Pedro Miguel de Freitas Alves

Licenciado em Engenharia Informática

Analyzing Audit Trails in a Distributed and Hybrid Intrusion Detection Platform

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador: Henrique Domingos, Professor Auxiliar, DI/FCT/UNL

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março 2016

Analyzing Audit Trails in a Distributed and Hybrid Intrusion Detection Platform

Acknowledgements

First and foremost, I would like to thank *Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa* and to all professors who I had the pleasure to learn from, and who always kindly offered me their help and support throughout my journey here. Especially, I would like to thank my mentor, Prof. Henrique Domingos, for his invaluable help and for the fortunate opportunity to work with and learn from him. His insightful ideas, dedication and patience made this project possible, and for that I am forever grateful.

My family deserves endless thanks for their support and encouragement over the last years. In particular, my parents, my grandmother, and my brother who also helped me with his invaluable insights in the face of many technical difficulties.

I also want to say thank you to all my friends and colleagues for their companionship, encouragement and for making my academic journey so much more enjoyable and gratifying.

Abstract

Efforts have been made over the last decades in order to design and perfect Intrusion Detection Systems (IDS). In addition to the widespread use of Intrusion Prevention Systems (IPS) as perimeter defense devices in systems and networks, various IDS solutions are used together as elements of holistic approaches to cyber security incident detection and prevention, including Network-Intrusion Detection Systems (NIDS) and Host-Intrusion Detection Systems (HIDS). Nevertheless, specific IDS and IPS technology face several effectiveness challenges to respond to the increasing scale and complexity of information systems and sophistication of attacks. The use of isolated IDS components, focused on one-dimensional approaches, strongly limits a common analysis based on evidence correlation. Today, most organizations' cyber-security operations centers still rely on conventional SIEM (Security Information and Event Management) technology. However, SIEM platforms also have significant drawbacks in dealing with heterogeneous and specialized security event-sources, lacking the support for flexible and uniform multi-level analysis of security audit-trails involving distributed and heterogeneous systems.

In this thesis, we propose an auditing solution that leverages on different intrusion detection components and synergistically combines them in a Distributed and Hybrid IDS (DHIDS) platform, taking advantage of their benefits while overcoming the effectiveness drawbacks of each one. In this approach, security events are detected by multiple probes forming a pervasive, heterogeneous and distributed monitoring environment spread over the network, integrating NIDS, HIDS and specialized Honeypot probing systems. Events from those heterogeneous sources are converted to a canonical representation format, and then conveyed through a Publish-Subscribe middleware to a dedicated logging and auditing system, built on top of an elastic and scalable document-oriented storage system. The aggregated events can then be queried and matched against suspicious attack signature patterns, by means of a proposed declarative query-language that provides event-correlation semantics.

Keywords: *Intrusion Detection Systems (IDS), Distributed and Hybrid IDS, Analysis of Audit-Trails*

Resumo

Nas ultimas décadas têm sido desenvolvidos esforços no sentido de conceber e aperfeiçoar Sistemas de Detecção de Intrusões (IDS). Adicionalmente ao uso difundido de Sistemas de Prevenção de Intrusões (IPS) como instrumentos de defesa de perímetro de sistemas e redes, várias soluções IDS são utilizadas em conjunto numa abordagem holística na detecção e prevenção de incidentes de ciber-segurança, incluindo NIDS (IDS orientados para a segurança de redes - *Network*) e HIDS (IDS vocacionados para a segurança em nós – *Hosts*). Não obstante, a tecnologia especifica IDS e IPS, defronta-se com vários problemas de eficácia na resposta às crescentes escala e complexidade dos sistemas de informação e sofisticação dos ataques. O uso de componentes IDS isolados, focados em abordagens unidimensionais, limita a possibilidade de uma análise uniforme baseada na correlação de evidencias. Atualmente, os centros operacionais de ciber-segurança da maioria das organizações ainda dependem da tecnologia convencional de gestão de informação e eventos de segurança (SIEM). Porem, as plataformas SIEM também têm inconvenientes significativos ao lidar com fontes de eventos heterogéneas e especificas, carecendo de suporte a uma analise multinível flexível e uniforme de registos de segurança envolvendo sistemas distribuídos e heterogéneos.

Nesta tese, propomos uma solução de auditoria que, aproveitando diferentes componentes IDS, combina-os sinergicamente numa plataforma IDS híbrida e distribuída (DHIDS), tirando partido dos seus benefícios enquanto mitiga as suas ineficácias individuais. Nesta abordagem os eventos de segurança são detectados por múltiplas sondas que compõem um ambiente de monitorização pervasivo, heterogéneo e distribuído sobre a rede, integrando NIDS, HIDS e sistemas *Honeypot*. Os eventos provenientes destas fontes heterogéneas são convertidos para um formato de representação canónico e enviados através duma plataforma intermediária *Publish-Subscribe* para um sistema de dedicado de registo e auditoria que assenta numa plataforma de armazenamento orientada a documentos elástica e escalável. Os eventos agregados podem então ser consultados e comparados com padrões suspeitos de assinaturas ataques, por meio de uma linguagem de consulta declarativa que possibilita correlação de eventos.

Palavras-chave: *Sistemas de Detecção de Intrusões, IDS Híbridos e Distribuídos, Analise de Registos de Eventos*

Contents

Index of Tables

Index of Figures

Index of Listings

List of Acronyms

AWS	Amazon Web Services
CAM	Central Auditing Module
DAG	Directed Acyclic Graph
DBMS	Database Management System(s)
DDoS	Distributed Denial of Service
DHIDS	Distributed and Hybrid Intrusion Detection System(s)
DIDS	Distributed Intrusion Detection System(s)
DoS	Denial of Service
DSL	Domain Specific Language
EDP	Event Dissemination Platform
ELK	ELK Software Platform, composed by the *Elasticsearch, Logstash and Kibana* Software Components
EMMS	Event Monitoring and Management System
HIDS	Host Intrusion Detection System(s)
HMA	Host Monitor Agent
HW	Hardware
IDMEF	Intrusion Detection Message Exchange Format
IDS	Intrusion Detection System(s)
IETF	Internet Engineering Task Force
IPS	Intrusion Prevention System(s)
JRE	Java Runtime Environment
LAMP	Linux, Apache, MySQL and PHP
MOM	Message Oriented Middleware
NIDS	Network Intrusion Detection System(s)

NMA	Network Monitor Agent
SIEM	Security Information and Event Management
SOC	Security Operational Center
SPE	Stream Processing Engine(s)
SQLI	SQL Injection (vulnerability)
SSL	Secure Sockets Layer
SW	Software
TLS	Transport Layer Security

1

Introduction

Intrusion attacks on networks are today's one of the most prevalent threats to information security. Detection and prevention of such attacks on different and heterogeneous systems, as well as recovery of the caused damages, are key elements in an adequate holistic approach to cyber security. In these comprehensive approaches, different components and technology are usually involved, ranging from Intrusion Detection Systems (IDS), including the network-based (NIDS) and the host-based (HIDS), to Intrusion Prevention Systems (IPS), comprising firewall systems and perimeter defense components [Stallings 14].

Regarding IDS approaches, efforts have been made over the last decades in order to design and improve IDS solutions, either network-oriented or host-oriented, and more recently, different combinations of both. On the other hand, more or less specialized IDS solutions are often used together as tools and components in a holistic approach to incident detection and prevention. Nevertheless, specific IDS and IPS technology face several effectiveness challenges responding to the increasing scale and complexity of information systems, heterogeneity of the technology present in datacenters, and specialization of distributed applications. Those challenges also involve the possible sophistication of attacks and the difficulty to establish a correlation base, covering a complete analysis of security incidents. However, the use of isolated IDS or IPS components, each focused only on a one-dimensional approach to security-event detection, strongly limits a common correlation analysis for a more effective response.

Intrusion attacks

Intrusion threats can either come from outsider agents - someone external to organization, performing malicious activities against machines, applications or network components - or by insider agents - an otherwise legitimate user, misusing or trying to expand his or her privileges inside the system. Sometimes, these attacks also result from inadvertent activities of well-intended users by the indirect use of "malicious software components" previously installed as the result of past intrusions. The intent of the attacks can range from harmless, doing it for fun or recognition, to more obscure and harmful goals [Stallings14]. Nevertheless, intrusions are invariably undesirable, as they deviate systems from the correct specification, and even when no sensitive information or critical resources are present, intrusive actions may consume resources that must be available for correct use and for legitimate users.

Counter-measures against intrusions

There are several lines of defense against intrusion threats, ranging from perimeter defense to in-depth intrusion detection solutions; from the hardware and system infrastructure level to the application-level; and from more generic to more specific solutions in terms of granularity analysis [Stallings14, Kaufman02, Anderson08].

The first line focuses on prevention: mechanisms and services explicitly designed to ensure the correct use of systems and networks (IPS). In those mechanisms and services we include authentication and single-sign-on systems, access control, or the use of cryptography in order to protect the confidentiality and the integrity of data. In the IPS category we include systems such as network-monitoring systems with traffic analysis and traffic-shaping functions, firewalls ranging from screening-routing control, packet-filtering or packet-blocking functions to specialized application-proxy filters or specialized application-firewalls that can include the perimeter detection of malicious traffic or malicious contents. Often these mechanisms, services and systems are enough to defeat most of the intrusion attempts. However intrusion prevention is a challenging security goal, as the attacker possesses an enormous advantage over the defender, as he just needs to find one specific weak point in the targeted system, in a specific time frame, to perform a successful attack; by contrast the defender must try to predict every possible angle of attack. Therefore, it is considered wise to assume that every intrusion prevention system will eventually fail.

Based on that assumption, a second line of defence, operating independently from the previously discussed, is materialized by the evolution of

Intrusion Detection Systems [Lazarevic05, Scarfone07, Axelsson00, Mitchell14]. As initially introduced, these systems and their implementations are primarily classified in two main families: Host-based IDS (or HIDS) – focused on intrusion-detection at the host level [AIDE, OSSEC, Tripwire, SAMHAIN] – and Network-based IDS (or NIDS) – focused on traffic analysis of intrusion patterns in network protocols [Snort, Suricata]. Their purpose is to detect attempted, in progress, or accomplished intrusions attacks based on suspicious signs of malicious activity. These signs could be traces or signatures left by the attacks or detectable anomalies in the system operation patterns.

The IDS are configured to look for certain telltale signs, which could be expressed in three distinct ways:

i. As signatures expressed by rules of non-valid patterns in the correct operation of systems and networks;

ii. As signatures expressed as rules of valid patterns in the correct operation of systems and networks;

iii. Hybrid signatures composing the former approaches, with possible inclusion of admissible deviations.

SIEM platforms

Many organizations centralize their cyber-security operations in dedicated monitoring centers (usually designed as SOC - Security Operational Centers) and specialized cyber security incident teams. In such centers, SIEM technology[1] plays an important role, as frontline operational platforms. Despite that relevant SIEM technology is today reasonably effective in the detecting and helping react to frontline incidents, as Distributed Denial of Service (DDoS) attacks, it is becoming too common that more sophisticated attacks involve events occurring in different distributed components, with different anomalous operational patterns and behaviors not immediately detected as correlated incidents. Also, the increasingly sophisticated intrusion techniques use aggressive and

[1] **Security information and event management** (SIEM) is a term that describes software products and services combining security information management (SIM) and security event management (SEM). SIEM systems provides real-time analysis of security alerts generated by network hardware and applications. SIEM are sold as software, appliances or managed services, and used to log security data and generate reports for compliance purposes.

"noisy" frontline attacks as a diversion to more surgical but subtle specific attack vectors. Unfortunately, current SIEM technologies are not so effective at detecting long tail subtle anomalies, particularly in environments of large-scale distributed and heterogeneous targets, inter-related within one or more threats or one attack incident as the final manifestation of such threats.

Another difficulty of conventional SIEM platforms is their specialization in monitoring only certain targets. In general, the existent approaches fail to capture incidents and events of heterogeneous sources and have significant drawbacks in terms of openness, extensibility and scalability. Openness limitations are particularly noticed because current SIEM platforms are in general based on proprietary solutions. Extensibility limitations result from the fact that many SIEM-based monitoring platforms are not designed to evolve beyond the scope of specific monitoring targeted functions. Scalability issues come from the subjacent data-repository solutions to store events and to support query operations that are generically based on rigid storage models, such as centralized SQL relational databases.

1.1 Motivation

In general, "intrusion detection" refers to the process of identifying computing or network activity that is potentially malicious, unauthorized or unconsciously incorrect. This could be caused by misconfiguration of computers and network components including the perimeter-defense mechanisms. Most Intrusion Detection Systems have a common generic structure and components. These consist of a probing module that monitors one or more data sources capturing relevant security related events, and a processing module that applies filters and detection algorithms to the captured events [Stallings14]. They may or may not automatically react to a detected intrusion, but at least they notify the network administrator.

