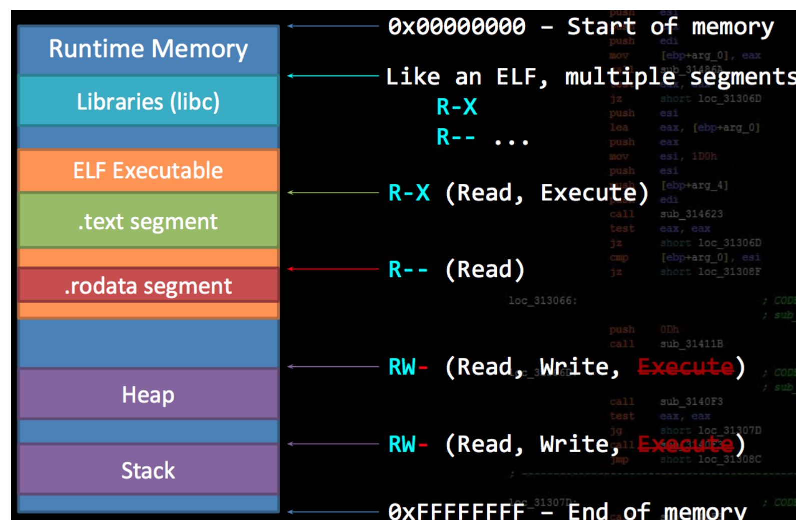


Figure 2.10: Data Execution Prevention.

Data Execution Prevention (DEP), also known as NX, XN, XD or W^X, is a mitigation for this type of attacks and it was introduced in the Linux kernel 2.6.8 back on August 14th, 2004, 11 days later on Windows and it was implemented later on Mac OSX in the year of 2006. DEP is a mitigation technique used to ensure that only code segments are ever marked as executable [16]. With this protection, a memory segment cannot be both writable and executable at the same time. In fact, heap and stack segments do not need to be executable, but only readable and writable, as shown in the Figure 2.10 (RPISEC MBE - Lecture 7). Every assembly instruction belongs to the code segment (*.text*). With this protection, which is enabled by default in many compilers, such as GNU Compiler Collection (GCC), *shellcode* cannot be executed in the stack and the heap as long as these segments are not executable.

Stack Canaries

Stack canaries, also known as stack cookies, named for their analogy to canaries in coal mines, can detect and prevent buffer overflows. This protection works by placing a randomly chosen value just before the return address of the current function. This value is chosen when the program starts and is static across functions and also in forked processes. Then, before the function returns, the canary value is checked for corruption.

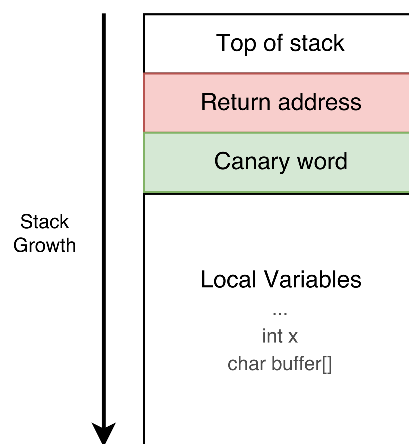


Figure 2.11: Stack Canary.

If a buffer overflow occurs and a return address is modified, the canary value will also change and the program will abort before returning to the new return address with the exception `stack smashing detected`. However, attackers can always corrupt other variables in the stack frame before overwriting the canary value and then use other techniques.

Address Space Layout Randomization

Address Space Layout Randomization (ASLR) was introduced in order to make exploitation harder in terms predicting target addresses. It was created back in 2001 and is enabled by default in the Linux kernel since the version 2.6.12, released in June 2005. This protection randomizes addresses, including stack, heap and libraries. Without ASLR, addresses do not change in different executions and can be easily predicted. A Position-independent executable (PIE) also randomizes

the code segment in order to make code-reuse attacks unreliable in terms of address prediction. For example, if there is a buffer overflow vulnerability in a given program, an attacker, while trying to exploit it, does not know how to jump to *shellcode*, since he does not know its location in memory due to address randomization.

The best implementations of ASLR were included in *grsecurity* [13] patches for the Linux kernel and are currently maintained by the PaX Team [45]. ASLR increases the consumption of the system's entropy pool since every task creation requires some bits of randomness to determine the new address space layout [44].

```
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /home/vagrant/a.out
0x0000000000600000 0x0000000000601000 0x0000000000000000 r-- /home/vagrant/a.out
0x0000000000601000 0x0000000000602000 0x0000000000001000 rw- /home/vagrant/a.out
0x0000000000602000 0x0000000000623000 0x0000000000000000 rw- [heap]
0x00007ffff7a0d000 0x00007ffff7bcd000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 0x00000000001c0000 --- /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 0x00000000001c0000 r-- /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 0x00000000001c4000 rw- /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 0x0000000000000000 rw-
0x00007ffff7dd7000 0x00007ffff7dfd000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fdf000 0x00007ffff7fe2000 0x0000000000000000 rw-
0x00007ffff7ff8000 0x00007ffff7ffa000 0x0000000000000000 rw-
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000025000 r-- /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x0000000000026000 rw- /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

Figure 2.12: Memory Space Mapping with ASLR Disabled.

```
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /home/vagrant/a.out
0x0000000000600000 0x0000000000601000 0x0000000000000000 r-- /home/vagrant/a.out
0x0000000000601000 0x0000000000602000 0x0000000000001000 rw- /home/vagrant/a.out
0x0000000000c9e000 0x0000000000cbf000 0x0000000000000000 rw- [heap]
0x00007f61875c3000 0x00007f6187783000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f6187783000 0x00007f6187983000 0x00000000001c0000 --- /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f6187983000 0x00007f6187987000 0x00000000001c0000 r-- /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f6187987000 0x00007f6187989000 0x00000000001c4000 rw- /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f6187989000 0x00007f618798d000 0x0000000000000000 rw-
0x00007f618798d000 0x00007f61879b3000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/ld-2.23.so
0x00007f6187b97000 0x00007f6187b9a000 0x0000000000000000 rw-
0x00007f6187bb0000 0x00007f6187bb2000 0x0000000000000000 rw-
0x00007f6187bb2000 0x00007f6187bb3000 0x0000000000025000 r-- /lib/x86_64-linux-gnu/ld-2.23.so
0x00007f6187bb3000 0x00007f6187bb4000 0x0000000000026000 rw- /lib/x86_64-linux-gnu/ld-2.23.so
0x00007f6187bb4000 0x00007f6187bb5000 0x0000000000000000 rw-
0x00007ffd71d1c000 0x00007ffd71d3d000 0x0000000000000000 rw- [stack]
0x00007ffd71d7b000 0x00007ffd71d7d000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

Figure 2.13: Memory Space Mapping with ASLR Enabled.

RELRO

RELRO was introduced to harden the data sections of an Executable and Linkable Format (ELF) binary [21]. The Global Offset Table (GOT) entries contain *libc* addresses of dynamically linked functions used by the program in runtime. This mitigation is a response to exploitation techniques that are dangerous to the GOT, such as overflows in the program data sections and *write-what-where* primitives that directly overwrite GOT entries. There are two modes: partial RELRO and full RELRO. Partial RELRO features:

- ELF sections are reordered - internal data sections, such as *.got* and *.dtors*, precede the non-internal data sections (*.data* and *.bss*)
- GOT is writable
- Non-PLT GOT is read-only

Full RELRO includes all the features from partial RELRO but also maps the GOT as read-only.

FORTIFY_SOURCE

The FORTIFY_SOURCE patch was provided by Jakub Jelinek in 2004. This patch supports buffer overflow detection and prevention of format string exploitation at both compile-time and runtime. FORTIFY_SOURCE provides an extra layer of validation for some function that can be a source of buffer overflow flaws [34]. It works by computing the number of bytes remaining to the end of a destination, which is passed to memory and string functions. If an exploit tries to copy more bytes from a source to a destination the program will terminate with the exception `buffer overflow detected`. It is also important to mention that this patch does not prevent all buffer overflows, but should prevent many common ones [18]. Users can turn on this mitigation in GCC compilers using the flag `-D_FORTIFY_SOURCE`, with both values 1 and 2. With `-D_FORTIFY_SOURCE=2` more checking is added, such as format string checks, but some programs might fail.

Given this buffer overflow example C code, the following two assembly code snippets are generated by GCC, each one resulting in activating and deactivating FORTIFY_SOURCE. The function `strcpy(dst, src)` was replaced with `__strcpy_chk(dst, src, dstlen)` in

the FORTIFY_SOURCE enabled assembly snippet.

```
#include <string.h>

int main(int argc, char **argv) {
   char buffer[5];
   strcpy(buffer, argv[1]);
}
```

Listing 2.3: Buffer Overflow Example.

```
main:
...
0x00000000004005bc <+38>:  mov    rdx,QWORD PTR [rax]
0x00000000004005bf <+41>:  lea    rax,[rbp-0x10]
0x00000000004005c3 <+45>:  mov    rsi,rdx
0x00000000004005c6 <+48>:  mov    rdi,rax
0x00000000004005c9 <+51>:  call   0x400460 < strcpy@plt >
0x00000000004005ce <+56>:  mov    eax,0x0
0x00000000004005d3 <+61>:  mov    rcx,QWORD PTR [rbp-0x8]
0x00000000004005d7 <+65>:  xor    rcx,QWORD PTR fs:0x28
0x00000000004005e0 <+74>:  je     0x4005e7 <main+81>
0x00000000004005e2 <+76>:  call   0x400470 <__stack_chk_fail@plt>
0x00000000004005e7 <+81>:  leave
0x00000000004005e8 <+82>:  ret
```

Listing 2.4: Assembly Code: FORTIFY_SOURCE Disabled.

```
main:
0x00000000004004c0 <+0>:   sub    rsp,0x18
0x00000000004004c4 <+4>:   mov    rsi,QWORD PTR [rsi+0x8]
0x00000000004004c8 <+8>:   mov    edx,0x5 ; dstlen
0x00000000004004cd <+13>:  mov    rdi,rsp
0x00000000004004d0 <+16>:  mov    rax,QWORD PTR fs:0x28
0x00000000004004d9 <+25>:  mov    QWORD PTR [rsp+0x8],rax
0x00000000004004de <+30>:  xor    eax,eax
0x00000000004004e0 <+32>:  call   0x4004a0 < __strcpy_chk@plt >
0x00000000004004e5 <+37>:  mov    rcx,QWORD PTR [rsp+0x8]
```

```
0x00000000004004ea <+42>:   xor    rcx,QWORD PTR fs:0x28
0x00000000004004f3 <+51>:   jne    0x4004fc <main+60>
0x00000000004004f5 <+53>:   xor    eax,eax
0x00000000004004f7 <+55>:   add    rsp,0x18
0x00000000004004fb <+59>:   ret
0x00000000004004fc <+60>:   call   0x400480 <__stack_chk_fail@plt>
```

Listing 2.5: Assembly Code: FORTIFY_SOURCE Enabled.

2.3.1.3 Modern Exploitation Techniques

Before trying to build an exploit, researchers can inspect the mitigations present in a given binary. For Linux, there is a tool named *checksec.sh* [20] that was created by Tobias Klein and can be used to perform this task.

```
gef➤  checksec
[+] checksec for '/home/vagrant/a.out'
Canary                                  : Yes
NX                                      : Yes
PIE                                     : No
Fortify                                 : No
RelRO                                   : Partial
gef➤
```

Figure 2.14: Checksec.

Bypassing Data Execution Prevention

Modern binary exploits use code-reuse attack techniques to bypass DEP: Return-oriented Programming (ROP), Jump-oriented Programming (JOP) and *return to libc* [5]. These techniques are based on reusing sequential assembly instructions, called *gadgets*, that are part of the original program or libraries. Typically, these gadgets can be used in a ROP chain if they end in a *ret* instruction. Code-reuse attacks control the call stack and therefore the program flow. ROP and JOP can be used to modify the stack, overwrite CPU registers and call functions in order to do unintended operations, such as leaking information or even run arbitrary code. *Return to libc* is another technique, found by Alexander Peslyak (Solar Designer), that basically returns to *libc* functions, such as *system*, to run arbitrary code. Overwriting GOT entries is also a

common technique to perform *return to libc* attacks: for example, the GOT entry of `puts` can be overwritten with the address of `system`.

Bypassing ASLR and Stack Canaries

One of the first techniques to bypass ASLR was the No-operation (NOP) sled. This technique works reasonably well on 32-bit systems, since they only have 16 randomized bits. The NOP sled consists in using a large number of NOPs, followed by *shellcode* in a payload, in order to massively increase the probability of returning to a fixed stack address that contains a NOP. NOPs will be executed and finally the *shellcode*, whose location in memory is unknown for a remote attacker.