The following two premises broadly summarize the motivation behind effective IDS approaches:

i. The quicker an on-going intrusion can be detected, the faster the defense and recovery mechanisms can react and the lesser is the damage it can produce;

ii. The collected evidences and its subsequent investigation, allow us to understand and anticipate future intrusions.

The first consideration implies fast detection based on possible real-time constraints. The second argument, closer related to the present dissertation, suggests a soft-real-time or asynchronous analysis of multi-source diverse events. Also, it follows from this premises that a richer set of evidences (in diversity and number) contribute to a more accurate analysis from possible correlations in a complete knowledge-base of the anatomy of such potential attacks or its possible variants. Furthermore, the information gained from IDS audit trails about the attacker's techniques can be used to strengthen the first line of defense and to refine the perimeter defenses. Another advantage is the fact that systems or networks known to be armed with effective IDS solutions represents by itself a disincentive for the attacker.

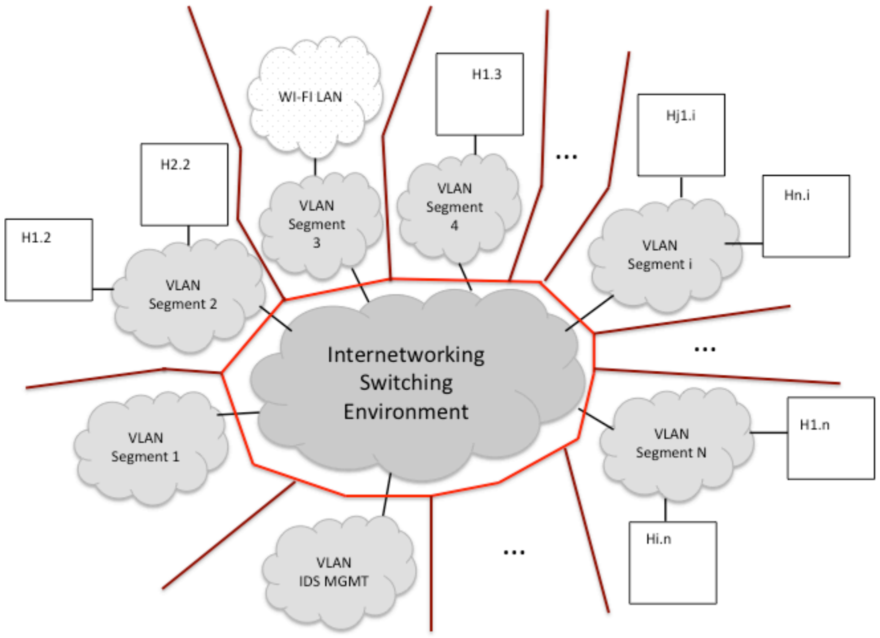


Figure 1.1: Typical internetworking environment

Current IDS technology is increasingly unable to protect the global information infrastructure due to several problems [Stallings14a, Kumar14]:

i. The existence of intruder attacks that cannot be detected based on single site observations (a single host or network segment). E.g. coordinated multiple attacker intrusions that require global scope for assessment.

ii. HIDS and NIDS technology exhibit reliability problems related to the occurrence of false positives and failures due to possible false negatives. Normal variations in system behaviour and changes in attack behaviour may cause false detection and misidentification.

iii. Detection of attack intention and trending, capturing correlated patterns and variants from previous audit trails is needed for future prevention.

iv. Advances in automated and autonomous attacks, i.e. rapidly spreading worms, require quick assessment and mitigation.

v. The sheer volume of events in large-scale or high-speed networks with a large number of interconnected hosts can become overwhelming to the IDS, causing possible losses of relevant events;

vi. Absence of aggregation and correlation mechanisms of multiple evidences that would provide more detailed information about the attack.

To address these problems related to the scale and reliability aspects, a possible direction is to consider a distributed intrusion detection platform (DIDS), composed by multiple HIDS and NIDS systems, cooperating in a pervasive and distributed intrusion monitoring environment [Kothari02, Huang09, Johnson14]. The addition of diversity by combining different HIDS and NIDS with other components providing more specific events from application-level analysis (for example, Honeypot systems) is expected to enhance the accuracy of detection reducing false positives and false negatives, resulting in a so-called Distributed and Hybrid IDS (DHIDS) [Mairh11, Kreibich04]. Such approaches require solutions to deal with the possible heterogeneity of multiple event-sources using fast local probing solutions, and the reliable transmission of events to the place where they will be analysed.

To respond to the asymmetry between effectiveness tradeoffs of IDS, IPS and SIEM-based monitoring platforms, and to better tackle sophisticated external or internal attacks, cyber security auditing functions in a large organization must be organized around two separate but collaborative functions: frontline SOCs focused on short tail (from seconds to a few hours) event series, and backline extensible SOCs. The former functions fit more easily in the adoption of standard SIEM technology that works reasonably well for the main functions involved. The latter must be based on open source data science related technology, focused on more subtle anomaly event streams that must be detected and correlated on long tail (from seconds to a few months). In the backline SOC, the use of scalable and highly available non-SQL data repositories is aligned in a direction of particular interest.

1.2 Objectives

In this thesis, we propose an auditing solution leveraging on different intrusion detection components put together and synergistically combined in a Distributed and Hybrid IDS (DHIDS) platform. The combination takes in advantage the benefits of different IDS component combined in a pervasive probing environment while overcoming their individual effectiveness shortcomings. Thus, in our DHIDS proposal, security events are detected by multiple and diverse probes spread over the network, integrating: NIDS and HIDS probes, as well as, specialized Honeypot probing systems. Events detected by these multiple sources are converted to canonical event representations, and then conveyed through an event publishing/subscribing middleware to a dedicated scalable event logging and auditing platform, built on top of an elastic and scalable document-oriented storage system. In this platform the detected events can then be aggregated, queried and matched against suspicious attack patterns, expressed as attack signatures by means of a declarative query-language. Expressed signatures represent query patterns allowing for the correlation analysis of aggregated events, originally detected by independent sources.

The objectives of the dissertation consist on designing, prototyping and testing the DHIDS proposal, addressing the identified shortcomings of the current IDS technology in order to build an extensible large-scale monitoring environment. The proposed DHIDS platform emerges from the following specific contributions:

i. Support for a pervasive environment of probing agents spread all across the network, including diversity and multiplicity in the variety of agents, exploring or leveraging from the diversity of different technological options and specializations, including NIDS, HIDS and Honeypot solutions;

ii. Materialization of a distributed publish/subscribe middleware, supporting decoupling and interoperability between the diversity in the probing environment and the auditing system where events are analysed as audit-trails of security incidents, potential threats or reported attack evidences;

iii. Materialization of an Event Monitoring and Management System, materializing an auditing platform built as an elastic auditing data repository, to store and to manage detected events for audit-trail analysis based on the aggregation and correlation of such events;

iv. Proposal, implementation and evaluation of DHIDS-QL, a query-based language to express patterns used as signatures for querying security audit-trails, as well as a runtime to interpret and execute such queries over the data-repository.

1.3 Document Organization

The remaining of this report will be organized in the following way: chapter 2 is dedicated to relevant related work regarding the different components of the proposed DHIDS platform; chapter 3 presents the DHIDS system model and architecture; chapter 4 describes the platform prototype's implementation discussing the adopted technologies and relevant challenges; chapter 5 is dedicated to the experimental observation and evaluation of the implemented prototype; finally the chapter 6 summarizes the main conclusions, addressing other open-issues and future work directions.

2

Related Work

In the previous chapter, we stated the thesis objectives and intended contributions. Now, we present the related work regarding the different dimensions involved. First, we introduce the identified dimensions; then, we present the respective related work references; finally, we briefly summarize the studied work from a critical analysis perspective to establish the design principles and main components for the DHIDS platform prototype.

The design of our DHIDS platform involves different related work dimensions that had to be conjugated for the targeted solution, namely:

- Intrusion Detection Systems
- IDS message exchange formats
- Distributed IDS approaches using Honeypots
- Message-Oriented Middleware for Event Dissemination
- Event flow analysis and correlation

These dimensions are addressed in the next sections, where we summarize relevant related work references, following the above order.

2.1 Intrusion Detection Systems

An Intrusion Detection System (IDS) monitors the actions occurred in a system or network, looks for any suspicious activity and notifies the administrator about any unexpected actions discovered.

2.1.1 Common Classification of IDS

As suggested in [Stallings14], IDS solutions can be split into two main classes, regarding the source of the captured events: Host-based (HIDS) and Network-based (NIDS). The former monitors the activity that takes place on the host, for example state changes in the hardware, operating system, file system, etc. The latter is concerned about the events occurring on the network itself, including interactions between the hosts. Both will be discussed in more detail next. In [Stallings 14], the notion of Distributed Intrusion Detection Systems (DIDS) is also introduced, a key notion for this thesis. A DIDS approach integrates both HIDS and NIDS event-sources, working cooperatively, in a distributed monitoring environment.

HIDS – Host Based Intrusion Detection Systems

The first IDS were Host-based. Their concern is to collect data from the machine where they are installed, and detect signs of possible intrusions. The HIDS main advantage is to observe the events locally on the host, which is the potential starting point of an attack. They are in that sense closer to intrusion attack detection on computer nodes than NIDS technology, making the capture of events easier and more reliable. They can observe consequences of possible intrusion attacks, even when they are conducted by attack vectors based on encrypted communications. However, when operating isolated they do not have a global knowledge of all activities, for example they cannot be aware of an eventual network scan that precedes an attack, or about attacks targeting another host. Additionally, the task of configure HIDS on many hosts in a heterogeneous environment can become considerably complex.

NIDS – Network Intrusion Detection Systems

In order to capture and analyze network events, NIDS may be installed to capture packets crossing different network segments. NIDS can be running on hosts, dedicated appliances, as well as, in connection devices like gateways or routers. NIDS technology is also available as specific HW/SW appliances, operating in the perimeter defense infrastructure. These components are also integrated in current technology for firewalls or routers (inspecting inbound/outbound traffic), as well as, in managed switches (inspecting traffic flows in aggregation ports). The main advantage of NIDS elements is their coverage, in the sense that a small number of these, strategically placed in appropriate network locations can monitor a large number of traffic originated or with destination to different hosts, even in host-heterogeneous environments.

There are however some limitations, which are naturally related to the low level at which NIDS operate. First, on heavily loaded links they may become overwhelmed by the sheer volume of the passing traffic and consequently be forced to drop packets. Furthermore, to perform a more sophisticated analysis some state information may have to be maintained, for example about ongoing TCP connections, which in turn requires more memory. It becomes obvious that there is a trade off between performance and the range of attacks investigated. Their coverage can also be impaired in a switched network environment. To overcome this problem, some techniques have been studied like embedding the sensor inside a switch or directly taping into the cables, but again the event loss problem remains. There are also attacks and evasion techniques that could be used against the NIDS. One is to launch a series of simultaneous simulated attacks in an effort to "snow blind" the sensor making it hard for the administrator to know which one was the real one. Another possible attack would be a denial of service (DoS) as the NIDS are analyzing protocols making them as vulnerable to DoS attack as the hosts. Additionally, there are ways for an intruder to bewilder the NIDS by obfuscating his actions, not showing a clear signature of an attack. For example, an intruder can perform a port scan at a very slow rate so the NIDS does not correlate each scan to the same occurrence, or use a very large number of machines to perform a distributed port scanning [Schupp00].

Finally, other problems arise regarding the use of NIDS-only approaches for a complete IDS strategy: with the increasing use of encryption, NIDS have lost access to possible significant content, hindering their ability to function well. Therefore, while NIDS have an important role to play, they can only be used effectively today as part of a broader IDS solution.

DIDS – Distributed Intrusion Detection Systems

HIDS and NIDS are focused on single-system intrusion detection functions, usually running in stand-alone machines, developed as dedicated HW/SW solutions or as SW solutions running independently in different hosts. However, in complex internetworked environments, the actions relating to an intrusion incident are often distributed over multiple network segments and in heterogeneous hosts running different applications, and therefore could be partially observed by such different components on the entire network. In a complex computing infrastructure, as found in organizations or in business-oriented data centers, it is required to defend a large distributed collection of hosts, as relevant assets that must be monitored in a global way.

Although we can mount individual defenses as counter-measures against possible intrusions, using multiple stand-alone IDS (HIDS and NIDS) running in different computers, with independent management functions, a more effective defense would take advantage of the cooperation of multiple IDS platforms installed and operating across the internetworked infrastructure. As identified by initial approaches for DIDS [Porras02], the two major requirements for their design are:

1. The ability to deal with different event formats and types (HIDS and NIDS events), as canonical representations, with data-types originally obtained from heterogeneous security related audit records;
2. Events must be obtained and efficiently transmitted with high throughput, reliability, integrity and confidentiality, from the multiple probing devices installed in particularly relevant places (for example, NIDS probes in aggregation ports of switches or close to routers' ports).