Information leaks are also considered a serious vulnerability since they are the key to bypass mitigations such as ASLR and stack canaries. Information leaks usually leak memory addresses that can be later used for offset calculation and therefore defeating ASLR. A stack canary can be easily bypassed in a buffer overflow scenario if there is a leak of its value. In this case, the exploit should overflow the buffer and keep the canary value unchanged. Information leaks can happen by reading contiguous memory, due to the presence of format strings, UAF and pretty much everything that can be a *read-what-where* primitive, and can be used to further exploitation.

Other techniques

Generally, if both *read-what-where* and *write-what-where* primitives are achieved, the control flow of the program can be hijacked. The use of one primitive can also lead to the creation of the other in certain scenarios. This is the reason why a format string is a critical vulnerability since both primitives are supported by *printf* if no format is used and contents are controlled by the attacker. More advanced techniques exist, such as heap exploitation techniques, even for *off-by-one* vulnerabilities in the heap [46], that are able to bypass the most recent heap corruption checks. It is also important to mention that code-reuse attacks can be prevented with Control Flow Integrity (CFI), there some implementations such as Reuse Attack Protector (RAP) from *grsecurity* [42].

Chapter 3

Architecture

3.1 Overview

Hosts are protected by the Briareos Host Component (BHC) and contribute to the intrusion knowledge base of the protected network. The BHC receives network traffic and decides if packets are dropped or accepted, intercepts outgoing traffic and contains processing pipelines where the traffic is inspected in order to detect intrusions. The inspection of outgoing traffic is also necessary to detect unknown attacks and prevent leaks of information. It also supports offline modules, which are useful to detect intrusions inside the Operating System (OS) by monitoring resources and modifications in sensitive files. The BHC is capable of sharing information with the Briareos Manager Server (BMS) in order to protect all the other hosts in the network even if they did not suffer from the same attack. Every host can receive security feeds from the BMS and then activate new modules or pipelines. Both local and global rules can be propagated to all the other hosts and the NIDS as well. The NIDS can be any well-known IDS solution, since *Briareos* will provide a library to populate rules to some well-known IDSes. Traffic processing can be also distributed in a parallel mode throughout the Briareos Distributed System (BDS) in order to reduce the workload of hosts, but this feature does not support inline mode. The *Briareos* architecture was designed with unknown attack detection in mind since hosts can observe their own anomalies and correlate those with the incoming or outgoing traffic.

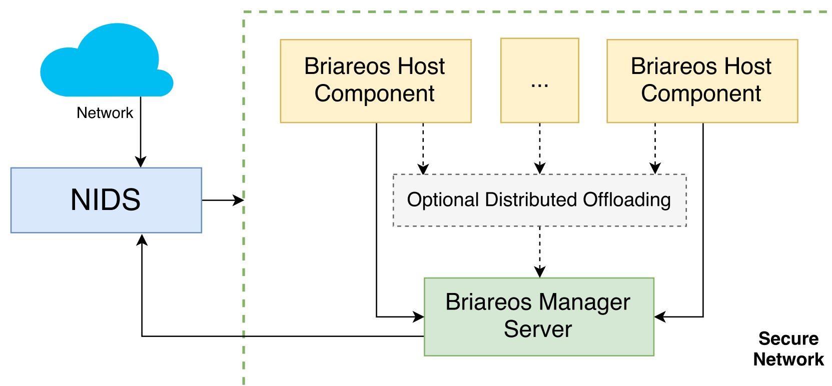


Figure 3.1: Briareos Architecture.

3.2 Host Component

The BHC has two main components: the network interceptor and the processing engine.

Host Component Requirements

- Interception of incoming and outgoing packets
- Detection and prevention of known and unknown attacks
- Traffic processing in inline and parallel modes
- Support for detection of intrusions at the OS level
- Forwarding traffic to a distributed system for analysis
- Automatic intelligence sharing
- Ability to receive feeds from the BMS

3.2.1 Loading Phase

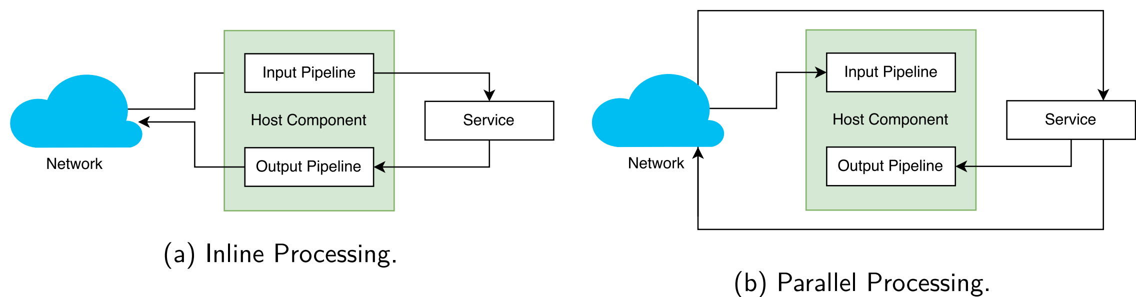
Initially, BHC loads a configuration file that specifies BMS information, pipelines, mappings between pipelines and ports. The loading phase initializes the processing engine and the network interceptor with the given configuration.

3.2.1.1 Processing Engine

The processing engine loads pipelines, which are stored in hash tables or a similar structure, and initializes all modules required by the loaded pipelines. It should know how to choose the correct pipeline to run a given packet. The mapping uses a dictionary defined by the attributes `port`, `name` and `type`. Multiple pipelines should be supported in a given port. However, the order of the pipelines has to be specified and the processing will be performed sequentially. When a pipeline drops a packet, it will not be processed by the next pipelines that are monitoring the same port.

	Advantages	Disadvantages
Inline Processing	• Sensitive information blocked before leaving host • Packets dropped before reaching services • Strict security - better for critical hosts	• Response delays if heavy processing is required
Parallel Processing	• Appropriate for distributing heavy processing jobs • Tolerant security - better for non-critical infrastructures • Better performance	• Window of intrusion • Actions are taken *a posteriori*

Table 3.1: Inline Processing vs Parallel Processing.



(a) Inline Processing.

(b) Parallel Processing.

Figure 3.2: Processing Modes.

The Table 3.1 shows the advantages and disadvantages of using the inline and parallel processing modes. While in parallel mode, if an attack is detected the connection can also be terminated and the client blocked *a posteriori*, but there is a small time window of intrusion since the beginning of the processing phase and until the attack detection. A hybrid mode is a mixture of these two modes, i.e., an input pipeline can run in a different mode than the matching output pipeline. This flexibility is useful when intrusion prevention is useful just for the input or the output flow of a given service.

3.2.1.2 Pipelines

A pipeline is a directed graph with the following attributes:

- Type: Pipelines can be attached to input and output chains
- Protocol: Protocols supported by iptables (TCP, UDP, ICMP)
- Interface: An interface can be specified
- Mode: inline or parallel
- IP: source or destination range of IP addresses
- Verdict: The default verdict issued on a packet (accept or drop)
- Modules: Description of the graph. Each node contains a module and specifies the next nodes

If the pipeline is not acyclic, an input/output matching verification phase occurs in order to verify if the pipeline is valid, since the output type of each module must match the input type of a given module that it connects to. It is not mandatory to have a final node in the pipeline because the original traffic is allowed to pass through by default unless a module blocks it during the processing phase. A pipeline package installer should be supported in order to automatically install module requirements.

3.2.1.3 Network Interception

The Netfilter library for Linux will be used to intercept and issue verdicts on packets. As soon as the processing engine is ready to receive packets, the interceptor uses the attributes of every pipeline to automatically create and apply iptables rules

If a rule has been successfully applied, the processing engine maps the pipeline with the current queue number in order to accelerate the decision of choosing the correct pipeline for a packet. Then, the interceptor binds to all queues using a dispatcher that stores a handler and a queue number. The handler function receives the Netfilter packet and the queue number as arguments and starts the processing phase. This efficient architecture allows the existence of 65537 pipelines since Netfilter queue numbers must be in the interval [0, 65536], which is not a disadvantage given the low number of exposed services in a common server.

3.2.2 Processing Phase

The pipeline runs a packet through its modules and returns an action, which is necessary for inline mode but is ignored in parallel mode. If the pipeline is in inline mode the next packet will be processed only when the current packet finishes the same procedure. On the other hand, if the pipeline is running in parallel mode, the packet is queued for further analysis and the pipeline default action is immediately returned. If the pipeline is running in distributed mode, then its traffic will be forwarded to the BDS. The pipeline processing uses a Breadth-first Search (BFS) algorithm with output propagation to next nodes. The following algorithm is in its simplified form, the actual algorithm should use multithreading for nodes in the same depth in order to improve the performance. In the case of multiple incidences, modules need to specify its input mode. In the single input mode, if there are n incident nodes, a given module processes inputs n times. In the multiple input mode, the module processes all the inputs at once.

The Figure 3.3 illustrates a web server pipeline with Structured Query Language (SQL) injection, cross-site scripting and local and remote file inclusion detectors. The root node contains an HTTP decoder and the input of each detector is an HTTP Object. Finally, a handler collects information, logs incidents and takes actions, such as blocking further connections from the same client, for example.

Input: $packet$

Output: $action$

$action = pipeline.default_verdict$

Queue Q;

Q.enqueue(($root_node$, $packet$));

while Q *is not empty* **do**

    $current_node, input = Q$.dequeue();

    $output, action, packet = current_node$.process($input, packet$);

    if $action$.*stop* $== TRUE$ **then**

        break;

    end

    for *each node* n *adjacent to* $current_node$ **do**

        Q.enqueue((n, $output$));

    end

end

Algorithm 1: BFS With Output Propagation.

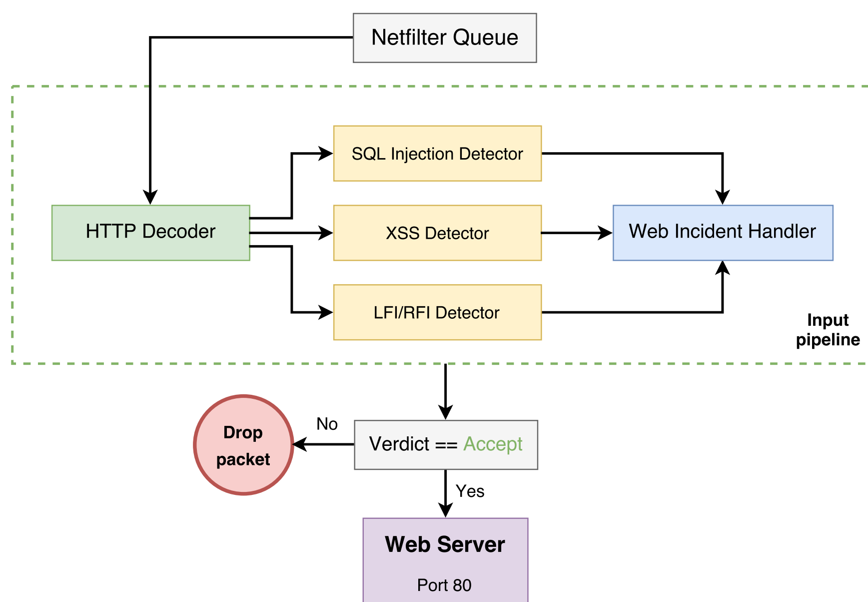


Figure 3.3: Example: Input Pipeline of a Web Server.

3.2.3 Modules

The root module receives the packet as bytes, in order to offer users more control and flexibility, but any packet parsing library can be used. An option that uses *scapy* [33] to automatically parse packets and construct objects should also exist. Modules are classes with one mandatory `process()` function and two optional `init()` and `cleanup()` functions. The `init()` function is called as soon as the module is loaded, which is useful for preloading resources, while the `cleanup()` function is called when the BHC service is stopped. Every `process()` function should return an output that should match its output type. However, if the module does not have next nodes, it can return nothing. The packet is always available and can be modified in every module, including its verdict and the payload itself.

There are four main types of pipeline modules: decoders, detectors, monitors and incident handlers. Decoders are responsible for parsing and decoding protocols and payloads or extracting any other information that is not evident in the raw packet. Typically, this type of modules should be used as the root node of the pipeline. Detectors should use the output of decoders and are capable of classifying traffic as malicious or not, and therefore blocking or allowing a packet to go through. Detectors can also be used as the root node of the pipeline if no decoding or parsing is necessary. Monitors should monitor the OS, such as processes or filesystem modifications and events. Incident handlers perform actions after the detection phase such as logging, sending alarms and creating new rules based on the attack that was detected. However, given the flexibility of module design and the infinite number of possibilities for both input and output types, it is possible to create modules of any other type.

Briareos should provide a powerful and extensible module library. The verdict of a packet issued by a given module can be accept or drop and it is possible to make the processing flow stop from any module. The payload of the packet can also be easily modified in order to filter known patterns, for example. It should also be possible to automatically get the application data of a given packet, get the input packet or packets that originated an output packet, create and populate new rules based on new anomalies detected by both input and output pipelines, among other features.

3.3 Intelligence Sharing

Both new local and global detection methods can be propagated to all the other hosts and the main NIDS, respectively. If a given host is the target of an unknown attack and detects or prevents it, then all the other hosts will be automatically protected. For example, every BHC can create new signature-based rules and share them with the other hosts in the network. The BMS is the component that distributes intelligence through the BHCs and the NIDS.

The BMS is responsible for collecting new rules from hosts and also publishing local rules to every BHC that subscribes the feed. On the other hand, if a given rule is a global rule, it will be propagated to the NIDS instead. It is also important to mention that the BDS can also create new rules and push them to the BMS.

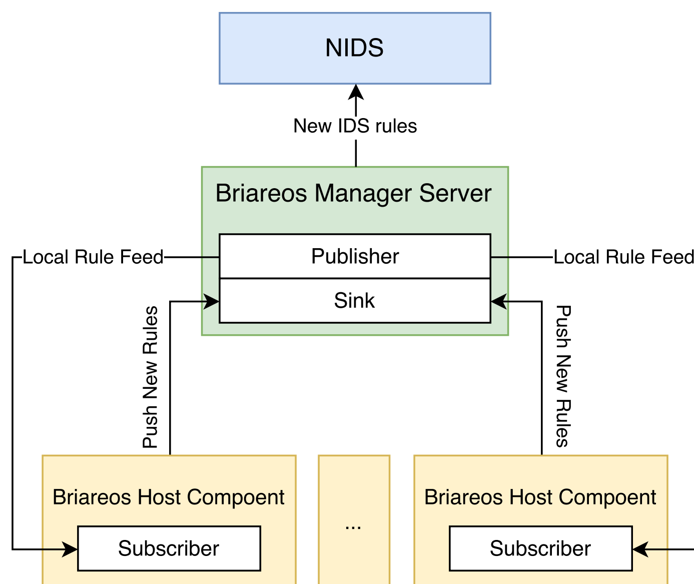


Figure 3.4: Intelligence Sharing.

3.4 Distributed System

The BDS main purpose is to reduce the workload of BHCs if no inline prevention is needed, allowing heavy processing tasks such as data mining techniques over the network traffic. This architecture was designed with scalability in mind. The BDS is elastic since it automatically starts

or stops worker instances inside a cluster or multiple clusters, called ZClusters, according to the workload stats of the workers reported by ZClusters.

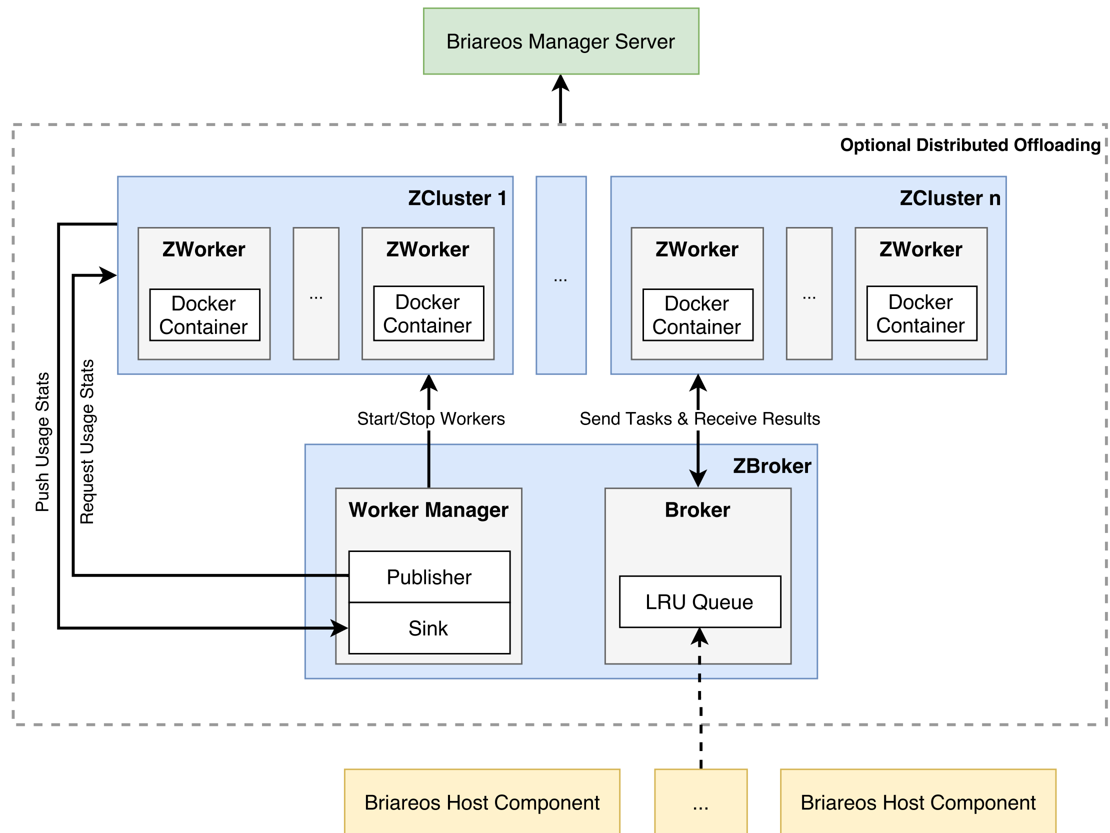


Figure 3.5: Distributed Offloading Architecture.

The BHCs send network traffic to the ZBroker, which will automatically distribute the workload for every worker in multiple clusters using a LRU queue algorithm (Algorithm 2). Two interfaces should be used, the backend, i.e., the worker pool, and the frontend for receiving new tasks from BHCs. This LRU queue will always poll the backend, but it will poll the frontend only if there is at least one worker ready. When a worker replies, then the task is completed and it will be queued as ready for new tasks.

ZWorkers are Docker containers [12] with limited, but configurable quota. ZClusters can start or stop new ZWorker instances upon ZBroker Worker Manager request. The Worker Manager request usage stats periodically from all available clusters, which are continuously collecting usage metrics, such as CPU and memory stats, from all worker instances. Then, ZClusters and push

those stats back to the Worker Manager sink.

The Worker Manager runs a sliding window algorithm and updates `average_cpu` and `average_memory` based on the history of worker usage stats that are received in the sink (Algorithm 3). At the same time, but in a different thread, the Worker Manager will then chose a ZCluster based on its overall resource consumption and will decide if a new worker instance should be started or stopped (Algorithm 4). A new worker should be started if the average CPU or memory usage is higher than a static upper bound, i.e., if CPU or memory resources are low. On the other hand, if the average CPU or memory is low and the other one is not high, then a new worker can be stopped.

Queue $available_workers$;
while $TRUE$ **do**
  if *new message from* $backend$ **then**
    $worker_id = backend$.recv();
    $available_workers$.enqueue($worker_id$);
  end
  if $available_workers$ *is not empty* **then**
    if *new message from* $frontend$ **then**
      $task = frontend$.recv();
      $worker_id = available_workers$.dequeue();
      $worker = backend$.get($worker_id$);
      $worker$.send($task$);
    end
  end
end

Algorithm 2: LRU Queue.

```
List cpu_values;
List memory_values;
current_time = time();
for stats in worker_stats_history do
    for stat in stats do
        if current_time - stat.time <= metric_interval then
            cpu_values.add(stat.cpu);
            memory_values.add(stat.memory);
        end
        else
            stats.remove(stat);
        end
    end
    average_cpu = average(cpu_values);
    average_memory = average(memory_values);
end
```

Algorithm 3: Sliding Windows.

```
while TRUE do
    if zclusters is not empty then
        if average_cpu >= cpu_upper_bound Or average_memory >=
          memory_upper_bound then
            zcluster = zclusters[0];
            for c in zclusters do
                if c.overall_usage < zcluster.overall_usage then
                    zcluster = c;
                end
            end
            zcluster.send(START_NEW_INSTANCE_MSG);
        end
        else if (average_cpu <= cpu_lower_bound And average_memory <
          memory_upper_bound) Or (average_memory <= memory_lower_bound And
          average_cpu < cpu_upper_bound) then
            if zclusters.length > 1 then
                zcluster = zclusters[0];
                for c in zclusters do
                    if c.overall_usage > zcluster.overall_usage then
                        zcluster = c;
                    end
                end
                zcluster.send(STOP_INSTANCE_MSG);
            end
        end
    end
    sleep(interval);
end
```

Algorithm 4: Decision Algorithm.

Chapter 4

Implementation

4.1 Project Overview

Briareos was implemented in Python, but it also uses third-party libraries that are implemented in low-level languages, such as C and C++. The main idea behind implementing this project in python was to reduce the development time needed to demonstrate the concept and also to easily achieve a modular architecture, with modules also written in Python. It is also important to mention that Python is a very common programming language in the information security community, which can also be a plus if this project goes open source.

The Figure 4.1 shows the overview of the classes in the current project version. Not all classes and attributes are shown, but only the most important ones. There are 5 main components, the BHC, the BMS, the ZCluster, the ZWorker and the ZBroker. Unfortunately, we did not have time to finish the implementation of the BMS, including rule propagation, and a pipeline package manager, which are still under development.

4.1.1 Instructions

First of all, users should install all the requirements of this project using the script `install-requirements.sh`, which will install all necessary packages on Debian-based distributions like Ubuntu, including `build-essentials`, `python-dev`, `python-pip` and `libnetfilter-queue-dev`, and also Python packages, such as `python-graph-core`,

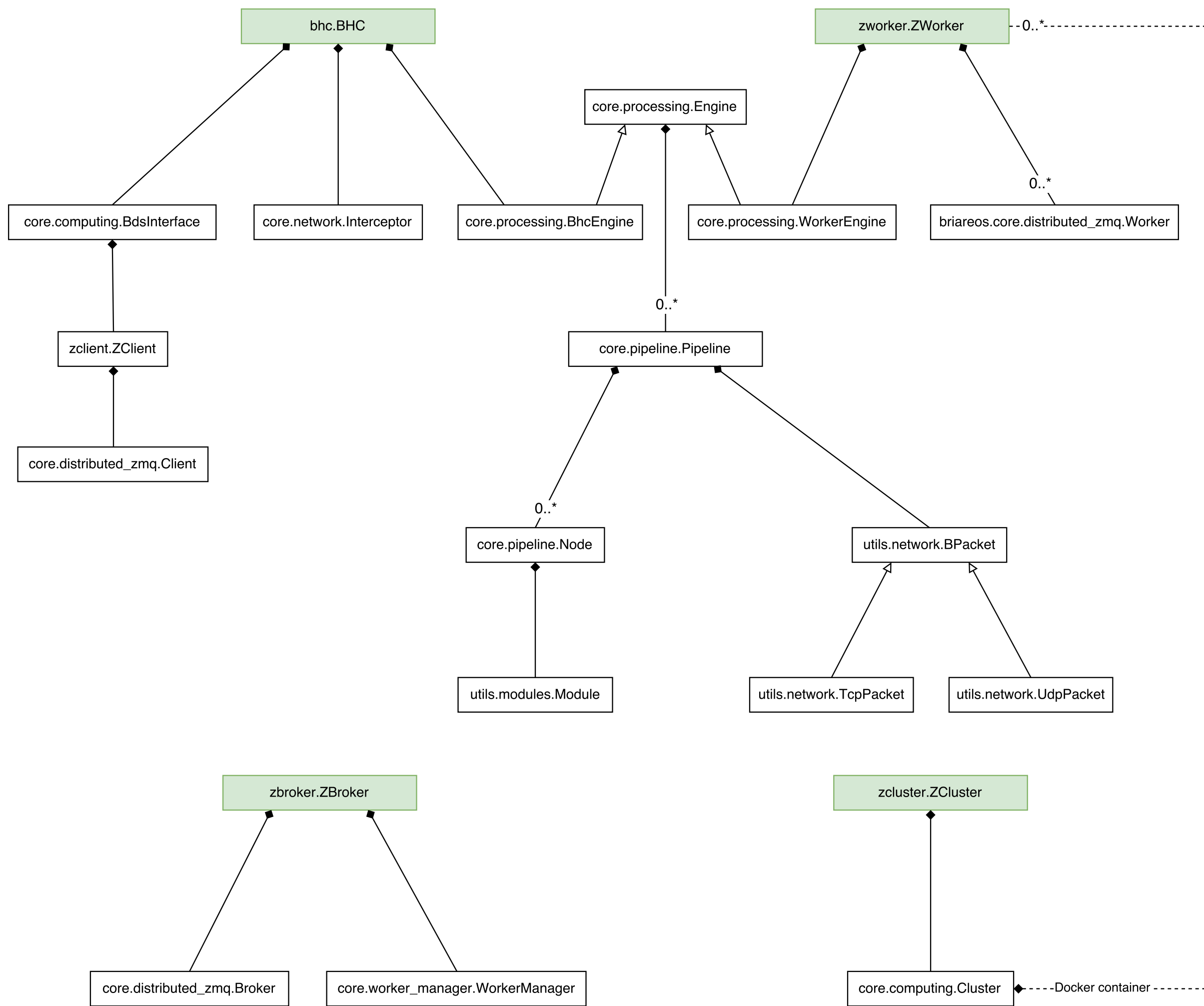


Figure 4.1: Class Diagram.