Additionally, depending on the global architecture considered in the DIDS design and the related system model, other issues may be considered:

- For centralized systems, a solution must be considered to avoid single points of failures;
- In distributed architectures, complexity comes from the requirement imposed by the coordination of analysis activities in more than one analysis node, requiring a consistent coordination.

The typical DIDS architecture can be summarized as represented in the Figure 2, where different modules are defined.

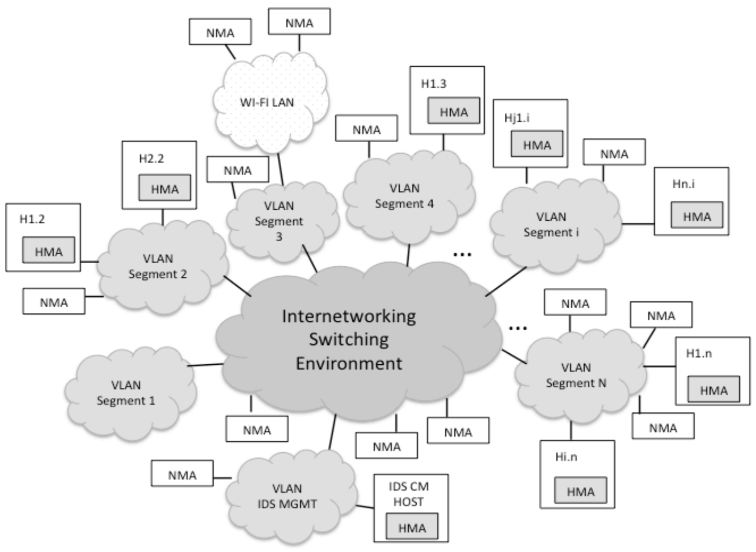


Figure 2.1: DIDS architecture example

The terminology in the Figure 2 follows the concepts and notions as described in [Stallings14], as follow:

- **Host Monitor Agent (HMA)** is a software appliance providing the probing function of a typical HIDS. Usually operating as a background process in a monitored system and collecting relevant data on security related events. It may provide local filtering, selection or aggregation capabilities (according to specific parameterizations). The HMA collects and selects data or attack-behaviors, and transmits the observed data to the CAM module (discussed below);
- **Network Monitor Agent (NMA)** is a possible hybrid appliance, possibly combining hardware, firmware and software, operating in the same way as the HMA but focusing on the observation of events mapped from parameterized LAN traffic patterns. These agents send observed events to the CAM node. NMAs correspond to the probing and parameterization capabilities of conventional NIDS.
- **Centralized Auditing Manager (CAM)** is a centralized module to process agents' collected data. HMA's and NMA's events are received and processed by the CAM Software Module, for auditing purposes. This involves the classification and correlation of events, represented as IDS auditable data-structures or audit trails. Auditable IDS trails can be analyzed in real-time, or stored as persistent audit trails, for asynchronous auditing purposes.

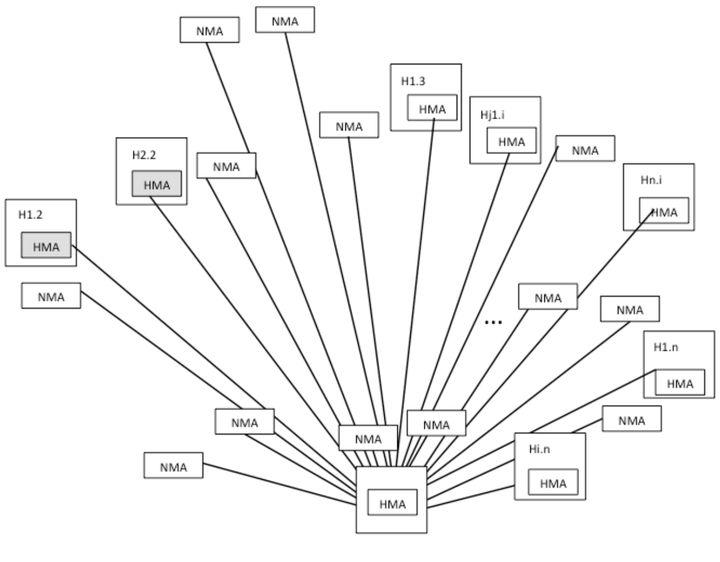


Figure 2.2: Synthetic representation of a typical DIDS architecture

The presented DIDS architecture suggests the combined use of HMAs and NMAs, as a cooperative and pervasive IDS solution. In this approach, NIDS and HIDS are put together to form an integrated distributed auditing infrastructure.

We should also notice that Figure 2 (and also Figure 3), are generic representations that map many practical DIDS approaches. From an abstract model definition standpoint, the presented abstraction is independent from the monitored network.

To summarize, the typical approach of different implementations of DIDS based on the above architectural model is a generic common base observed in different practical implementations. Sometimes, different authors address only a sub-set of that model [Wang10, Huang09, Kreibich04].

We must notice that a DIDS approach only using HIDS and NIDS probing sensors, may miss possible intrusion attacks. For example, network attacks may be missed, if the NIDS's are slow to recognize that an attack is under way, or if throughput conditions are not satisfactory for the requirements of observed network traffic – a typical situation in very large internetworking environments and high-speed networks. Analysis of network traffic at the host level provides an environment in which there is much less network traffic. On the other hand, HIDS can make use of a richer set of data, possibly using some evidences of the application level, as possible inputs for event-classification. However, it is not expected of an HIDS to be able to process other relevant information from intrusions against applications, namely if these attacks have specific vulnerabilities of such applications as preferential targets.

2.1.2 Other Classification Criteria for IDS

In literature, there are other classification criteria for IDS approaches. Among different differentiation criteria, we highlight the following: detection, timing, cooperation and reaction, as presented next.

Detection

There are two main methods for intrusions detection: misuse detection (or also called signature or rule-based detection) and anomaly detection (also called behavioral detection) [Johnson14]. Misuse detection uses previous knowledge about intrusion attacks, expressed as well-known intrusion patterns, and attempts to match current behavior against those patterns. This form of detection is more reliable, with low false-positive rates and requiring lightweight processing. The shortcoming is that the approach excludes unknown intrusion patterns [Axelsson00a] and new rules must be designed by analysis on possible future penetration identification patterns, requiring the work of cyber security specialists and who search and infer on possible suspicious behaviors. This is sometimes named Statistical Anomaly Detection, involving collections of data relating to the behavior of legitimate machines or users, over a time interval. Then statistical tests are applied to observed behavior to determine with the highest possible certainty, whether that behavior is legitimate or not. For anomaly detection, two possible techniques are used: threshold-based or profile-based detection. In the former, the detection criteria are derived from the definition of threshold values, independent of users, as metrics for frequency occurrence of different events. In the profile-based detection approach, a profile of the activity of each user is developed and used to detect changes including the behavior of individual accounts. In summary, statistical approaches attempt to define valid thresholds for the expected behavior, whereas signature-based approaches attempt to define the proper behavior [Stallings14].

Timing

This criterion refers to the time when the event analysis, and consequent eventual intrusion alerts take place. Two methods can be distinguished: "real-time detection" – the events are being analyzed as they occur, and "delayed detection" – the events are collected for later analysis.

Cooperation

The cooperation criterion is used to distinguish IDS that are natively ready to work cooperatively with other IDS instances, for example exchanging event information.

Reaction

IDS approaches can be differentiated as reactive or passive systems [Anderson08]. A passive IDS, simply detects potential intrusions, logging the corresponding observed events and may notify the administration when it detects a possible intrusion. A reactive IDS (usually associated with the notion of IPS – Intrusion Prevention System) may automatically take action after the suspicious detection of intrusion activity, for example by resetting and closing connections, shutting down systems, changing the rules of firewalls, etc.

2.1.3 IDS Platforms

We will focus on a representative set of IDS platforms that fulfill the following requirements:

1. Relevant, well-known and "de facto" widely used platforms;

2. Open-source solutions or without imposed restrictions for the possible use in the achievement of the thesis objectives;
3. Effective and modular platforms for host and network -based intrusion detection, with minimal deployment requirements.

Snort

Snort [Snort] is a free and open-source NIDS capable of performing packet logging and real-time traffic analysis on IP networks. It was initially released in 1998 and has benefited over the years from the contributions of a wide and growing developers community, evolving into a powerful tool and likely the most currently and widely deployed. It runs on three distinct modes:

- Sniffer mode, which simply reads the packets off the network and displays them in continuous stream;
- Packet Logger mode, which reads and logs the packet into disk;
- Complex and configurable mode, a mode of operating that is as a full integrated stand-alone NIDS, capable of performing protocol analysis, content searching/matching, and able to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, and so forth.

Snort performs rule-based analysis and it uses a specific lightweight language for rule description. It is also highly modularized for extensibility, providing a C API for the development of Plug-ins.

Suricata

Suricata [Suricata] is another open source solution for a rule-based NIDS solutions, released in 2010. One of its main features is its high performance and scalability provided by the multi-threaded engine built to take full advantage of multicore CPU's. It provides automatic application layer protocol identification and the rule language expressiveness goes to level of the protocol fields. It also supports file identification and extraction, based on MD5 checksums. Similarly to *Snort*, *Suricata* was also designed to be extended by plug-ins and is accompanied by developer documentation.

OSSEC

OSSEC [OSSEC] is an open-source rule-based HIDS capable of monitoring most operating systems, performing log analysis, file integrity checking, policy monitoring, and so forth. Its architecture resembles a distributed HIDS system (not to be confused with DHIDS notion, which will be discussed later) in the sense that it adheres to a client-server model with distributed agents monitoring several machines and a centralized manager. For strictly local deployment, it also can run as an all-in-one stand-alone process. The manager encapsulates a centralized administration console to supervise a large number of agents, file integrity checking databases, system logs and rules. It is also responsible for the analysis and correlation of events, which may result in a simple passive alert or the execution of an active response script, delegated to multiple hosts.

AIDE (and *Tripwire*)

AIDE (Advanced Intrusion Detection Environment) [AIDE] is developed as a free and open-source replacement for the HIDS Tripwire [Tripwire]. In terms of its sensing functionality, it is similar to the previously presented OSSEC. It works by taking a snapshot of the system and then when the system administrator runs an integrity check it detects, by comparison, the modifications, producing a report.

Summary

The following table summarizes the characterization of the presented systems, regarding the criteria introduced earlier.

Table 2.1: Classification of IDS platforms

IDS	Data Source	Detection Method	Cooperation /Extensibility	Detection Time	Reaction
Snort	Network (NIDS)	Rule/Signature based	Prepared for the addition of Plug-ins	Real-time	Passive alert
Suricata	Network (NIDS)	Rule/Signature based	Prepared for the addition of Plug-ins	Real-time	Passive alert
OSSEC	Host (HIDS)	Rule/Signature based	Agent-Manager archit. / not extendible	Real-time	Passive alert / Active response
AIDE	Host (HIDS)	Rule/Signature based	Insufficient documentation	Delayed Detection	Produces an integrity report

2.2 IDS Message Exchange Formats

In order to facilitate the development and interoperability of DIDS across a wide range of platforms and internetworked environments, it is required the definition and adoption of a standard, namely at the level of interoperable message-type formats carrying alerts or notifications corresponding to events, as well as remote parameterization commands. With particular emphasis on TCP/IP based environments, the IETF Intrusion Detection Platforms Working Group [IETF-IDWG], established since 2007, has defined relevant standardization of data-formats and exchange procedures. The most suitable standards for a DIDS or IDS cooperative approach are presented next [Stallings14]:

1. The Intrusion Detection Message Exchange Format (IDMEF) [RFC4766] defines the requirements, unifying terminology and an abstract definition for IDMEF messages;
2. The IDMEF-XML based definition [RFC4765] refers to an IDMEF data model, supported by a XML Document Type Definition and providing integration examples for interoperability;
3. The Intrusion Detection Exchange Protocol or IDXP [RFC4767] supports IDMEF interoperability on top of TCP transport, defined as an application-level protocol for exchanging data between intrusion detection entities in a DIDS model.

Another possible approach is to support IDMEF-messages as secure objects transported by SSL or HTTPS (as a IDXP/SSL proposal). Different implementations for processing RFC 4765 compliant XML-IDMEF are today freely available in a variety of languages, including *C*, *C++*, *Python* and *Java* (e.g. [IDMF-Java]), which simplifies the adoption of the format in many research contributions and projects.

2.3 Distributed IDS Approaches Using Honeypots

Intrusion detection techniques can be categorized according to the inherent type of analysis: misuse signature-based detection and anomaly detection. Misuse signature detection requires the previous knowledge about known attacks and matches current behavior against the defined attack patterns. It has the advantage that known attacks can be detected reliably and in general with low false positive rates. The shortcoming is that it cannot detect unknown and unspecified attacks. However, the introduction of Honeypots can help solving this problem [Qiao13], as it can detect an unknown attack, by the simple fact that it was "touched".