scapy, netfilterqueue, pyzmq, docker, among others.

```
apt update
apt install -y $(cat packages.txt)
pip install -r requirements.txt
```

Listing 4.1: Script: install-requirements.sh

If the user wants to use the BDS, additional steps are required. Docker should be installed on the operating system. The instructions available on the website https://docs.docker.com/engine/installation/ are recommended to install Docker Community Edition (CE). The bash script build-zworker-docker-image.sh builds the ZWorker docker image based on a given worker JSON config file. In the current version, if any configuration is modified, such as the ZBroker IP address, users need to rebuild the docker image.

```
docker build -f docker/zworker/Dockerfile -t zworker .
```

Listing 4.2: Script: build-zworker-docker-image.sh

Three programs are currently provided with *Briareos*: bhc, zbroker and zcluster.

```
# Run Briareos ZBroker
zbroker [-c broker.json]

# Run Briareos ZCluster
zcluster [-c cluster.json]

#Run Briareos Host Component
bhc [-c bhc.json]
```

Listing 4.3: How To Run Briareos.

```
############################################
#### Briareos Z-Broker || v0.36 - Alpha ####
############################################

[*] Initializing Briareos Z-Broker
[*] -> Worker Manager address: 'tcp://*:10003' | 'tcp://*:10004'
[*] Starting Z-Broker
[*] -> Frontend address: 'tcp://*:10001'
[*] -> Backend address: 'tcp://*:10002'
[*] Z-Broker ID: 69d739bc-9d74-11e7-b656-080027ee30f9
[+] Briareos Z-Broker is running
[*] New worker: 836a7c9a-9d74-11e7-81b5-080027ee30f9
```

Figure 4.2: Running the ZBroker.

```
#############################################
#### Briareos Z-Cluster || v0.36 - Alpha ####
#############################################

[*] Initializing Briareos Z-Cluster
[*] Starting new Z-Worker instance
[*] Connecting to Worker Manager: 'tcp://127.0.0.1:10003' | 'tcp://127.0.0.1:10004'
[*] Z-Cluster ID: 39d3692e-9d75-11e7-ba99-080027ee30f9
[+] Briareos Z-Cluster is running
```

Figure 4.3: Running the ZCluster.

```
##################################################
#### Briareos Host Component || v0.36 - Alpha ####
##################################################

[*] Initializing BHC
[*] Starting Processing Engine...
[*] -> Loading pipeline: 'engine/pipelines/simple_web_app_firewall.json'
[*] ---> Initializing module: HTTP Parser
[*] ---> Initializing module: SQL Injection Detector
[*] ---> Initializing module: XSS Detector
[*] ---> Initializing module: Incident Processor
[*] -> Loading pipeline: 'engine/pipelines/exploit_detection_leaks.json'
[*] ---> Initializing module: PID Fetcher
[*] ---> Initializing module: Address Leak Detector
[*] 2 pipelines loaded
[+] Processing engine is running
[*] Starting BDS Interface
[*] Connecting to Z-Broker: 'tcp://127.0.0.1:10001'
[+] BDS is ready
[*] Starting Interceptor...
[*] Configuring iptables...
[*] Binding queues...
[*] -> 3 queues binded
[+] Interceptor is running
[+] BHC is running
```

Figure 4.4: Running the BHC.

4.2 Host Component

In the BHC initialization phase, its configuration is loaded and the processing engine, the BDS interface and the network interceptor are initialized. The `start()` function starts the previous components. There is also a `stop()` function to perform cleanup tasks before quitting.

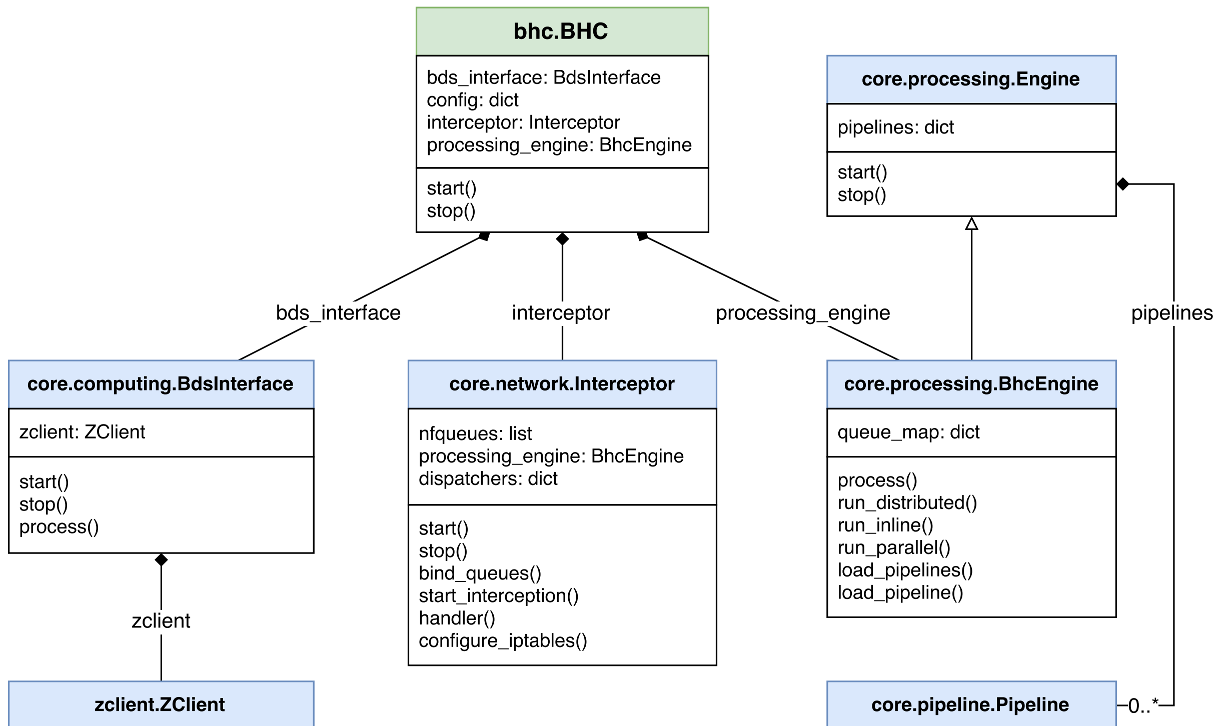


Figure 4.5: Host Component Class Diagram.

A configuration file specifies which pipelines should be used and ports to apply them. In distributed mode, the ZBroker address can also be specified.

```
1 {
2   "pipelines": [
3     {
4       "name": "simple_web_app_firewall",
5       "ports": [80, 8080]
6     },
7     {
```

```
8         "name": "exploit_detection_leaks",
9         "port": 1337
10      }
11    ],
12    "broker": {
13        "ip": "127.0.0.1",
14        "port": 10001
15    }
16  }
```

Listing 4.4: Example: BHC Configuration File.

4.2.1 Processing Engine

The class dependency of the processing engine is shown in the Figure 4.6. There are two types of processing engines, the BHC Engine and the Worker Engine. They both extend the class `core.processing.Engine` but they differ in some functions. For example, the engine of the BHC needs to map queues and pipelines based and also know how to run a packet in various modes, such as inline, parallel and distributed. On the other hand, a worker does not need to know queue IDs and in only has one mode of operation. The processing engine starts with pipeline loading, which looks for pipelines in the BHC configuration, and then tries to load pipeline configurations and the respective modules. A pipeline configuration should specify its type (input or output), protocol, a network interface, a processing mode (inline, distributed, parallel), the default verdict (accept or drop) and the directed graph of the modules, as shown in the Listing 4.5.

```
1  {
2    "type": "output",
3    "protocol": "TCP",
4    "interface": "any",
5    "mode": "inline",
6    "verdict": "accept",
7    "name": "Exploit Detection (Memory Leaks)",
8    "modules": [
9      {
```

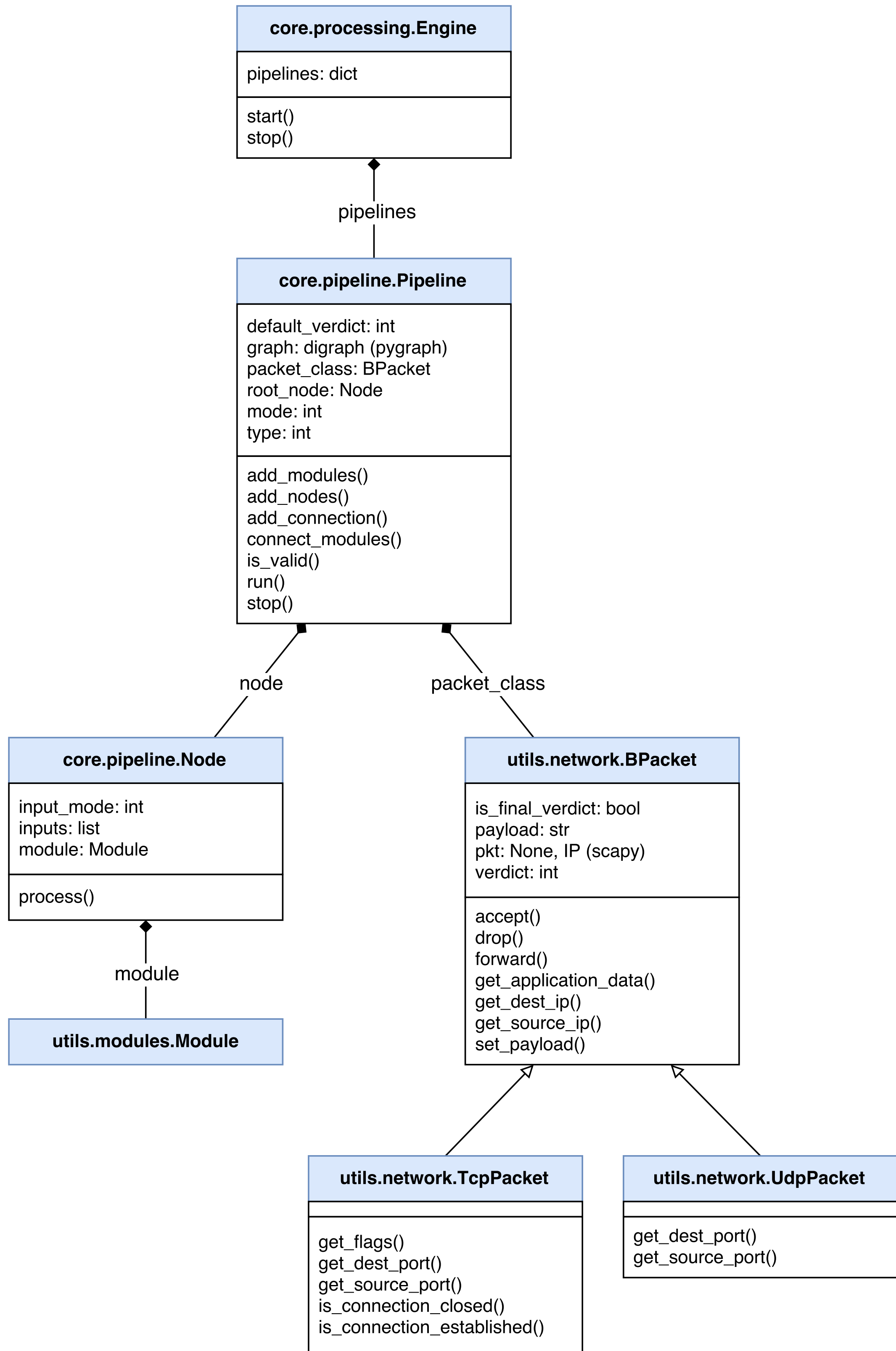


Figure 4.6: Processing Engine Class Diagram.

```
10         "name": "pid_fetcher",
11         "next": "leak_detector"
12       }
13   ]
14 }
```

Listing 4.5: Example: Pipeline Configuration File.

A pipeline contains a `digraph` from the `pygraph` library. The loader automatically loads a pipeline from a configuration file and connects every module to each other. The following functions handle the connections between nodes. If the input mode of a given module is single and not multiple, the input type of the target module must match the output type of the source module. In the single input mode, if a node has n incident nodes, then it will be processed n times for each input. In the multiple input mode, all the inputs are combined together in a list.

```
1  def connect_modules(self, source_module, target_module):
2      source_node = None
3      target_node = None
4      for i in range(len(self.nodes)):
5          if self.nodes[i].module.__module__ == source_module:
6              source_node = self.nodes[i]
7          elif self.nodes[i].module.__module__ == target_module:
8              target_node = self.nodes[i]
9
10     if source_node is not None and target_node is not None:
11         if target_node.module.input_mode == SINGLE_INPUT_MODE:
12             output_type = source_node.module.output_type
13             input_type = target_node.module.input_type
14             if input_type != output_type:
15                 return False, (input_type, output_type)
16         self._add_connection(source_node, target_node)
17     return True, None
18
19 def add_connection(self, source_node, target_node):
```

```
20     self.graph.add_edge((source_node, target_node))
```

Listing 4.6: Connecting Functions.

Regardless of the processing mode being inline, distributed or parallel, a pipeline has a `run()` function that basically runs the packet through its nodes and implements the Algorithm 1. A node is a class that contains a module.

```
def run(self, packet_payload):
    packet = self.packet_class(packet_payload, self.default_verdict)

    queue = [(self.root_node, packet)]
    output = None

    while len(queue):
        current_node, obj = queue.pop(0)

        if current_node.input_mode == SINGLE_INPUT_MODE:
            output = current_node.process(packet, obj)
        elif current_node.input_mode == MULTIPLE_INPUT_MODE:
            current_node.inputs.append(obj)
            if len(current_node.inputs) == \
                    len(self.graph.incidents(current_node)):
                output = current_node.process(packet,
                                              current_node.inputs)
                current_node.inputs = []

        if packet.is_final_verdict:
            return packet

        for neighbor_node in self.graph.neighbors(current_node):
            queue.append((neighbor_node, output))

    return packet
```

Listing 4.7: Function: `Pipeline.run()`

The class utils.network.BPacket is a class that contains the packet payload, the verdict, the boolean is_final_verdict, which is set to True if no further processing is needed by the next nodes, the scapy packet object pkt, which is the result of parsing the packet payload. The packet parsing into a scapy object is purely optional, since it can reduce the performance of the processing engine. Finally, the BPacket class is currently extended by two classes, TcpPacket and UdpPacket, which support more functions that can be directly used in modules, such as is_connection_established() and get_dest_port().

4.2.2 Network Interceptor

The core.network.Interceptor relies on netfilterqueue for network traffic interception, issuing verdicts on packets and modifying payloads. The interceptor starts by configuring iptables, which will apply rules based on the loaded pipelines since the processing engine is loaded before starting the interceptor. The following example shows a pipeline and the respective iptables rules.

```
{
  "type": "output",
  "protocol": "tcp",
  "interface": "eth0",
  "mode": "inline",
  "ip": "192.168.1.1",
  "verdict": "accept",
  "modules": []
}
```

Listing 4.8: Pipeline Configuration File.

```
$ iptables -I OUTPUT -o eth0 -s 192.168.1.1 -p tcp --sport 80 -j NFQUEUE
    --queue-num 1 --queue-bypass
```

Listing 4.9: Iptables Rule Matching the Previous Pipeline.

In this example, outgoing packets sent by the interface eth0 with 192.168.1.1 as the destination IP address, using the TCP protocol and source port 80 will be processed by queue number 1 that

is being monitored by the corresponding pipeline. The `queue-bypass` option is necessary to avoid dropping packets if no software in the user space is listening to a given queue. The function `configure_iptables()` uses a iptables library created exclusively to apply Netfilter rules.

```
1 def configure_iptables(self):
2     for pipeline in self._processing_engine.pipelines:
3         queue_id = self._get_current_queue_id()
4         if queue_id == -1:
5             logger.warning("Maximum number of queues reached")
6             return
7 
8         chain = iptables.CHAIN_INPUT
9         if pipeline.type == OUTPUT_PIPELINE:
10            chain = iptables.CHAIN_OUTPUT
11
12        if iptables.create_nfqueue_rule(queue_id, chain,
13                                        pipeline.port,
14                                        pipeline.protocol,
15                                        pipeline.interface,
16                                        pipeline.source_ip):
17            self._processing_engine.queue_map[queue_id] = pipeline
18        else:
19            logger.warning("Error configuring iptables "
20                           "for pipeline: %s" % pipeline.name)
```

Listing 4.10: Function: `Interceptor.configure_iptables()`

Then, the interceptor creates a map of dispatchers, which contain handlers aware of the current Netfilter queue ID for a given packet. These handler functions call the `process()` function of the `processing_engine`, which automatically runs the network traffic in the pipeline that is attached to the given queue. These handler functions issue a verdict on the `nfpacket` and can also modify payloads, based on the result of processing a given packet.

```
1 def handler(self, nfpacket, queue_id):
2     packet = self._processing_engine.process(nfpacket.get_payload(),
```

```
    queue_id)
    if packet.verdict == VERDICT_ACCEPT:
        if packet.payload is not None:
            nfpacket.set_payload(packet.payload)
        nfpacket.accept()
    elif packet.verdict == VERDICT_DROP:
        nfpacket.drop()
```

Listing 4.11: Function: `Interceptor.handler()`

4.2.3 Modules

A module is a class that extends the class `utils.modules.Module`. Multiple attributes should be specified, such as name, description, author, version and license. The input mode of a given module should also be specified, otherwise it will be set to the single input mode by default. Users need to specify the input type and output type of a given module. The next module is an example of a module that returns the Process Identification Number (PID) and the application name of the process that is sending or receiving a packet.

```
from briareos.utils import *

from procmon import ProcMon
import proc


class PidFetcher(Module):
    name = "PID Fetcher"
    description = "Returns the PID of a connection"
    author = "Andre Baptista"
    version = "1.0"

    output_type = tuple  # same as: io = (None, tuple)
    # input_mode = SINGLE_INPUT_MODE

    def __init__(self):
```

```
        self.procmon = ProcMon()
        self.procmon.enable()
        self.procmon.start()

    def cleanup(self):
        self.procmon.disable()

    def process(self, packet):
        source_port = packet.get_source_port()
        dest_port = packet.get_dest_port()
        src_ip = packet.get_source_ip()
        dest_ip = packet.get_dest_ip()

        pid, app_name = \
            proc.get_pid_by_connection(self.procmon, src_ip,
                                       source_port, dest_ip,
                                       dest_port, packet.protocol)

        # Other usage examples:
        # data = packet.get_application_data()
        # packet.accept()
        # packet.drop(final=True)
        # block_ip_address(packet.get_dest_ip())

        return pid, app_name
```

Listing 4.12: Module Example.

4.3 Distributed System

The BDS was implemented using two main technologies: ZeroMQ and Docker. There are 4 main components in the BDS: the ZClient, the ZBroker, the ZCluster and the ZWorker. The BDS was implemented in a way that achieves elasticity and performance, taking advantage of fast communications and workload awareness. The ZClient sends tasks to the ZBroker, which distributes tasks through workers using a LRU queue. The ZBroker contains a Worker Manager

that controls the number of workers according to the global workload.

4.3.1 ZClient

The ZClient is part of the BHC and is the interface that sends tasks to the ZBroker. It starts the ZeroMQ client interface (`distributed_zmq.Client`) by initializing sockets according to the given configurations. The processing engine puts tasks in a queue, which are then consumed by the ZClient. Then, tasks are sent to the ZBroker along with a pipeline identification, since the workers need to how the right pipeline to run a given packet. The ZClient uses a *DEALER* socket and connects to a *ROUTER*, the ZBroker frontend.

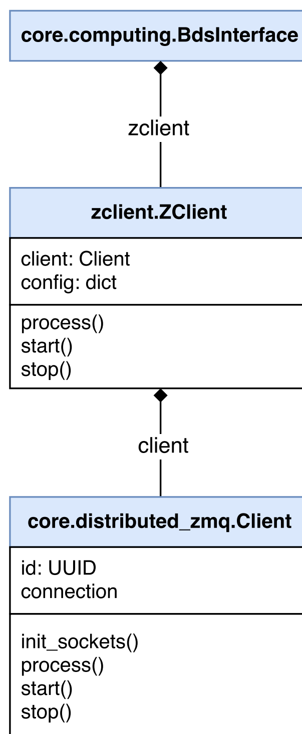


Figure 4.7: ZClient Class Diagram.

```
def init_sockets():
    self._context = zmq.Context()
    self.connection = self._context.socket(zmq.DEALER)

```

```python
# Connect to ZBroker
def start(self):
    self.connection.connect(self.broker_address)

# Send task and pipeline id
def process(self, task, pipeline_id):
    self.connection.send(task, zmq.SNDMORE)
    self.connection.send(pipeline_id)
```

Listing 4.13: ZeroMQ: ZClient.

4.3.2 ZBroker

The ZBroker contains two main components, the ZeroMQ broker interface and the Worker Manager, as shown in the Figure 4.8. The broker configurations are loaded from a configuration file that specifies the broker frontend and backend ports and also the Worker Manager ports and the interval for requesting ZWorker stats. The ZeroMQ interface uses two *ROUTER* sockets, one for frontend communication and the other for backend communication. The ZBroker follows the Algorithm 2 to implement the LRU queue.

```json
{
  "broker": {
    "frontend": {"port": 10001},
    "backend": {"port": 10002},
    "worker_manager": {
      "port": 10003,
      "sink_port": 10004,
      "interval": 10
    }
  }
}
```

Listing 4.14: Example: ZBroker Configuration File.

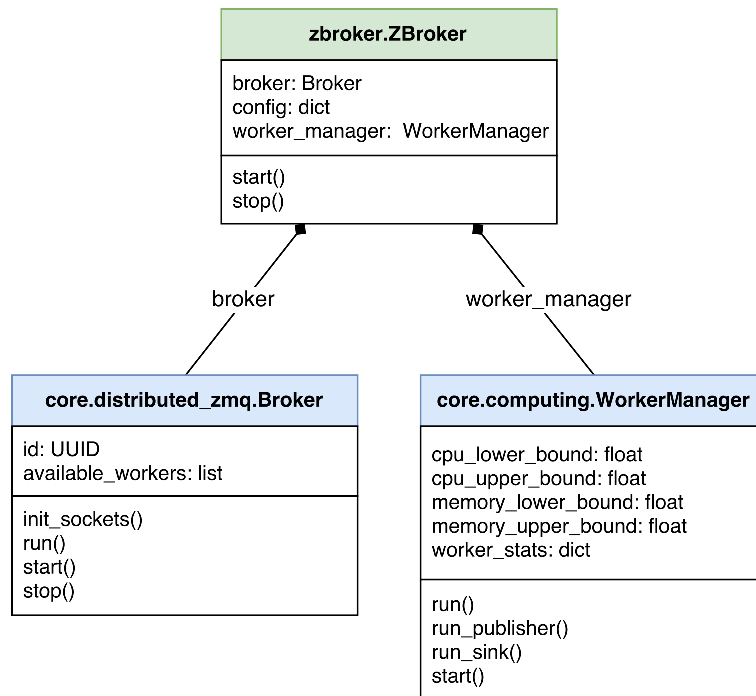


Figure 4.8: ZBroker Class Diagram.

```
1  def init_sockets():
2      self._context = zmq.Context()
3      self.frontend = self._context.socket(zmq.ROUTER)
4      self.backend = self._context.socket(zmq.ROUTER)
5
6  # Bind sockets
7  def start(self):
8      self.frontend.bind(self.frontend_address)
9      self.backend.bind(self.backend_address)
10     self._poller = zmq.Poller()
11     self._poller.register(self._backend, zmq.POLLIN)
12     self._poller.register(self._frontend, zmq.POLLIN)
13
14 def run(self):
15     # LRU Queue
16     while True:
17         sockets = dict(self._poller.poll())
18         if self.backend in sockets \
```

```
19                 and sockets[self.backend] == zmq.POLLIN:
20             worker_id = self.backend.recv()
21             self.available_workers.append(worker_id)
22 
23             self.backend.recv()   # empty
24             client_id = self.backend.recv()
25 
26             if client_id == Worker.ready_msg:
27                 logger.info("New worker: %s" % worker_id)
28             else:
29                 reply = self.backend.recv()
30 
31         if self.available_workers:
32             if self.frontend in sockets \
33                     and sockets[self.frontend] == zmq.POLLIN:
34                 client_id = self.frontend.recv()
35                 task = self.frontend.recv()
36                 pipeline_id = self.frontend.recv()
37 
38                 worker_id = self.available_workers.pop()
39 
40                 self.backend.send(worker_id, zmq.SNDMORE)
41                 self.backend.send("", zmq.SNDMORE)
42                 self.backend.send(client_id, zmq.SNDMORE)
43                 self.backend.send(task, zmq.SNDMORE)
44                 self.backend.send(pipeline_id)
```

Listing 4.15: ZeroMQ: ZBroker.

The Worker Manager binds two sockets, one *PUB* socket for the publisher and one *PULL* socket for the sink, i.e., to request usage stats from the workers and collect those same stats, respectively. Three threads are used, one for the publisher, other for the sink and the other for usage calculation and deciding if one more worker is needed or if the cluster can stop one worker instance. The Worker manager chooses a cluster according to Algorithm 4 to start a new worker, or it chooses a worker after choosing a cluster to stop a worker.

```
def init_sockets(self):
    self._context = zmq.Context()
    self.publisher = self._context.socket(zmq.PUB)
    self.sink = self._context.socket(zmq.PULL)

def start(self):
    self.publisher.bind(self.publisher_address)
    self.sink.bind(self.sink_address)
    Thread(target=self.run_publisher).start()
    Thread(target=self.run_sink).start()
    Thread(target=self.run).start()

def run_publisher(self):
        while True:
            self.publisher.send(USAGE_MSG)
            time.sleep(self.interval)

def run_sink(self):
    while True:
        cluster_id = self.sink.recv()
        stats = self.sink.recv_json()
        self._process_usage_stats(cluster_id, stats)

def run(self):
    while True:
        self._sliding_windows()
        self._decision_algorithm()
        time.sleep(self.interval)

def _start_new_instance(self):
    self.publisher.send(START_NEW_INSTANCE_MSG)

# After choosing a worker based on average cluster usage
def _stop_instance(self, worker_id)
    self.publisher.send(STOP_INSTANCE_MSG, zmq.SNDMORE)
```

```
36     self.publisher.send(worker_id)
```

Listing 4.16: ZeroMQ: Worker Manager.