As stated previously (in 1.2), this thesis objectives focus on the design and creation of an enhanced DIDS architecture in order to deal with the conjugation of reliability, scalability and attack coverage dimensions of the intrusion detection problem. In this vision, diverse components must be combined in hybrid architectures. The heterogeneity of data sources (NIDS, HIDS and Honeypots) will result in broader attack range coverage. This extended notion of a DIDS architecture gives origin to a new term: DHIDS – Distributed and Hybrid IDS. In a DHIDS, the pervasiveness and multiplicity of sensors will promote the scalability of the solution and reliability on event collection. Additionally, the diversity in IDS platforms used will contribute to the event capturing reliability, as one system may be better suited to detected one specific kind of event and vice-versa.

Next, we delve deeper into the concept of Honeypot and after that follows a brief discussion of some concrete implemented Honeypot systems.

2.3.1 Honeypots

A Honeypot is defined as an information system resource whose purpose lies in detection of unauthorized or illicit use of that resource [Maihr11]. They are materialized by programs, machines or systems on a network, as bait for potential attackers, purposely built to attract and deceive them. Honeypots mimic real systems but don't contain, real operation data [Gollman11]. They try to feel and capture details of the attacker's intentions. When a Honeypot is "touched", the captured information can be then used to learn about the attacker's tactics, intentions and tools, for future consideration [Kreibich04]. The main difference between an IDS and a Honeypot is that the IDS just parses base activity logs (and respective type data) and selectively reports events that might be indicative of a potential intrusion; while a Honeypot is a decoy system, set up with deliberate weaknesses or vulnerabilities and usually announced with a high profile. In some sense, Honeypots are prevention systems that can generate active countermeasures against a discovered threat. At the same time, Honeypots present a way to gain insight into the process of an attack; while an IDS simply notifies that the attack happened.

Honeypots can engage in sessions with attackers at different levels of interaction and in the literature different approaches are usually characterized according to main interaction criteria [Maihr11]: LIH (Low Interaction Honeypots) or HIH (High Interaction Honeypots). LIH's are those related to the detection of automated attacks, while HIH's are focused in the support of detailed analysis of the behaviors of malware/attackers. In a LIH approach, Honeypots offer basic emulations of functions or vulnerabilities of some software services or operating systems. Complete honeypot solutions can deceive the attacker leading her/him to think that is interacting with a real system, but this requires an high degree of completeness of the honeypot in a possible emulation of a complex system. If this is not the case, the attacker can quickly identify that she/he is "speaking" or "touching" a Honeypot system. Furthermore, the development of Honeypot technology has also been accompanied by the development and availability of tools, to detect Honeypots. Then, in HIH approaches, the more sophisticated the emulation becomes, the more types of interaction behavior can be observed and logged. HIH technology involves many times replicas of real services (indistinguishable form the real operation services), but with fake data. The more interactions made possible, the greater is also the danger that the attacker can also misuse the Honeypot itself, as a staging post for launching attacks against other critical systems.

2.3.2 Honeypot Implementations

There are different implementations of Honeypot systems, in the commercial and in the free software world, with many available systems reported in recent surveys [Wang10]. The challenge behind the development of different solutions for Honeypots has been strongly inspired by three major challenges: (1) the construction of convincing Honeypots and Honeynets [Wang10], in particular when the supported behavior at the application level should be investigated (2) the protection of Honeypots themselves and (3) the extraction of novel attacks from the monitored behavior.

There are some relevant Honeypot systems, usually described in the literature as practical approaches for the emulation of application-level protocols, (e.g.: [Kippo, Dionaea]). However, more sophisticated honeypots to completely emulate the behavior of specific applications require in general a complete replica of the targeted application, with a similar behavior regarded from outside, but without real production data.

2.4 Message Oriented Middleware Systems

One relevant component for the DHIDS architecture is the Message Oriented Middleware (MOM) substrate for the dissemination of events. We studied different candidate solutions, focusing on publish/subscribe event-bus systems used for interoperability in enterprise-application platforms.

MOM systems are typically offered since the 90s as products from different vendors, such as *IBM MQSeries* [Lewis99], *Oracle Advanced Queuing* [OracleAQ] or *Microsoft MQ* [MSMQ]. In general, MOM platforms fall into two main categories [Birman05]:

1. Provisioning of network access to conventional mainframe systems and other forms of batch message delivery services when client-applications want to send messages to servers that are busy, overloaded or not currently running;

2. Message-dissemination middleware systems used as a high-level asynchronous message passing abstraction for direct use in the integration of applications.

Independently of the environment for which MOM products are more directly targeted, there are some common design principles. First of all is the asynchronous nature of the communication model, in the sense that the sending of a request to the remote destination is decoupled from the handling of its reply – as if one were sending mail to a remote server which will later send mail back containing the results of some inquiry. Second point is concerned with persistency support, mainly because the decoupling between requests and replies must consider also the possibility to deal with non-availability of the destination. This also relates with an intrinsic reliability support provided by such platforms, if it guarantees to reschedule the delivering of internally stored messages when the destination is available again or repeating the sequence in the opposite direction when a possible replay is sent back. A third issue is related in providing a well-defined request/reply API, with options to deal with different situations. These include: priority levels in message dispatching, flow-control, queue management functions, load-balancing when several processes consume from the same queues, security support allowing for message authentication, integrity and confidentiality guarantees, and fault-tolerance for long-running applications.

For our DHIDS approach, a MOM-oriented middleware is particularly interesting in the sense that it decouples the distributed pervasive intrusion-detection probes from the auditing platform where the detected events will be stored and managed for auditing purposes. In the envisaged solution is particularly relevant the use of a scalable event-bus architecture, where the MOM API is regarded as an event publish/subscribe substrate.

Publish/Subscribe (or Message-Bus) architectures are in most respects very similar to asynchronous messaging systems and may be supported by conventional MOM platforms [Birman05, Eugster03]. The only significant difference is that message-bus protocols tend to be optimized for high-speed, using broadcast hardware for example, and they typically deliver messages as soon as they reach their subscribing destinations, through some form of up-call to the application processes. Examples of message-bus architectures were implemented in well-know systems such as *TIBCO Rendezvous* [Okie93]. However the requirements of scalability, reliability, security, persistence and performance for the operation of such platforms in the internet-scale, originated the design of a new generation of message-oriented publish/subscribe systems, for fast-event dissemination purposes and remote event log processing. Among these systems, we consider systems such as *Rabbit MQ* [RabbitMQ], *Amazon SQS* [AmzonSQS], *HornetQ* [HornetQ], *MongoDB* [MongoDB] or *Apache Kafka* [Kreps11]. These systems can be studied from different classes of criteria as usually addressed for extensive comparative purposes of Message-Bus systems [Eugster03].

As a starting point, any of the above solutions can be regarded as a good candidate. All of them address the base decoupling support considering space and time criteria, as well as, synchronization conditions and reliability guarantees. Anyway, to drive a rational for the choice of the best one, several other requirements must be particularly taken into account. Some of the requirements can be analyzed from a comparative analysis of the different solutions. Other requirements are imposed by external conditions. We established the following order of main criteria for our analysis and our final choice:

1. High persistence
2. Possibility of scale out (in a possible cluster solution)
3. Adequate performance and event throughput
4. Publish/subscribe API for fast integration with *Logstash* [Logstash] (an "event-manager" used in the CAM)

From a study previously addressed on the candidate solutions for the thesis preparation phase (out of scope of the dissertation itself) [Costa14] and from complementary observations on published work on the evaluation of the different candidate solutions [Eugster03], we summarize the main issues behind the decision to use the *RabbitMQ* platform. *RabbitMQ* exhibited the best conditions for high-persistency guarantees, with a possible deployment using a cluster-based architecture, as well as, good indicators in disk access processing and communication throughput, addressing also our second requirement.

According to [Costa14], *HornetQ* has good performance and provides a rich messaging interface with different routing options, offering different tuning alternatives for message scheduling. Although these advantages, the gains in performance are not considerably better when compared with *RabbitMQ* or *Amazon SQS*. As a tradeoff, *HornetQ* don't have support for scalability. *MongoDB*, support replicated message queues for scalability, but comparatively it appears as the worst solution in terms of performance. *Kafka* appears in different comparisons as the best solution for performance, but the gains compared with *RabbitMQ* or *SQS* are not so considerable for our expected input requirements. *Amazon SQS* can be comparable with *Kafka* in performance when multiple SQS nodes are used, but to achieve the same performance more resources are required. On the other hand, *SQS* is a solution especially designed for the *AWS* cloud, with minimal setup required for *AWS* cloud-based applications (an interesting issue but not particularly relevant for us).

Considering the criteria more directly related to external conditions, *RabbitMQ* is seamlessly compatible with *Logstash*. Additionaly, *RabbitMQ* is well-known and already adopted in the context of ongoing research collaboration projects involving the NOVA LINCS Research Center and its Computer Systems Group and external industrial partners (e.g., Amazon AWS and Portugal Telecom), namely in the domain of event-dissemination in the context of cyber-security auditing tools.

From the previous analysis and a the study on performance of different adoptable MOM solutions [Costa14] we decided to use *RabbitMQ* as the best balance on the different above tradeoffs, as the reference solution to leverage the development and materialization of the DHIDS event-dissemination substrate[2]. At the same time, *RabbitMQ* can support transparently the event-dissemination process over HTTP or HTTPS over TLS v2, allowing for its transparent use in internetworking security environments, with no changes for example in firewall management policies and configurations.

2.5 Event Flow Analysis Methods

This segment is dedicated to the study of event logs processing and analysis approaches. It includes the following topics:

- Analysis of events based on information flow control,

[2] This decision was made during the thesis preparation phase as the result of previous specific assessment criteria involving different alternatives [Costa14], and due to other institutional reasons indirectly related to the specific context of the thesis. These relate to other ongoing projects being developed in PT Comunicações S.A. in the direction of design options to build and to consolidate a common MQS infrastructure that must be shared by different SIEM monitoring platforms. In the context of a collaboration and partnership between DI/FCT/UNL and PT Comunicações S.A., it was decided that the contributions of this thesis would be aligned with that standardization effort, as a step to align the implementation effort in the thesis elaboration with the future possible adoption of the thesis' contributions in the context of PT Comunicações S.A.

- Event tracing in systems oriented for dynamic instrumentation,
- Event flow processing in stream databases and,
- Event processing systems specifically oriented to intrusion detection.

2.5.1 Event Analysis Based on Information Flow Control

Information flow is the process of information transference from one place to another. That could be for example, data read from a file by a process or an object shared between different processes. In this context, Information Flow Control is a technique used to trace the information flow inside a system, as explained in [Blankstein11]. In short, this consists of tagging each piece of data in the system with security labels, which are sets of tags. The processes manipulating this same data object become "contaminated" (associated) with these tags and as processes communicate with each other, tags flow accordingly.

In [Blankstein11] the author presents a system for analyzing security event logs, also called audit trails. This is designed to run on top of an already existing distributed security platform named *Aeolus* that generates audit trails following an information flow control scheme. The idea is to gather these audit trails generated by *Aeolus*, represent them using a convenient model (preserving the information flow relationships) and export an interface that facilitates an efficient further analysis of these events in order to detect and trace information misuses. The system supports two distinct features. First, it provides an interface for directly querying past events, using SQL. The second is a publish-subscribe interface for registering watchers for receiving, in (almost) real time, events that match pre-specified filters. A watcher is a client object that receives event notification from a logging facility. Upon registration, the watcher supplies the logging server with some filters that define the events he is interested in, expressed using a domain-specific language.

The presented system deals with events produced by a distributed platform that monitors the user processes running on top of it. Our source of events is the probing agents (acting like sensors) running IDS software and forming a pervasive mesh in the network. The concept of the watchers as a client objects and the language used for event filtering provides us meaningful inspiration as we face similar challenges. Although, we intend to go one step further, as we do not only want to filter events individually but also determine correlations between them (this aspect will be further studied in 2.5.3 – Stream Processing Engines).

2.5.2 Event-tracing in Dynamic Instrumentation

Instrumentation refers to the techniques of implanting blocks of code into a program in order to monitor its behavior as it executes (e.g. to measure its performance, diagnose errors or write logs). Dynamic instrumentation means that it occurs at run-time as opposed to static instrumentation where the analysis code is injected (and the executable rewritten) before the program runs. Event tracing in active log monitoring environments is related to work focused in generic solutions found in dynamic instrumentation systems. These systems require interfaces for specifying events to monitor (as dynamic configurable filters) and actions to take. Conceptually, instrumentation consists of two components: the code that is executed at each instrumentation point and a mechanism that decides where to insert the analysis code. We are interested in studying specifically the latter. Many techniques for dynamic instrumentation and event tracing systems are available in the literature, from which we studied a representative set.

Pin [Luk05] is a framework for dynamic binary instrumentation, which enables the creation of customized dynamic analysis tools, known as *Pintools*. These specify code (written in C/C++) to insert in arbitrary places of the target program. This is done by intercepting each binary instruction of the executable, and generating and running identical code "on-the-fly", providing the opportunity for the *Pintool* to run its own code. The original target application code is only used as reference and the code that actually runs is the one generated by Pin. The interface provided to user is by means of a hook function in the Pintool that is called by Pin every time a new instruction is encountered. The user can then inspect the instruction and specify actions to be executed.