4.3.3 ZCluster

The ZCluster loads a configuration that specifies the Worker Manager address, which is part of the ZBroker. It connects to the Worker Manager by subscribing the publisher and after receiving a usage request, it calculates the CPU and memory usage of all workers and sends back the stats to the Worker Manager sink.

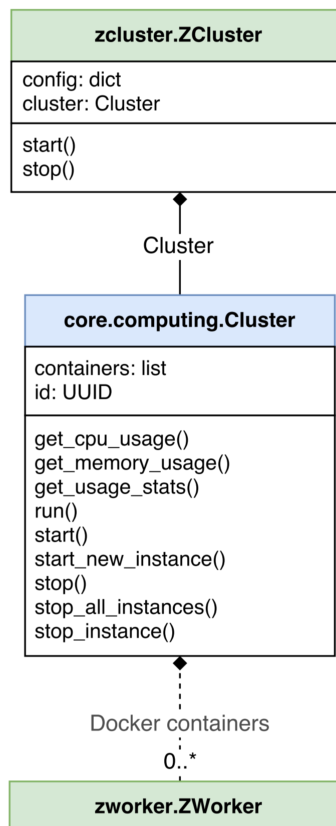


Figure 4.9: ZCluster Class Diagram.

```
1 {
2   "cluster": {
3     "worker_manager": {
```

```
4         "ip": "127.0.0.1",
5         "port": 10003,
6         "sink_port": 10004
7       }
8     }
9 }
```

Listing 4.17: Example: ZCluster Configuration File.

The ZCluster uses a *SUB* socket to connect to the publisher and a *PUSH* socket to send results to the sink. If it receives a start instance message, it uses the Docker client to run a new container. If it receives a stop instance message, it waits for the worker ID and if the worker ID is one of the containers, then it stops it using the Docker client.

```
1  def init_sockets(self):
2      self._context = zmq.Context()
3      self.publisher = self._context.socket(zmq.SUB)
4      self.publisher.setsockopt(zmq.SUBSCRIBE, "")
5      self.sink = self._context.socket(zmq.PUSH)
6      self._docker_client = docker.from_env()
7
8  def start(self):
9      self.start_new_instance()
10     self.publisher.connect(self.worker_manager_address)
11     self.sink.connect(self.sink_address)
12
13 def start_new_instance(self):
14     client = self._docker_client
15     container = client.containers.run(Cluster.docker_image_name,
16                                       detach=True,
17                                       network_mode="host",
18                                       cpu_period=self._cpu_period,
19                                       cpu_quota=self._cpu_quota)
20 def stop_instance(self, worker_id):
21     if self.containers:
22         container = self._worker_ids[worker_id]
```

```
23          container.stop()
24          self.containers.remove(container)
25
26  def run(self):
27      while True:
28          data = self.publisher.recv()
29          if data == USAGE_MSG:
30              usage_stats = self.get_usage_stats()
31              self.sink.send("%s" % self.id)
32              self.sink.send_json(usage_stats)
33          elif data == START_NEW_INSTANCE_MSG:
34              self.start_new_instance()
35          elif data == STOP_INSTANCE_MSG:
36              worker_id = self.publisher.recv()
37              if worker_id in self._worker_ids:
38                  self.stop_instance(worker_id)
```

Listing 4.18: ZeroMQ: ZCluster.

The function `get_usage_stats()` returns dictionary with the usage of CPU and memory, whose values are between 0 and 100. These values are calculated by the ZCluster since it can watch its containers using the Docker Stats API.

```
1  def get_usage_stats(self):
2      usage_stats = {}
3      for container in self.containers:
4          container_stats = container.stats(stream=False)
5          container_id = container_stats["id"]
6          memory_usage = self.get_memory_usage(container_stats)
7          cpu_usage = self.get_cpu_usage(container_stats)
8          usage_stats[container_id] = {"memory": memory_usage,
9                                       "cpu": cpu_usage}
10     return usage_stats
11
12 @staticmethod
13 def get_cpu_usage(container_stats):
```

```
14        cpu_percent = 0.0
15        precpu_stats = container_stats["precpu_stats"]
16        cpu_stats = container_stats["cpu_stats"]
17        cpu_total_usage = cpu_stats["cpu_usage"]["total_usage"]
18        precpu_total_usage = precpu_stats["cpu_usage"]["total_usage"]
19        system_cpu_usage = cpu_stats["system_cpu_usage"]
20        presystem_cpu_usage = precpu_stats["system_cpu_usage"]
21        percpu_usage = cpu_stats["cpu_usage"]["percpu_usage"]
22
23        cpu_delta = cpu_total_usage - precpu_total_usage
24        system_delta = system_cpu_usage - presystem_cpu_usage
25
26        if cpu_delta > 0 and system_delta > 0:
27            cpu_percent = (cpu_delta*1.0 / system_delta) \
28                          * len(percpu_usage) * 100.0
29
30        return cpu_percent
31
32    @staticmethod
33    def get_memory_usage(container_stats):
34        memory_stats = container_stats["memory_stats"]
35        usage = memory_stats["usage"]
36        limit = memory_stats["limit"]
37        return usage*100.0/limit
```

Listing 4.19: Usage calculation with the Docker Stats API.

4.3.4 ZWorker

A ZCluster contains multiple ZWorker instances as Docker containers, but the ZWorker can also be run as a standalone. If the ZWorker is run as a standalone, it also starts one subworker for each CPU core, taking advantage of multithreading. It starts the processing engine, which is a `core.processing.WorkerEngine` class instance, and a ZeroMQ Worker interface. These two components must be configurated using a configuration file.

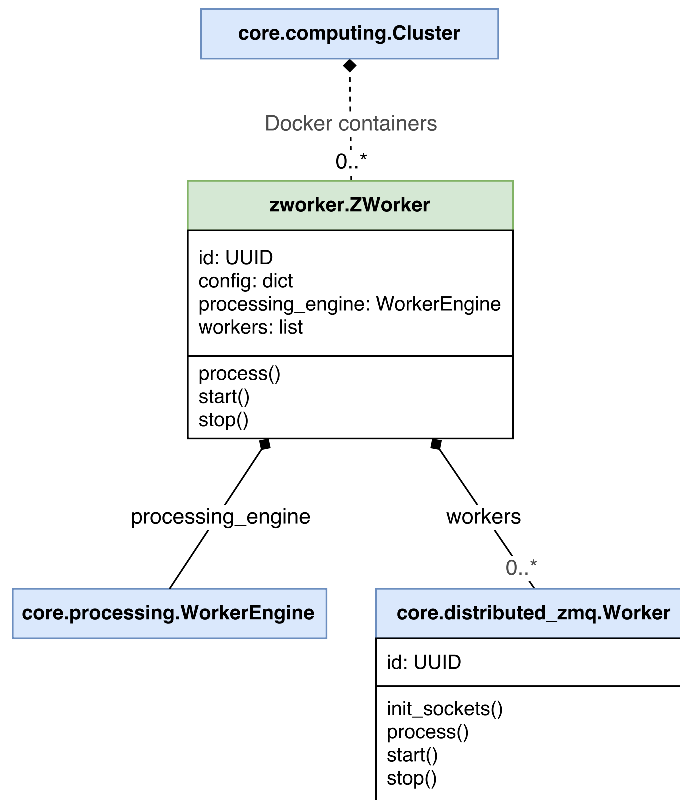


Figure 4.10: ZWorker Class Diagram.

```
1  {
2    "worker": {
3      "broker": {
4        "ip": "127.0.0.1",
5        "port": 10002
6      },
7      "worker_manager": {
8        "ip": "127.0.0.1",
9        "port": 10003,
10       "sink_port": 10004
11     },
12     "pipelines": [
13       {
14         "name": "exploit_detection_leaks"
15       },
16       {
```

```
17          "name": "simple_web_app_firewall"
18        }
19      ]
20    }
21  }
```

Listing 4.20: Example: ZWorker Configuration File.

The class `core.distributed_zmq.Worker` uses a *REQ* socket and connects to the ZBroker. Then it waits for tasks, processes tasks in the processing engine and sends a message to the ZBroker as soon as the task is completed.

```
1  def init_sockets(self):
2      self._context = zmq.Context()
3      self.connection = self._context.socket(zmq.REQ)
4
5  def start(self):
6      self.connection.connect(self.broker_address)
7
8  def run(self):
9      self.connection.send(Worker.ready_msg)
10
11     while True:
12         address = self._connection.recv()
13         task = self._connection.recv()
14         pipeline_id = self._connection.recv()
15         self._processing_engine.process(data, pipeline_id)
16         self.connection.send(address, zmq.SNDMORE)
17         self.connection.send(Worker.completed_msg)
```

Listing 4.21: ZeroMQ: ZWorker.

Chapter 5

Results

5.1 Preliminary Results

With *Briareos*, it is easy to build a module for pretty much everything. We tried to build a simple pipeline for a HTTP Web Server that simply drops packets and permanently blocks the client IP address if a given URL contains unsafe characters.

```
{
  "type": "input",
  "protocol": "TCP",
  "interface": "any",
  "mode": "inline",
  "verdict": "accept",
  "name": "Simple Web App Firewall",
  "modules": [
    {
      "name": "app_data_filter",
      "next": "simple_http_parser"
    },
    {
      "name": "simple_http_parser",
      "next": "generic_url_exploit_detector"
    }
```

```
  ]
}
```

Listing 5.1: Simple Web Application Firewall Configuration File.

Three modules were built: An application data filter, which basically only lets packets to go further in the pipeline if application data is present, a simple HTTP parser, which parses the application data into a HTTP object and finally a detector for unsafe URL characters.

```python
from briareos.utils import *


class AppDataFilter(Module):
    name = "Application Data Filter"
    description = "Filters packets without application data and returns
    it otherwise"

    output_type = str

    def process(self, packet):
        data = packet.get_application_data()

        if data is None:
            packet.accept(final=True)
            return None

        return data
```

Listing 5.2: Module: `app_data_filter.py`

```python
from briareos.utils. Import *

from http_parser import pyparser
from http_object import HttpObject


class HttpParser(Module):
```

```
 8     name = "Simple HTTP Parser"
 9     description = "Parses application data and returns a Http object"
10 
11     io = (str, HttpObject)
12 
13     def process(self, packet, data):
14         parser = pyparser.HttpParser()
15         parser.execute(data, len(data))
16 
17         method = parser.get_method()
18         status_code = parser.get_status_code()
19         url = parser.get_url()
20         headers = dict(parser.get_headers())
21         body = "".join(parser._body)
22 
23         http_object = HttpObject(method, status_code, url,
24                                  headers, body)
25         return http_object
```

Listing 5.3: Module: `simple_http_parser.py`

```
 1 from briareos.utils import *
 2 
 3 from http_object import HttpObject
 4 import string
 5 
 6 
 7 class GenericUrlExploitDetector(Module):
 8     name = "Generic URL Exploit Detector"
 9     description = "Drops packet and bans client if the URL contains
       characters that are not allowed"
10 
11     input_type = HttpObject
12 
13     def __init__(self):
14         self.whitelist = string.ascii_letters + string.digits \
15                          + ":/?#[]@!$&()*+,;=$-_%"
```

```
16
17    def is_bad_url(self, url):
18        for c in url:
19            if c not in self.whitelist:
20                return True
21        return False
22
23    def process(self, packet, http_object):
24        if self.is_bad_url(http_object.url):
25            packet.drop()
26            logger.info("Dropping packet: "
27                        "(Endpoint: %s)" % http_object.url)
28            block_ip_address(packet.get_source_ip())
```

Listing 5.4: Module: `generic_url_exploit_detector.py`

In fact, this simple pipeline can prevent attacks such as SQL injection payloads that directly use single quotes, for example, or even some Cross Site Scripting (XSS) payloads.

```
1 # Simple XSS payload:
2 # http://127.0.0.1/s=<script>alert(1);</script>
3 [*] Dropping packet: (Endpoint: /s=<script>alert(1);</script>)
4
5 # Testing for SQL injection:
6 # http://127.0.0.1/article='
7 [*] Dropping packet: (Endpoint: /article=')
```

Listing 5.5: Testing Pipeline for SQL and XSS.