DynamoRIO [Bruening03] is a dynamic tool for runtime code manipulation. This resembles the previously studied, *Pin*. It exports an interface for building dynamic customized tools where the client supplies the specific hook functions to deal with the events. On the other hand, it is not limited to the insertion of instruction points (i.e. adding code), as it also allows for arbitrary modifications to the instructions. Furthermore, the granularity regarding the generation of events also differs, as it notifies the client for each straight-line code sequence (or basic block) as opposed to generating an event for each instruction.

DTrace [Cantrill04] is a dynamic instrumentation and monitoring platform. Several instrumentation methods are supported, one of them being "system call tracing", which allows the user to place instrumentation points at the entry and return of any system call invoked in a single host. To specify which system calls should be instrumented and what actions are to be executed, it uses a C-like domain-specific language (or DSL).

AspectJ [Kiczales01] is an extension of *Java* for aspect-oriented programming. Its relevancy for this section is motivated by one of the its features - dynamic crosscutting - which makes it possible to instrument java code, be it our own code, for enhanced modularity, or already compiled classes. There are several possible well-defined points of insertion - designated by join points - which represent specific occurrences in the program execution, for example when a method call is received by the target object or when an object is initialized. The injected code is specified in a method-like structure where the signature contains an expression that matches one or many join points.

2.5.3 Stream Processing Engines

Stream processing engines (SPE) are systems whose purpose is to address some of the limitations of the traditional database management systems (DBMS) in order to better suit stream-oriented applications, like monitoring applications. As opposed to business-oriented applications, monitoring applications have to deal with large continuous streams of events, usually real-time, and to perform computations such as filtering – remove unwanted events; correlation – detect patterns across different events; and aggregation – compute aggregation functions values. These requirements render the typical DBMS unsuitable for the task of process high-volume data streams in order to extract useful and actionable information to monitoring applications in real-time. The solution we propose is also in its essence a monitoring system, intended to process continuous streams of events, originating from multiple sources, and analyze them based on correlations between each other. In that sense, SPE's are closely related to our problem. In this class of systems, there are many different and relevant approaches. Here, we focus on a representative subset.

Aurora* and *Borealis are two similar and related SPE's presented in [Abadi03] and [Abadi05] respectively. In both approaches, the queries over streams are expressed in the form of data flows, known as query diagrams. These are "boxes and arrows" diagrams, composed by multiple operators, and can be expressed and stored in a XML file. There are two kinds of operators: stateless operators - meaning they perform operations one tuple at a time maintaining no state between tuples - and stateful operators - which rather than processing tuples in isolation, perform computation over groups of input tuples. A relevant feature of stateful operators is the possibility of processing windows of data that move with time, in which both the window size and the sliding increment are parameterizations. Besides extending some of the functionality introduced by its predecessor *Aurora*, the main idea behind *Borealis* is to perform stream processing in a distributed and scalable manner. *Borealis* also allows for dynamically modifying queries and to "time travel" in order to query past events.

Cayuga* and *SASE+, presented in [Demers07] and [Agrawal08], are interesting approaches on this topic, offering rich query languages for matching complex event patterns. The idea of *Cayuga* is to provide a general-purpose system for high performance, on-line pattern matching of complex events on a large scale, offering an expressive and composable query model. Queries adhere to the publish-subscribe paradigm, each representing a pattern of events and publishing it under a given name that a client may subscribe to, or that can be used as input for another query. They are expressed in an SQL-like syntax but introducing some interesting new constructs, allowing to correlate events over time. *SASE+* shares mainly the same claims as *Cayuga*. Despite the language differences, it also provides a rich declarative language for event correlation and an efficient implementation for high throughput. *SASE+* provides a very powerful and elegant declarative language for pattern matching over event streams, presented in [Agrawal08, Diao07], which we find quite adaptable to our vision and which is worth highlighting. This contains constructs for expressing sequences of events, Kleene closures (an indefinite number of similar events), event negation (test for its absence) and event filtering techniques

Relating to our own system, we do not use any of these solutions directly in our event processing as we choose to use a distributed and elastic event storage system instead, promoting scalability. Nevertheless, these approaches provide invaluable inspiration and a model for our own pattern matching system, as we will see in chapter 3. It is also worth mentioning that our approach is not particularly targeted for real time event processing and analysis, but more for deferred analysis of previously stored auditable event trails.

2.5.4 Event-Flow Processing Applied to Intrusion Detection

This thesis objectives include the exploration of an event-flow processing technique to support querying and analysis of the audit trails corresponding to those events. There has been some significant work regarding the processing of audit-trails in the context of intrusion detection and recovery platforms. Between those, we emphasize the references below.

RETRO [Kim10] is an intrusion recovery system. In *RETRO*, after an intrusion is discovered (by the network administrator, perhaps with the help of an IDS, or by noticing a suspicious file, or by other means), usually some actions must be taken in order to undo the effects of the attack. In other words, the system must be restored to a state that mirrors the system as if the attack had never happened, loosing none of the legitimate information and modifications in the meantime. *RETRO*'s solution is to periodically make checkpoints of the system state and to log all the subsequent actions, in the form of a directed acyclic graph (DAG), which it then uses to trace an intrusion incident to its origin. From there, it uses rollbacks and re-execution in order to repair just the compromised objects in the system, with minimum user input. The graph represents the system execution history over time. In the graph, nodes represent the system objects, like files and processes and each edge represents an action by an object over other, for example a process writing to a file. An action has a set of dependencies which are the objects accessed or modified by it.

BackTracker [King05] is another approach based on DAG's to trace a detected intrusion to its origin point. The system analyzes operating system events from system logs, to determine the source of a given intrusion on a single system (by looking for targeted intrusion patterns in the persistent logs). The events are then represented as nodes in a DAG, where each edge in the graph represents a causal relationship between the nodes. When an intrusion is discovered, *BackTracker* traverses the event DAG backwards from that discovery point searching for the source of the intrusion.

Summary. What we found most relevant in the *RETRO* and *Backtracker* approaches, considering the context of our work, is the graph model used and the analysis in pursuance of the origin of the intrusion. Our system must be able to analyze events occurring across a distributed set of IDS sources. Consequently, backtracking graph analysis is more difficult because we must deal with different audit logs and these logs may branch significantly. For example, a user may want to trace the source of a "write event to a file" but also "a specific TCP packet" in a network segment supporting a SSH session related to that write. The analysis starts by examining the process that wrote to the file and any inputs to that process could have contributed to the write. These inputs could be file reads, remote procedure calls, accesses to shared state, or startup parameters and we must follow from this if events are related to a remote SSH session logged in the system and traced in a network segment where the host is located. Those inputs require further analysis of what processes and what network traffic may have affected them. Additionally, our solution should offer query support for audit trails, expressed as potential attack signatures.

There are some important differences comparing our objectives with the previous approaches: the analysis performed in the *BackTracker* proposal is also different from active monitoring or direct querying. While our envisaged platform is more focused on discovering misuses or violations from defined patterns expressed as "malicious activity patterns", the *BackTracker* approach is more concerned with discovering how specific misuses occurred in a specific node. This idea is suitable for our approach. Despite that in our case we have the additional problem of dealing with multiple distributed heterogeneous NIDS and HIDS sources (as a Distributed and Hybrid Intrusion detection Platform), requiring some form of previous local "filtering" functionality, as well as, previous event-type conversion, before the adoption of a global graph representation, as proposed in the *BackTracker* system. Similarly, the *RETRO* system was not particularly designed as an Intrusion Detection system.

2.6 Related Work Summary and the DHIDS Approach

Revisiting the generic DHIDS architecture, as previously represented in Figure 2.2, we have addressed the initially identified dimensions regarding the design and implementation of a DHIDS prototype using diverse probing devices to materialize a distributed and pervasive probing environment. We will now highlight the lessons learned from this chapter as a starting point for our DHIDS design model.

We began to study a representative set of the IDS implementations now available, which can be leveraged in a heterogeneous and pervasive monitoring environment. These included NIDS – where *Suricata* [Suricata] and *Snort* [Snort] have demonstrated to be viable options as implementations instances of NMA components in our conceptual architecture, running in dedicated small devices (e.g. *RaspberryPi*); and HIDS – installed in potential target machines as implementation instances of HMA components (e.g. *Tripwire* [Tripwire], *AIDE* [AIDE] and *OSSEC* [OSSEC]). Then, we demonstrated that Honeypots could be a valuable addition to that same heterogeneous monitoring structure.

This pervasive DHIDS architectural model, promotes the extensibility, scalability and availability attributes. For it to be effective, the appropriate conditions for reliability and persistence of detected events must be ensured, until they reach the CAM for later analysis. For the purpose, we addressed the implementation of the Message Oriented Middleware substrate leveraging the *RabbitMQ* system and implementing adapters for IDMS event-formats. As we will explain in the DHIDS system model (described in chapter 3) and implementation (in chapter 4), we used the IDMEF base standard, adopting a JSON data-model for the implementation of the base IDMEF-XML data-model specification [RFC4765].

The Central Auditing Manager (as defined in the conceptual architecture) subscribes IDMEF events notified by all above IDS (NMA and HMA) and Honeypot components. For the CAM level, we designed a query based analysis environment on top of an elastic document-oriented event storage system. The stored events can then be matched against known attack-signature patterns by means of a declarative DSL (presented in chapter 3). The studied stream processing engines (specifically *Cayuga* [Demers07] and *SASE+* [Agrawal08, Diao07]) provided the general model for that language.

3

System Model and Architecture

This chapter presents the system model and the architecture for the proposed Distributed and Hybrid Intrusion Detection System (DHIDS) – we will name it so from now on. In this chapter, we focus on the design considerations and related specifications. Later, on chapter 4, we will describe the specific implementation options for the DHIDS prototype.

We begin with the overview of the DHIDS design model and architecture in section 3.1, followed by an explanation in more detail of each component, namely the Pervasive Probing Environment (explained in section 3.2), the Event Dissemination Platform (section 3.3) and the Event Monitoring and Management System (explained in section 3.4). Finally, we present the design and specification of the DHIDS Query Language (DHIDS-QL) for attack-signature expression.

3.1 System Architecture Overview

The DHIDS system is conceptually divided in three parts, as represented in Figure 3.1: the Pervasive Probing Environment, the Event Dissemination Platform and the Event Monitoring and Management System.

The Pervasive Probing Environment comprises a set of independent and heterogeneous probing agents spread over the network, observing and capturing security-related events according to their specialization and local configuration. The relevant events detected are then encapsulated in a common format and dispatched to the Event Dissemination Platform.

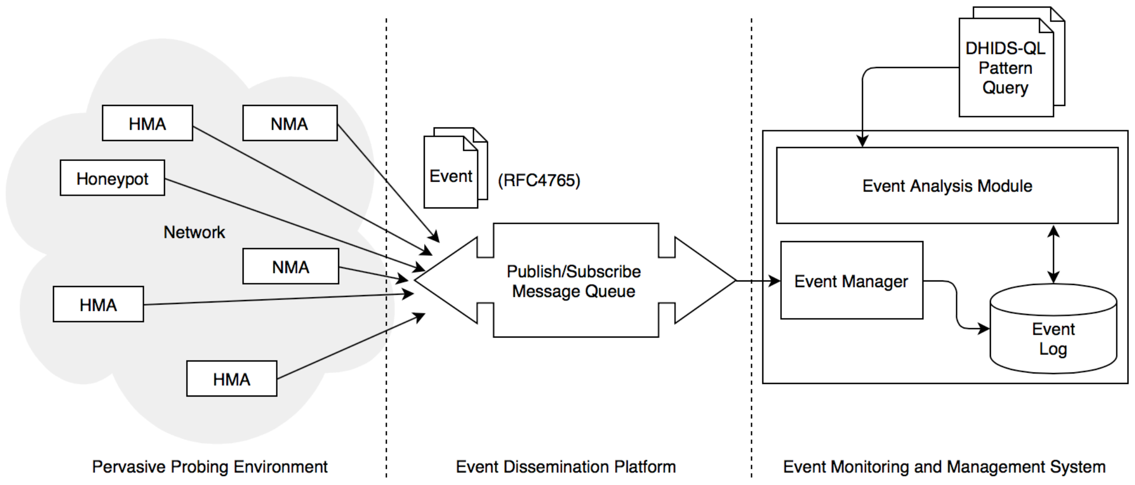

HMA
NMA
Honeypot
Network
NMA
HMA
HMA
Event
(RFC4765)
Publish/Subscribe Message Queue
DHIDS-QL Pattern Query
Event Analysis Module
Event Manager
Event Log
Pervasive Probing Environment
Event Dissemination Platform
Event Monitoring and Management System

Figure 3.1: System macro-components overview

Conceptually, the Event Dissemination Platform is designed as an event-bus, publish/subscribe substrate for the entire DHIDS architecture, providing reliable and asynchronous delivery (with persistency message-queuing properties). Published events are subscribed by the Event Monitoring and Management System (EMMS), specifically by the Event Manager and are then merged and stored in a scalable logging system. This logs and maintains agent-detected events, and results of "a posteriori" aggregations of correlated events. Then, these events can be queried according to known intrusion attack patterns or suspicious behaviors, expressed in DHIDS-QL. These queries are interpreted and executed over the elastic log by the Event Analysis Module.

3.2 Probing Agents

The Pervasive Probing Environment is composed by "in-the-field" agents, whose function is to collect data. Agents can act as NMA, HMA or honeypots. Therefore, the placement strategy in the network for the specialized probing agents is a key decision of system and network administrators, as it has implications on the span of detectable activity patterns.

3.2.1 Classes of Agents

Regarding the conceptual characterization and specialization of agents, our DHIDS approach adopts the DIDS terminology used in [Stallings14], with the addition of another category – the Honeypot. The diversity of the probing environment is described as follows, using the notion of classes of agents, namely: NMA, HMA and Honeypots.

- **Network Management Agents (NMA)** capture the traffic-related events occurring on a network segment. These are the conceptual equivalent to the sensor capabilities of typical NIDS platforms. According to its specializations, NMAs materialize on-line NIDS-based probing functionality, to deal with the network observation in real time. This may include traffic analysis at different protocol levels, and filtering based on local configuration rules to decide if an event is relevant to be sent to the EMMS.
- **Host Management Agents (HMA)** capture events occurring on specific hosts. HMAs correspond to the probing capabilities of HIDS solutions in monitoring the internals dynamic behavior and state of a system. They may detect which program accesses what resources. For example, it can be detected that a specific process has suddenly and inexplicably started, modifying the system resources (for example the password database or security critical configurations). So, depending on each specific materialization, HMAs may look at the state of the system, its stored information, monitor local log files and check that the contents of these appear as expected. Each HMA in the DHIDS platform, according to its specialization, monitors whether anything or anyone, whether internal or external, has circumvented expected system's security policies.
- **Honeypots** are systems purposely deployed as bait for potential attackers. It simulates a potential target while firing security events as noticeable evidences when anyone interacts with it. Therefore, Honeypot agents are related to the typical event-logging capabilities of specific Honeypot systems, as introduced in section 2.5.

We must notice that each of these classes is associated with a specific function and agent specialization. We can also have diversity of different implementations for the same function. For this reason, our conceptual model for DHIDS includes this diversity dimension, which is particularly related to the heterogeneity of the probing base, captured by the word “Hybrid” in DHIDS.

3.2.2 Generic Agent Architecture

As DHIDS components, each probing agent (independently of their specialization) can be abstractly viewed as generic architecture composed by four sub-components, as represented in Figure 3.2:

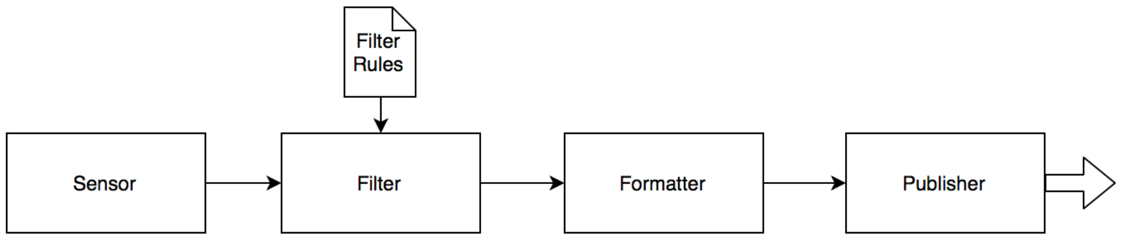


Figure 3.2: Generic agent's internal architecture

- **Sensor:** It is the agent's data collecting component. It gathers information (potential intrusion evidence) from the environment where it is implanted. Examples of input data could be network packets, system logs, application logs, etc.
- **Filter:** The events collected by the sensor are then handed to a filtering component set by a local parameterization interface to specify rules. It selects or discards each element of the captured data set, and returns a subset of the observed events according to defined rules.
- **Formatter:** The events are then passed to a formatting component responsible for the standardization and encapsulation of the event data into the message format as required to satisfy the interoperability model supported by the defined IDMEF message format specification [RFC4765].
- **Publisher:** This component implements the publishing interface with the Event Dissemination Platform and is responsible for all the necessary steps to publish the captured, filtered and formatted events.

3.3 Event Dissemination Platform

As initially presented, the Event Dissemination Platform is a highly scalable "event bus" responsible for the carriage of events in the DHIDS platform. It is supported by a message-queuing substrate providing a publish-subscribe interface. The messages comply with the IDMEF interoperability model (described in section 3.3.2).

3.3.1 Requirements for the Message Queuing Support

The Event Dissemination Platform simultaneously supports decoupling, reliability and scalability. These characteristics result in several tangible benefits.

It facilitates the growth of the Pervasive Probing Environment, simplifying the addition of new specialized probing agents, as long as they support the defined message interoperability model and the implementation of the publishing interface. As the agent network and consequently the volume of events grow, the dissemination platform should scale accordingly. Decoupling also promotes the future interoperability with other systems.

Asynchronous message queuing with persistency is a desirable feature considering that the Pervasive Probing Environment on one side and the Event Monitoring and Management System on the other side, can publish and subscribe the events, each at its own pace, and events are not lost in between.

It also has the added benefit of facilitating the implementation of different queuing policies if needed, for example based on priority, to tweak the system performance.

Of course, to prevent the system from becoming itself target of attack, the Event Dissemination Platform must to be supported by secure message-queuing channels. Specifically all the communications should be SSL-encrypted safeguarding the dimensions of confidentiality, integrity and authenticity of messages. For this purpose, the required public key infrastructure (PKI) should be in place and complying with the X.509 standard, in order to support public-key certificates used by all the components involved in publishing and subscribing the disseminated events.

3.3.2 Interoperability Model

The interoperability model for the DHIDS Event Dissemination Platform complies with the IDMEF standard [RFC4765] (Intrusion Detection Message Exchange Format). Event messages moving through the event bus should be encapsulated into this format, as the canonical representation of events published by the agents and subscribed by the Event Monitoring and Management System.

The IDMEF standard was adopted as the interoperability reference in order to enable the potential interoperability between commercial, open-source, and research systems, allowing for the integration of different components into the same IDS solution. This also has the advantage of facilitating the extension of the DHIDS Probing ecosystem. Figure 3.3 generically represents the structure of the IDMEF message model.

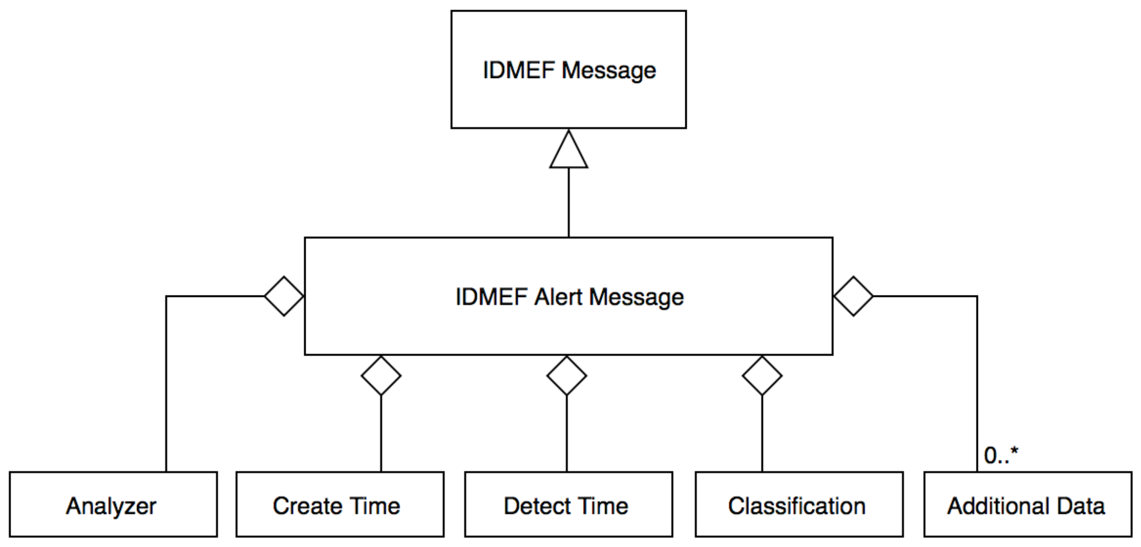


Figure 3.3: RFC4765 IDMEF message model

Due to the heterogeneity inherent to all kinds of security-related messages and alerts, a universal data model has to allow for flexibility, which makes it somewhat complex, comprising several optional and required fields. Here, for simplicity, we focus on a partial representation of the IDMEF data model, considering only the standard-required fields and the additional ones that are required in the context of this solution. A complete specification of the data model can be found in [RFC4765].

IDMEF defines more than one type of message. In our model, we only consider messages of type *Alert*, with the following fields:

- **Analyzer**: Identifies the analyzer (the agent) that originated the alert.
- **Create Time**: The time the alert was created.
- **Detect Time**: The time the event leading up to the alert was detected. In some circumstances, this may not be the same value as Create Time.
- **Classification**: A “tag” that identifies what the alert is; in our specific case, it indicates the type of the event (e.g. a network-related occurrence).
- **Additional Data**: This is a format-free data field including all the specific event details (that don’t fit in any of the other fields). The agent defines the format of the data in this field.

Although the Additional Data is a format-free data field, the implementation should guarantee EMMS component is able to parse all the implementation-specific possible formats and convert them into a “queryable” data structure. The XML-Specification according to the IDMEF [RFC4765] can be found in Appendix A.

3.4 Event Monitoring and Management System

The Event Monitoring and Management System stores and analyses the events collected by the agents, looking for suspicious activity in the form of patterns. These patterns, expressed in DHIDS-QL, are explained in section 3.5.

3.4.1 Architecture

As illustrated in Figure 3.4, this component consists of the following subsystems:

1. **Event Manager:** it handles the subscription and reception of messages from the Event Dissemination Platform. Upon reception, it passes each event to the Event Pre-processor.
2. **Event Pre-processor:** it is responsible for the de-encapsulation of the events, parsing and preliminary processing. According to its type and source, each event is parsed (including the unstructured Additional Data field), and the fields sanitized, producing well-formed structured and "queryable" event, which is then placed into the Elastic Data Storage.
3. **Elastic Data Storage:** component that assures the persistency of events. It provides document-oriented storage with high availability, scalability and read/write performance. As the system has to deal with all the events that the agents generate, its storage capacity should be able to scale according to the size and possible growth of the monitored network, the increased volume of traffic and the extensibility of the probing environment. For that reason, this component should be implemented on top of a distributed and elastic key-value store based platform.
4. **Data Access Layer:** provides basic query conveniences over the Elastic Data Storage, allowing the insertion and retrieving of sets of events according to a certain filtering criteria.
5. **Event Analysis Module:** receives and parses the user-defined specifications of attack pattern expressed in DHIDS-QL and searches for pattern-matches in the event log. (This will be presented in more detail in 3.4.3).

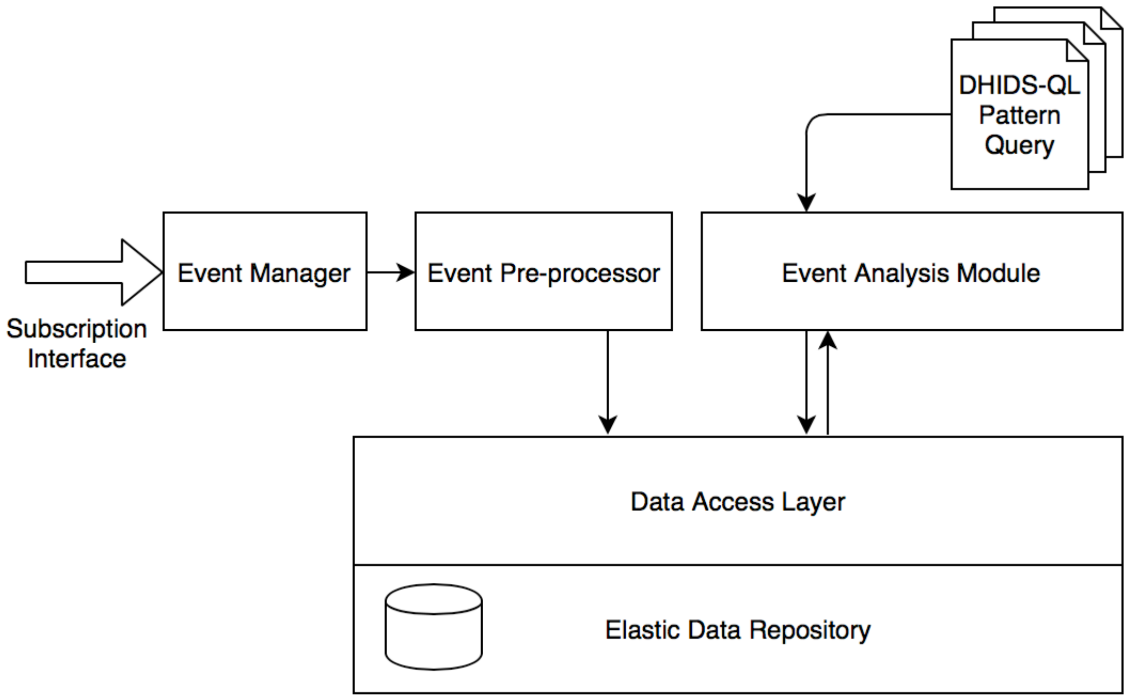


Figure 3.4: Event Monitoring and Management System (EMMS)

3.4.2 Event Model

Events in DHIDS belong to one of the following two categories:

- **Atomic**, the events produced by the agents, which represent an atomic occurrence in the monitored system.
- **Aggregated**, the events created as output of queries executed by the administrator. The idea underlying the representation of query results as events is that they should represent the semantics of higher-level events. For example, the event "TCP-handshake" could be a higher-lever representation of three lower-level events, the IP packets exchanged back and forth. Once represented, these higher-level events can then be used as input events in subsequent queries (as we will discuss later in section 3.5).