5.2 Performance Analysis

5.2.1 Setup

For performance analysis, we used 9 clustered virtual machines in total. We used 3 hosts with an Intel Xeon e5-2630 v2 @2.6 and emulated Sandy Bridge CPUs. We used a virtual machine with

10 CPU cores and 16 GB of Random-access Memory (RAM), and 8 virtual machines with 4 CPU cores and 1 GB of RAM, connected to a 10 Gbit/s network. The first machine was used to run the BHC and the Broker and the other machines were used as clusters of workers. This setup allowed us to use up to 32 workers with a limit of 4 per cluster since each cluster was running in a machine with 4 CPU cores.

5.2.2 Host Component

The performance of the BHC was tested by measuring the transfer rate of a Nginx Web Server after attaching a pipeline to the service using all supported processing modes: inline, distributed and parallel. The number of concurrent connections was progressively increased in order to compare the transfer rate of the Web Server in the various processing modes. We used ApacheBench [7] for benchmarking and therefore measuring the transfer rate. For each number of concurrent connections (100, 200, 300, 400 and 500) we performed 10 runs of 50,000 HTTP requests. The machine with 10 CPU cores was used as the server and one of the other machines with 4 CPU cores was used as a client.

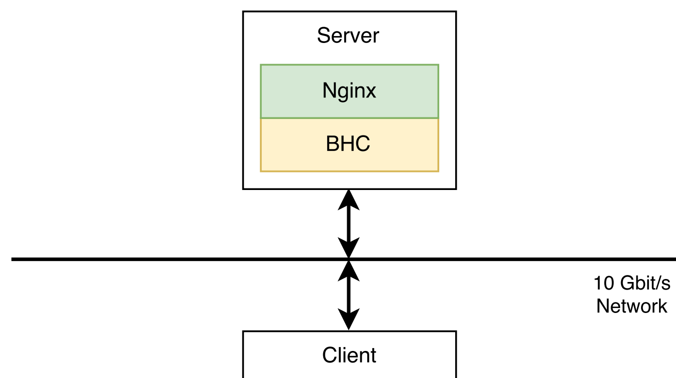


Figure 5.1: Setup for BHC Performance Analysis.

5.2.2.1 Low Processing Rate

For this test, we created a pipeline with one single node, with a simple module that checks the packet payload for a pattern. This test evaluates the relation between BHC modes and the transfer rate of the Web Server in a low processing rate scenario.

According to the Figure 5.2, we can observe that if the pipeline is in distributed mode the BHC issues a verdict faster than in inline or parallel modes, because it just needs to send a task to the BDS. The inline and parallel modes are very similar in terms of transfer rate, as expected, due to the low processing rate needed for each packet.

The overall average loss in terms of performance while using the BHC in this scenario is about 50%.

5.2.2.2 High Processing Rate

For this test, we created a pipeline with two nodes, the first one parses every packet payload into an object, which is a slow operation, and the second takes the packet object as input and checks for a pattern in the application data. This test evaluates the relation between BHC modes and the transfer rate of the Web Server in a high processing rate scenario.

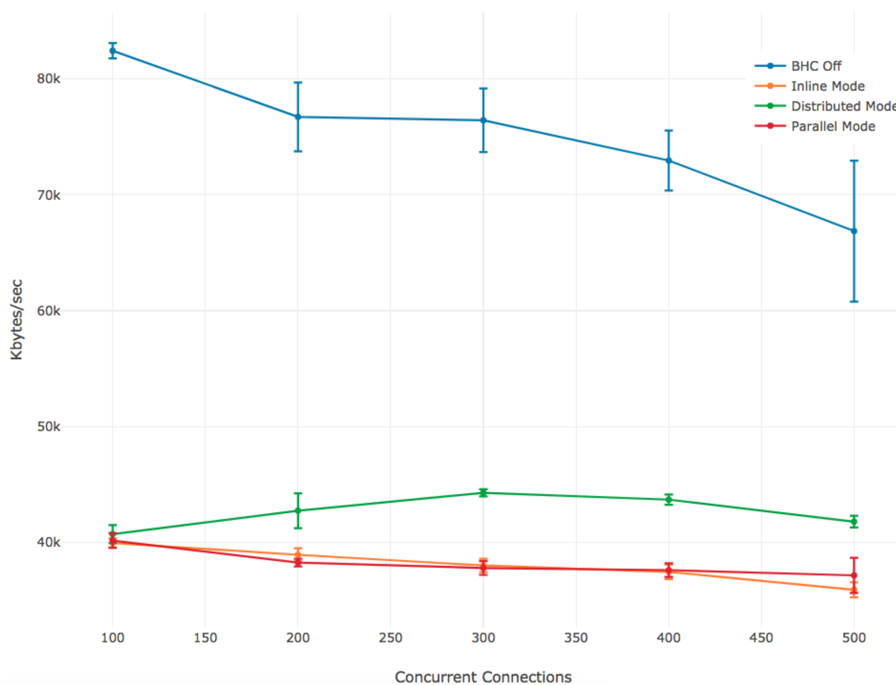


Figure 5.2: Web Server Transfer Rate - Low Processing Rate.

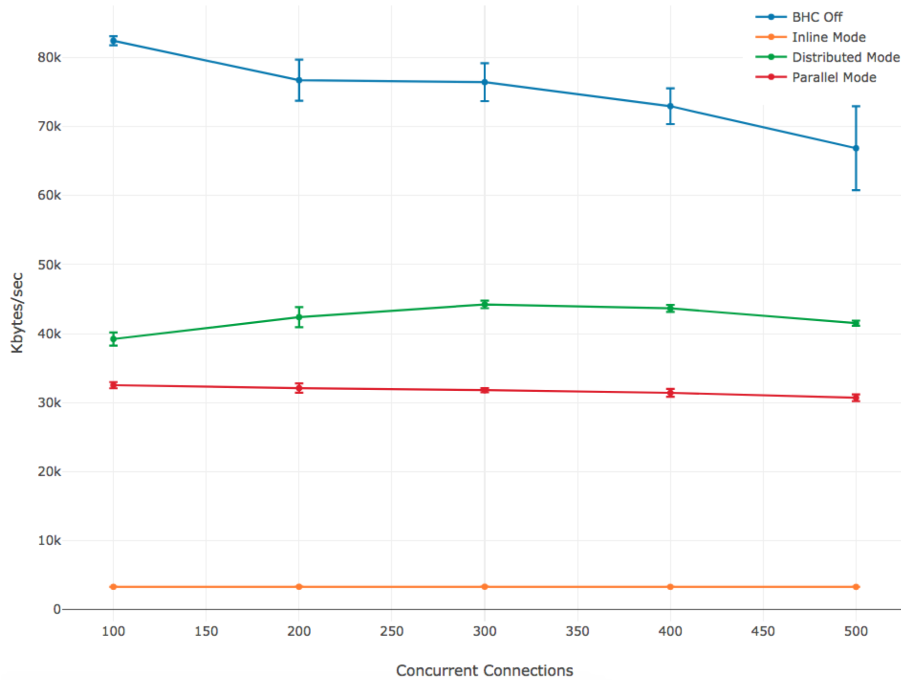


Figure 5.3: Web Server Transfer Rate - High Processing Rate.

According to the Figure 5.2, we can observe clear differences between the different modes of operation. First of all, the distributed mode behavior is the same as in the low processing rate scenario as expected. The inline mode is not so good in terms of transfer rate, since the BHC only accepts a given packet after the respective heavy task is complete. The parallel mode accepts a packet as soon as it puts it in a queue for further processing. Since the processing is done at the same time with high CPU usage, the parallel mode has a lower transfer rate in this scenario while comparing it with the low processing rate scenario.

5.2.3 Distributed System

We evaluated the performance of the BDS using two tests, one for elasticity and the other for distributed offload performance with fixed workers. Both tests used a pipeline with one module that performs 2,000,000 iterations for each packet to simulate a heavy task and increase significantly the CPU usage. We sent requests to the Nginx Web Server until the number of tasks executed by the workers was at least 75,000 for elasticity analysis and 25,000 for distributed offload performance analysis. We measured the number of tasks executed using intervals of 5

seconds and repeated each test 10 times.

The elasticity of the BDS was evaluated by comparing the performance of 8 workers with the performance of an elastic number of workers, i.e., starting or stopping instances according to the workload. The CPU and memory upper bounds were fixed in 90%, i.e., a new worker instance is started if the global average CPU or memory usage of the workers is above 90%. We used 8 clusters of workers for this test. It is important to mention that the version of *Briareos* that we used for these tests starts one worker instance by default as soon as a cluster starts. Thus, the BDS started with a pool of 8 available workers, one per cluster.

In the Figure 5.4, we can observe that the trace of the 8 workers is linear, i.e., the growth is constant. On the other hand, the elastic instances trace is superlinear, because it grows much faster than the trace of the 8 workers. Workers were started in each cluster until 32 workers became active. This feature allows *Briareos* to process more tasks in a shorter period of time and achieve a better performance.

We evaluated the distributed offload performance using a fixed number of workers. We measured the total number of tasks that were executed by the pool of workers in 5 different scenarios: 2, 4, 8, 16 and 32 workers. The Table 5.1 shows the number of clusters that we used for each scenario. As expected, the number of executed tasks is proportional to the number of workers, as shown in the Figure 5.5.

Workers	Clusters
2	1
4	1
8	2
16	4
32	8

Table 5.1: Workers per Cluster used for Distributed Offload Performance Analysis.

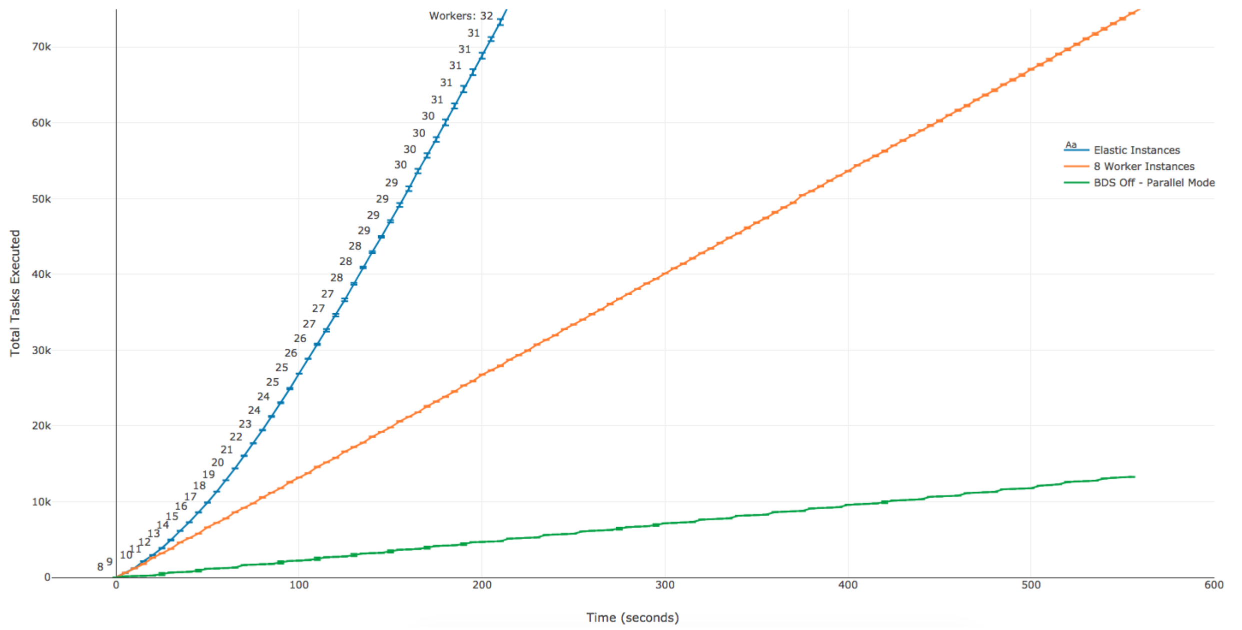


Figure 5.4: Distributed System Elasticity.