Storage Model

Upon reception by the EMMS, the events are pre-processed and stored in the Elastic Data Storage component, where they reside organized in indices. The notion of indices derives from the data repository model adopted – document-oriented. An index is identified by a name, indexing a collection of documents related to events sharing similar characteristics. There may exist any number of indices with indiscriminate names defined by the administrator. Although, we believe that one convention should be adopted: atomic and aggregated events should not share the same index. This rule may prove useful by simplifying the querying task and query readability as it makes this distinction clear.

When stored, we still need a way to discriminate events by type and source, know the time of occurrence and possibly the agent who detected it. For those reasons, it makes sense to adopt the same structure of the IDMEF Alert message model [RFC4765] for storage. That way, we preserve the "header" of all events in the same standard form. The event-specific information will be stored under the tag "Additional Data". All of these data representation intricacies will be transparent for the user, as the pattern query language will provide that abstraction (as explained later in 3.5).

Atomic Events

All agent-generated events should come with a type, explicitly defined in the *Classification* attribute (Figure 3.3), which is an IDMEF required field. Polymorphism is possible; as for example an event of type "TCP" could also be seen as an "IP" event. In this case, different types should be separated by dots (.) as in "Packet.Ethernet.IP.TCP". It is the responsibility of the Event-Preprocessor to validate the Classification field, as well as the other required fields ensuring that a well-formed event is stored. As we will see later, the types of events must be explicitly defined when querying a pattern.

Aggregated Events

An aggregated event encapsulates the lower-level events that gave origin to it in the first place resulting in the following additional fields stored under the tag "Additional Data":

- **Events**: the lower-level events that match the pattern in this particular instance.
- **Analysis Description:** optional field containing a user-defined message, resulting from the pattern-matching query.

This type of event (like any other event) complies with the IDMEF-based storage standard. Therefore, the "header" fields contain the following information:

- **Analyzer:** contains the signature of the Event Analysis Module;

- **Detect Time:** the Detect Time of its first occurring lower-level event;
- **Create Time:** the time the analysis took place;
- **Classification:** a user-defined category (expressed in the query) relating to the semantic meaning of the pattern. For example, if a query specifies an unsuccessful TCP handshake, a suitable Classification for the matching occurrences could be "IncompleteTCPHandshake".

3.4.3 Event Analysis Module

The Event Analysis Module is a sub-component of the EMMS. It is designed as a set of processing components to perform searches over the event log maintained in the Elastic Data Storage. Such components are: Query Parser, Query Validator, Query Plan Builder, Query Runtime and Data Access Manager. The Event Analysis Module interacts (via the Data Access Manager) with the Elastic Data Storage though the Data Access Layer.

The query processing support allows searching positive matches of attack patterns specified by the DHIDS-QL syntax, as represented in the Figure 3.5. The figure illustrates the processing workflow that each query goes through. Each step is executed by the corresponding component in the Event Analysis Module.

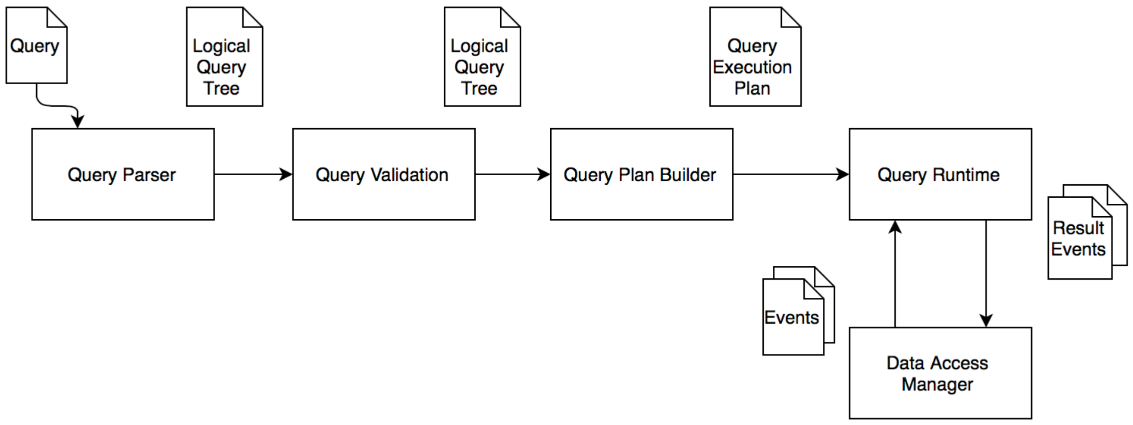


Figure 3.5: Query Processing Workflow

According to the Figure 3.5, the Query Processing Workflow is composed by the following phases:

1. **Parsing:** The Query Parser receives the query, converts it to the corresponding logical query tree, asserting that it is syntactically correct, and passes it to the next module.
2. **Validation:** The Query Validation Module decides if the query is valid, before execution. It performs all the necessary checks, (e.g. type-checking) and if needed, for normalization purposes, it may even rewrite some branches of the query tree.
3. **Planning:** The Query Plan Builder receives the validated and normalized logical query tree and produces a query execution plan. This plan consists of an ordered set of tasks including fetching, filtering and result-processing. Fetching should precede filtering tasks and result-processing tasks should happen after all the others have taken place.
4. **Execution:** The Query Runtime receives the execution plan and executes each task in chain. Each task receives as input, the output produced by the previous one. The fetching tasks imply the retrieving of events from the Elastic Data Storage (via the Data Access Manager). Filtering tasks are computations performed over stored data produced by the previous tasks. The result-processing tasks include printing the results in a console or writing the resulting events as a higher-level event, correlating its base relevant events (as high-value event "mashups" aggregating a set of lower-level discrete ones).

3.5 DHIDS Query Language

The purpose of DHIDS-QL is to provide a powerful, elegant and concise way of expressing the anatomy of potential intrusion attacks in the form of event patterns and search the log for occurrences of those patterns. As far as we known, there is no specific "attack-signature" expression language commonly accepted as a standard. Inspired by the related work on pattern-based query languages, we adopted *SASE+* [Agrawal08, Diao07] as the foundation for the design of our own. *SASE+* was originally proposed as a rich pattern-matching language, which resembles an SQL-like syntax, specifically oriented for pattern matching on event streams. Taking the original language definition as starting point, we introduced slight tweaks, defining the embryonic "attack-signature language" we call DHIDS-QL. In this approach, queries take events as input from the log (either atomic or aggregated events) and produce one output event for each distinguishable manifestation of the specified pattern.

In the next sections, we will present DHIDS-QL, starting with the prerequisites and then explaining the query structure and formalisms, accompanied by illustrative examples. Finally, we go through the more advanced constructs of the language for additional expressiveness. A complete abstract syntax specification can be found in Appendix B.

3.5.1 Language Prerequisites

The envisioned language should clearly express the traceable signatures of attacks in the form of patterns, in a highly concise and readable manner. Specifically, it should provide means to:

1. Select relevant events;
2. Express correlations between different events, through the specification of pattern structures (e.g. sequencing) and the use of complex predicates;
3. Test for the non-occurrence of events;
4. Express a finite but unbounded number of similar events;
5. Express admissible time windows for the pattern occurrence;
6. Compute aggregation values;
7. Compose queries that make use of the results of previously executed queries.

3.5.2 Query Structure

Next, we present the basic query structure (in Listing 3.1), which we then dissect and explain the meaning of each of its main components.

```
PATTERN <pattern structure>
[WHERE <matching condition>]
RETURN <output-event type>;
```

Listing 3.1: Basic query structure

The PATTERN Clause

The clause PATTERN announces a pattern specification. To express a pattern structure that represents a sequence of events, the operator SEQ should be used. Each event is declared with an explicit type and a unique identifier. Listing 3.2 defines the SEQ operator used to represent a sequence of events.

```
SEQ ( <type1> <id1>, <type2> <id2>, ... )
```

Listing 3.2: SEQ operator structure

The WHERE Clause

If applicable, a WHERE clause should follow, expressing the event selection and correlation conditions, using the previously specified identifiers. This uses a syntax similar to SQL. The following tables present the available logical, arithmetic and comparative operators. Parenthesis can also be used to compose complex predicates.

Operators

The following three tables summarize the logical (Table 3.1), arithmetic (table 3.2) and comparison operators (Table 3.3) available:

Table 3.1: Logical Operators

AND	Conjunction
OR	Disjunction
NOT	Negation

Table 3.2: Arithmetic Operators

+	Addition
-	Subtraction
*	Multiplication
/	Division

Table 3.3: Comparison Operators

=	Equals

<	Less than
>	Greater than
>=	Greater or equal
<=	Less or equal
<>	Different than
CONTAINS	Only applicable to strings. Returns true if and only if the preceding string contains the succeeding string.

Literals

There are three types of literal values: String, Number and Boolean and they are represented as Table 3.4 illustrates:

Table 3.4: Literal Values

String	Represented between quotation marks (“”). E.g. “foo”.
Number	Integers represented as sequence of digits. Negative numbers are represented by the precedence of a minus signal (-). E.g. 14, 0, -1.
Boolean	TRUE, FALSE, true, false, 0, 1.

Fields

To reference an object’s field the dot (.) notation is used, as demonstrated in Listing 3.3.

```
<event identifier>.<field name>
```

Listing 3.3: Field referencing

The RETURN Declaration

Each positive pattern-match found produces a higher-level event (nonetheless, with similar characteristics to any other event) representing one specific occurrence of the queried pattern and encapsulating the original events that gave origin to it. The RETURN declaration is one of the two possible ways to finish a query, which relates to the level of persistency of the output events. The other will be addressed later in 3.5.3 where the differences will be discussed. For now, with the RETURN declaration, this newly created event exists only temporarily, in memory, and can be “seen” only by subsequent queries in the same session.

The semicolon (;) indicates the end of a query, which allows us to batch multiple queries in the same script.

An Example

Listing 3.4 represents a concrete example of a query, which expresses the pattern of opening a TCP connection between two hosts:

```
PATTERN SEQ ( TCP e1, TCP e2, TCP e3 )
WHERE e1.tcp.flags = "syn"
   AND e2.tcp.flags = "syn-ack"
   AND e3.tcp.flags = "ack"
   AND e2.ip4.destination = e1.ip4.source
   AND e1.ip4.destination = e2.ip4.source
   AND e3.ip4.source = e1.ip4.source
   AND e3.ip4.destination = e1.ip4.destination
   AND e2.tcp.ack = e1.tcp.seq+1
   AND e3.tcp.seq = e2.tcp.ack
   AND e3.tcp.ack = e2.tcp.seq+1
RETURN "TCPConnectionOpen";
```

Listing 3.4: TCP handshake pattern (query example)

The SEQ construct indicates the specification of a sequence pattern: three events of type TCP. The WHERE clause expresses the event-matching predicate for the previously defined events: the definition of the TCP three-way handshake. Finally, the RETURN declaration specifies the resulting event’s type.

3.5.3 Chained Queries and Persistency

Queries can be executed directly on a console session or from a pre-written script file. The former is convenient for conducting quick analysis on querying event patterns directly, to immediately print the matches on the screen. The latter is more suitable for rules specification for frequently ran queries. Both methods allow chaining queries together. This feature is useful for break up complex queries into smaller ones, taking as input the results produced by the previously executed queries.

Another important distinction is about the level of persistency of the output events, which can be:

1. **Volatile**: when the events are resulting of the RETURN declaration. Output events live temporarily in the environment, being available to be used as input in subsequent queries. However, they will be discarded at the end of the querying session.
2. **Persistent**: when the events are indexed, in the event log. This is achieved with the use of the PUBLISH-IN declaration.