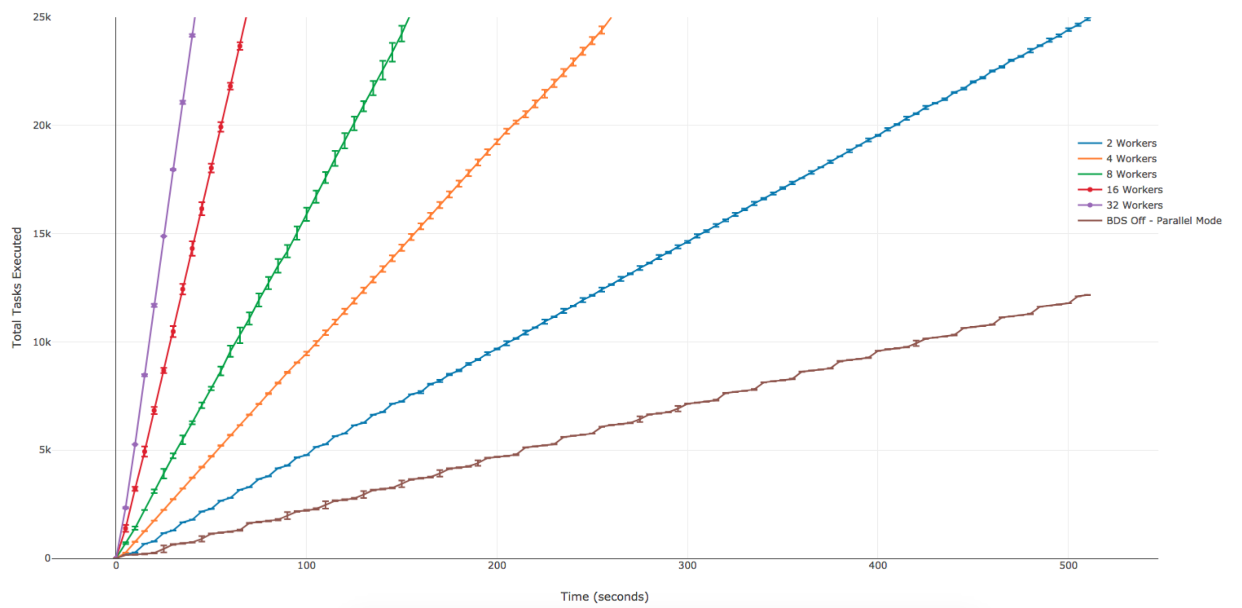


Figure 5.5: Distributed Offload Performance (Fixed Workers).

5.3 Exploit Detection Results

As described in the Section 2.3.1.3, modern exploitation techniques are hard to detect and prevent. However, *Briareos* can detect such new attack vectors using various traffic inspection techniques and monitoring services. By searching incoming traffic for code segment *.text* addresses, it is possible to automatically prevent code-reuse attacks, for example. Also, it is possible to inspect outgoing traffic in order to detect memory leaks, among other techniques.

In order to test the advantages of this framework in terms of unknown attack vectors, we built two pipelines for exploit detection. The first one is an output pipeline and contains a PID fetcher module, that returns the PID of a given connection, an address leak detector and a shell detector. The second pipeline is an input pipeline and contains a PID fetcher module, a *shellcode* detector and a ROP chain detector.

The address leak detector and ROP chain detectors use the PID of the process that is sending and receiving data, respectively, to get the memory maps of the process. Then, the network traffic is inspected in order to detect if any valid memory address is being sent. Naturally, this can lead to false positives in applications that send data corresponding to valid memory addresses, but services can use these modules if they do not usually send unprintable characters or random data, for example. The ROP chain detector leads to a smaller percentage of false positives since the valid addresses for ROPs belong generally to the *.text* and *libc* memory segments. However the address leak detector can be tuned to look only for *libc* or *heap* addresses or stack canary values, which are very common in modern exploits.

The *shellcode* detector uses the PID of a given process to get the architecture of the program and then inspect incoming packets for a minimum number of meaningful assembly instructions, using the Unicorn Engine [24] for emulation. The shell detector looks for new unintended shells, which is not the key to detect exploits but is a plus in terms of detecting the outcome of successful exploitation attempts. Our shell detector can lead to false positives if a given binary uses shell commands while executing legitimate inputs.

Any module contains three modes of operation: `block`, `replace` and `allow`. The `block` mode immediately drops packets and does not let the exploit continue. The `replace` mode can be used to confuse the attacker. Finally, the `allow` mode allows the exploit to run until the

end and saves all the packets. In the last mode, the shell detector mode can be used to detect if the exploit is complete and then block further connections and save the packets for exploit reconstruction.

We obtained binaries and the correspondent working exploits from well-known Security CTF events, such as DEFCON Qualifiers and Boston Key Party CTF. For the sake of efficiency, we did not have time to process all binaries using *Briareos*, we decided to randomly select 20 binaries and we detected 100% of the exploitations, with a 95% confidence interval of [80%,100%]. Meaning that for the full set of binaries available from past CTF competitions (n=489 to our knowledge), we would be able to detect between 80% and 100% of the exploitations with 95% of confidence. These samples represent the general vulnerabilities present in software nowadays. We tested exploits for these binaries against our pipelines. These exploits included modern exploitation techniques, such as code-reuse attacks and various heap exploitation techniques.

Modules	Detection rate
Leak detector	13/20
Leak + shellcode detectors	16/20
Leak + shellcode + ROP chain detectors	17/20
Leak + shellcode + ROP chain + shell detectors	20/20

Table 5.2: Exploit Detection Results.

The leak detector was able to detect 13 exploits. It detected 2 *.text* leaks in PIEs, 4 stack leaks, 8 heap leaks and 11 *libc* leaks. Then, we added the *shellcode* detector and we detected 3 more exploits. The number increased to 17 with the ROP chain detector. Finally, we used the shell detection module and achieved 100% detection rate for the 20 samples.

Chapter 6

Conclusion

Briareos is a disruptive framework that changes the way how intrusion detection and prevention is performed. Its modular architecture makes possible the detection of unknown attacks by combining both packet inspection and host monitoring techniques. This flexible design gives users the ability to easily build their own pipelines and modules according to their needs.

The performance of *Briareos* is reasonable in any mode, given that it was implemented in Python, which is likely to be slower than low-level programming languages. However, the BDS has clear advantages when intrusion prevention is not needed. If a given service is not sensitive, traffic processing can be distributed throughout multiple clusters in the network in order to adapt systems to workload changes. We achieved elasticity for efficient processing by starting or stopping instances, with a superlinear speedup.

Regarding unknown attack detection, we managed to detect and capture exploits that use real-world vulnerabilities such as *use-after-free* and *buffer overflows* in both stack and heap memory segments. The possibilities are unlimited and many more modules can be built in order to prevent this kind of attacks. Even machine learning techniques can be applied in order to learn models to classify a packet as malicious. Such heavy tasks can be performed in the BDS. *Briareos* has clear advantages in terms of unknown attack detection due to the ability to observe the operating system behavior and getting new knowledge from both incoming and outgoing traffic.

Briareos can be used to protect critical infrastructures and make them increasingly secure over time using novel unknown attack detection mechanisms that learn something new from each

attack. In fact, the knowledge base of the protected network is very likely to evolve and thus making systems more secure due to intelligence sharing.

6.1 Future Work

In the future, we are planning to finish the implementation of the BMS, including rule propagation, and creating a pipeline package manager for easy deployment. We are also planning to add more utilities to the *Briareos* module library, including efficient packet parsing modules and importing external rules, such as Snort rules and also YARA rules [28], for malware and known exploit detection using signatures. Also, new pipelines are still not being propagated through the workers of the BDS, unless the ZWorker docker image is rebuilt. We also did not use secure communications in ZeroMQ, but SSL should be supported in the future. Automatic framework updates should also be provided. Finally, the *Briareos* core, including the processing engine and Netfilter handlers, should be reimplemented in a low-level programming language to improve performance.

References

[1] Elasticsearch BV. Elastic. Website. https://www.elastic.co, November 2016.

[2] Jeffrey Carr. Snort: Open source network intrusion prevention. Website. http://www.esecurityplanet.com/network-security/Snort-Open-Source-Network-Intrusion-Prevention-3681296.htm, June 2007.

[3] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[4] Edward Curry, Desmond Chambers, and Gerard Lyons. Extending message-oriented middleware using interception. In *Third International Workshop on Distributed Event-Based Systems (DEBS'04), ICSE*, volume 4. IET, 2004.

[5] Lucas Vincenzo Davi. *Code-reuse attacks and defenses*. PhD thesis, Technische Universität, 2015.

[6] Roman Fekolkin. Intrusion detection and prevention systems: Overview of snort and suricata. 2014.

[7] The Apache Software Foundation. Apache http server benchmarking tool. Website. https://httpd.apache.org/docs/2.4/programs/ab.html, September 2017.

[8] The Open Information Security Foundation. Suricata tutorial. Presentation. https://resources.sei.cmu.edu/asset_files/Presentation/2016_017_001_449890.pdf, December 2015.

[9] Crippa Francesco. Europycon2011: Implementing distributed application using zeromq. Website. https://www.slideshare.net/fcrippa/europycon2011-implementing-distributed-application-using-zeromq, September 2017.

[10] iMatix Corporation. imatix - distributed systems are in our blood. Website. http://www.imatix.com/, September 2017.

[11] iMatix Corporation. Welcome from amqp. Website. http://zeromq.org/docs:welcome-from-amqp, September 2017.

[12] Docker Inc. Docker. Website. https://www.docker.com, September 2017.

[13] Open Source Security Inc. grsecurity. Website. https://grsecurity.net/, August 2017.

[14] Tripwire Inc. Open source tripwire. Website. https://github.com/Tripwire/tripwire-open-source, August 2017.

[15] UpGuard Inc. Tripwire enterprise vs tripwire open source. Website. https://www.upguard.com/articles/tripwire-enterprise-vs.-tripwire-open-source, August 2017.

[16] Rensselaer Polytechnic Institute. Rpisec - modern binary exploitation. Presentation. https://github.com/RPISEC/MBE, March 2017.

[17] SANS Institute. Host- vs. network-based intrusion detection systems. 2005.

[18] Jakub Jelinek. Object size checking to prevent (some) buffer overflows. Website. https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html, September 2004.

[19] G Khalil. Open source ids high performance shootout. February 2015.

[20] Tobias Klein. checksec.sh. Website. http://www.trapkit.de/tools/checksec.html, September 2017.

[21] Tobias Klein. Relro - a (not so well known) memory corruption mitigation technique. Website. http://tk-blog.blogspot.pt/2009/02/relro-not-so-well-known-memory.html, September 2017.

[22] Ricky Magalhaes. Host-based ids vs network-based ids. Website. http://techgenix.com/hids_vs_nids_part1/, July 2003.

[23] Mipu94. Simple plugin to detect shellcode on bro ids with unicorn. Website. https://github.com/Mipu94/BroIDS_Unicorn/, September 2017.

[24] Dang Hoang Vu Nguyen Anh Quynh. Unicorn - the ultimate cpu emulator. Website. http://www.unicorn-engine.org/, September 2017.

[25] John O'Hara. Toward a commodity enterprise middleware. *Queue*, 5(4):48–55, 2007.

[26] Michalis Polychronakis. *Generic Detection of Code Injection Attacks using Network-Level Emulation*. PhD thesis, University of Crete Heraklion, 2009.

[27] The Bro Project. Bro introduction. Website. https://www.bro.org/sphinx/intro/, August 2017.

[28] Yara Rules Project. Yara rules project. Website. https://github.com/Yara-Rules/, September 2017.

[29] Rapid7. Penetration testing software | metasploit. Website. https://www.metasploit.com, September 2017.

[30] Chad Robertson. Practical ossec. July 2011.

[31] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[32] Joe Schreiber. Open-source IDS tools overview. Website. https://www.alienvault.com/blogs/security-essentials/open-source-intrusion-detection-tools-a-quick-overview, November 2016.

[33] SecDev. Scapy. Website. http://www.secdev.org/projects/scapy/, August 2017.

[34] Siddharth Sharma. Enhance application security with fortify_source. Website. https://access.redhat.com/blogs/766093/posts/1976213, March 2014.

[35] Travis Smith. Integrating bro ids with the elastic stack. Website. https://www.elastic.co/blog/bro-ids-elastic-stack, August 2017.

[36] Robin Sommer. Bro: An open source network intrusion detection system. In *DFN-Arbeitstagung über Kommunikationsnetze*, pages 273–288, 2003.

[37] Sourcefire. Sourcefire. Website. https://support.sourcefire.com, January 2017.

[38] James Stanger. Detecting intruders with suricata. Website. http://www.admin-magazine.com/Archive/2015/27/Detecting-intruders-with-Suricata, June 2015.

[39] Suricata. Suricata. Website. https://suricata-ids.org, November 2016.

[40] Cisco Systems. Cisco. Website. https://cisco.com, January 2017.

[41] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[42] Brad Spengler & The PaX Team. Rap demonstrates world-first fully cfi-hardened os kernel. Website. https://grsecurity.net/rap_announce2.php, February 2017.

[43] OSSEC Project Team. Ossec's documentation. Website. https://ossec.github.io/docs/, February 2017.

[44] PaX Team. Address space layout randomization design and implementation. Website. https://pax.grsecurity.net/docs/aslr.txt, August 2017.

[45] PaX Team. Homepage of the pax team. Website. https://pax.grsecurity.net/, August 2017.

[46] Shellphish CTF Team. How2heap. Website. https://github.com/shellphish/how2heap, September 2017.

[47] Snort Team. Packet acquisition. Website. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node7.html, January 2017.

[48] Snort Team. Snort overview. Website. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node2.html, January 2017.

[49] Giovanni Vigna Vigna and Christopher Kruegel. Host-based intrusion detection systems. December 2005.

[50] Joshua S White, Thomas Fitzsimmons, and Jeanna N Matthews. Quantitative analysis of intrusion detection systems: Snort and suricata. In *SPIE Defense, Security, and Sensing*, pages 875704–875704. International Society for Optics and Photonics, 2013.