Evaluation Environment

The query evaluation environment temporarily stores the results of previously executed queries, so that they can be reused as input for subsequent chained queries. These exist during the query session or the script execution, being discarded thereafter.

The PUBLISH-IN Declaration

If the output events are to be made persistent, the query should be finalized by a PUBLISH-IN declaration inspired in *Cayuga*'s language [Demers07]. This specifies the type of the output event and names the index where it should be stored (instead of using the RETURN statement). Listing 3.5 defines the query syntax with the PUBLISH-IN declaration:

```
PATTERN <pattern structure>
[WHERE <matching condition>]
PUBLISH <output-event type>
IN <index>;
```

Listing 3.5: Persistent result query (PUBLISH-IN)

Scripts

A script is a file containing a sequence of queries. These are useful for executing complex queries that are broken into smaller ones, or to store frequently used queries. A script executes in its own isolated environment. Therefore, at least its last query should be a "PUBLISH-IN" query, so the results can be made persistence.

3.5.4 Rules and Alerts

Remembering chapter 2, the notion of rule is very familiar among the existing IDS systems, specifically the signature-based kind. The idea is to have pre specified rules running on background and, when a rule is triggered, an alert is raised. In this approach, we leverage the already discussed querying mechanism as system of rules. These can be seen as queries that are permanently running on the background and produce an output event every time a new occurrence is detected (as opposed to simple queries which run once by user command).

This is accomplished with the use of scripts. A rule is added to the system by placing a script (which may be composed by one or more queries) in a pre-specified rules-directory.

The output events of rules may be directed to a specific index in the log reserved to alerts. Additionally, a "listener" component may be connected to logging system in order to monitor that index and actively notifies the administrator when a threatening alert event occurs (e.g. by email, SMS, or some other form of message).

3.5.5 Additional Constructs

Event Negation

The negation construct is inspired in *SASE+*, with the same meaning and for the same purpose. It verifies the non-occurrence of an event. It is expressed by the addition of the tilde (~) before the event declaration.

As an example, if we wanted to capture a situation of an incomplete TCP connection-opening request typical of a SYN-Flood attack, that could be done by tweaking the previous example (from 3.4.2), as follows in Listing 3.6. The modifications appear in boldface font. With this query we detect the occurrence of the pattern: SYN-request from the client, followed by the SYN-acknowledge from the server, without the existence of the rightful final acknowledge from the client. When it happens repeatedly, it could mean a SYN-flood denial-of-service attempt.

```
PATTERN SEQ ( TCP e1, TCP e2, ~(TCP e3) )
WHERE e1.tcp.flags = "syn"
 AND e2.tcp.flags = "syn-ack"
 AND e3.tcp.flags = "ack"
 AND e2.ip4.destination = e1.ip4.source
 AND e1.ip4.destination = e2.ip4.source
 AND e3.ip4.source = e1.ip4.source
 AND e3.ip4.destination = e1.ip4.destination
 AND e2.tcp.ack = e1.tcp.seq+1
 AND e3.tcp.seq = e2.tcp.ack
 AND e3.tcp.ack = e2.tcp.seq+1
PUBLISH "TCPConnectionIncomplete"
 IN "threats";
```

Listing 3.6: TCP handshake incomplete (query example)

Kleene Closure

The Kleene Plus construct also comes from *SASE+* and enables the representation of one or more occurrences of a particular event type. It is denoted by a plus sign (+) after the event type declaration, accompanied by the square brackets "[]" after the event instance name, which denotes a list of events.

The next example query (Listing 3.7) expresses the pattern of denial-of-service (DoS) attacks targeting one particular host. As demonstrated, the Kleene closure is usually used in conjunction with aggregation functions. *count()* gives the number of element in the list (more on this next in Aggregation Functions). The WHERE clause here, illustrates the "every" semantics in Kleene closure (where $i,j \geq 1$).

```
PATTERN SEQ (TCPConnectionIncomplete+ c[])
WHERE c[i].e1.ip4.destination = c[j].e1.ip4.destination
HAVING count(c) > 100
RETURN "SYNfloodAttempt";
```

Listing 3.7: SYN-Flood attack pattern (query example)

Aggregation Functions

The example above demonstrates the use of an aggregate function – *count()* – already familiar from SQL. In this example the PATTERN and WHERE clauses generate the possible pattern matches. The HAVING clause specifies the pattern filtering condition. There may exist the following aggregation functions:

Avg():	returns the average of a set of values. E.g. *avg(packets[].length)* gives the average length of the packet array.
Sum():	returns the cumulative sum of a set of values.
Count():	returns the number of element in a set of values.
Max():	returns the largest of a set of values.
Min():	returns the smallest of a set of values.

Time Window

The last example may be further refined in order to consider the time dimension, using the WITHIN construct, also inspired in *SASE+* (Listing 3.8). The PATTERN, WHERE and WITHIN clauses select the possible event matches for this pattern: all the *TCPConnectionIncomplete* events targeting one host within the time frame of 1 hour. The HAVING then filters the possible pattern to just the ones that involve more than 100 occurrences.

```
PATTERN SEQ (TCPConnectionIncomplete+ c[])
WHERE c[i].e1.ip4.destination = c[j].e1.ip4.destination
WITHIN 1 hour
HAVING count(c) > 100
RETURN "SYNfloodAttempt";
```

Listing 3.8: Time window specification (query example)

Index Specification

By default, the pattern-matcher algorithm will look into all indices for the specified type of event. However, a specific index can be selected as input using the clause IN, in the event declaration. This mechanism, in conjunction with the PUBLISH-IN declaration, facilitates the process of composing queries on top of each other, giving finer control to the administrator. Listing 3.9 illustrates this situation.

```
PATTERN SEQ (TCPConnectionIncomplete+ c[] IN "threats")
WHERE c[i].e1.ip4.destination = c[j].e1.ip4.destination
HAVING count(c) > 100
RETURN "SYNfloodAttempt";
```

Listing 3.9: Index specification (query example)

Description

The DESCRIPTION clause, following the PUBLISH-IN declaration, allows us to specify a custom “description” string field to the output event. Listing 3.10 shows an example using the DESCRIPTION clause and PUBLISH-IN declaration.

```
PATTERN SEQ (TCPConnectionIncomplete+ c[])
WHERE c[i].e1.ip4.source = c[j].e1.ip4.source
HAVING count(c) > 100
WITHIN 1hour
PUBLISH "SYNfloodAttempt"
  IN "attacks1"
  DESCRIPTION “SYN flood attempt from host ” +
  c[1].e1.ip4.source;
```

Listing 3.10: PUBLISH-IN with description (query example)

4 Implementation

Taking DHIDS design model and architecture as a starting point, this chapter presents the implementation prototype. It begins with an overview of all implemented components (section 4.1). After that, the three following sections address to the implementation details about the technology behind the components of the specific implementation domains, namely: the Pervasive Probing Environment (4.2) the Event Dissemination Platform (4.3) and the Event Monitoring and Management System, integrating the query language and the implementation of its runtime support (4.4).

4.1 Implementation Overview

Our prototype instantiates the system model as discussed in chapter 3. Figure 4.1 illustrates the implementation components and technology used.

Pervasive Probing Environment

The network-probing agents (NMA) are implemented by leveraging the probing components provided by the *Snort* [Snort] and *Suricata* [Suricata]. To implement the functionality of Host-Based Agents (HMA), we leveraged *Tripwire* [Tripwire](open-source version). Finally, to implement the Honeypot agents, for evaluation purposes, we used two strategies: (1) a vulnerable web-server attack target website – *Wackopicko* [Wackopicko] – running on *Apache* and *MySQL* (from which we extracted logs); and (2) a representative agent fed with the logs extracted from a vulnerability scanning and attack injection tool tested against a web server (under attack).

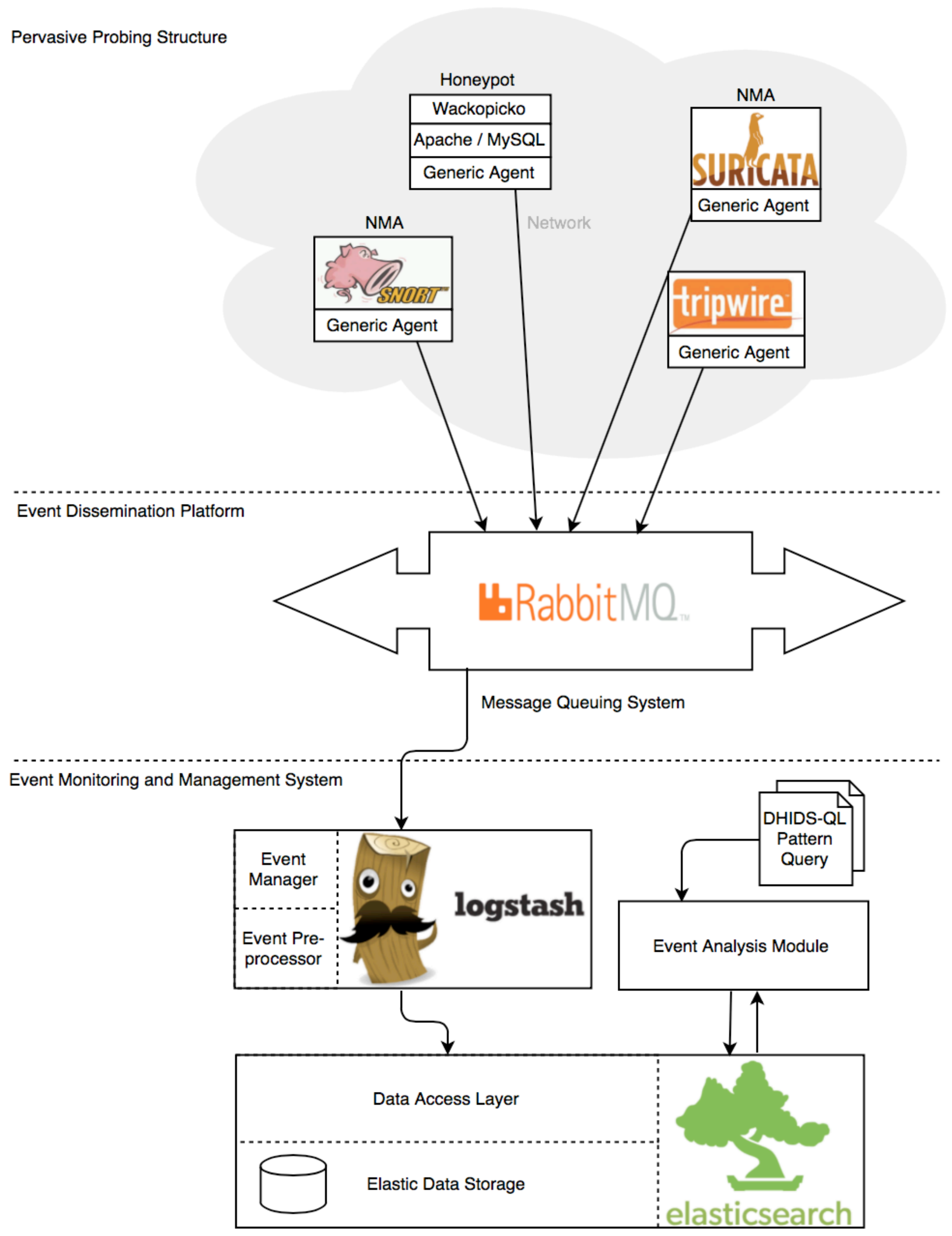


Figure 4.1: Implementation Overview

Event Dissemination Platform

The Event Dissemination Platform in the DHIDS prototype was implemented by the *RabbitMQ* message queuing system [RabbitMQ], initially discussed in chapter 2.

Event Monitoring and Management System (EMMS)

The EMMS implementation materializes the components initially described in chapter 3. The Elastic Data Storage and the Data Access Layer are supported by *Elasticsearch* [Elasticsearch], an efficient and elastic document-oriented database with support for powerful analytics capabilities. With this approach, we also benefit from the possibility for seamless integration with the variety of products from the *Elastic* family, most important being the *Logstash* and *Kibana* (forming the *ELK* stack[1]). Both Event Manager and Event Pre-processor's functions are handled by *Logstash* [Logstash]. It is used as an event parsing and formatting pipeline, seamlessly compatible with both *Elasticsearch* and *RabbitMQ*. This implementation options offers the immediate advantages of using code-shipping and relevant on-going developments as services integrated and compliant with the base *Elasticsearch* technology[2]. For example, we can use *Kibana* [Kibana], for data querying, exploration and graphical visualization over *Elasticsearch* data, making the development of a monitoring environment with centralized dashboards for the DHIDS proposal easier to address. The Event Analysis Module and their internal components were fully implemented in *Java 1.8*.

4.2 Pervasive Probing Environment - Agents

In our DHIDS implementation, we used a representative set of parameterized well-known HIDS and NIDS systems, as well as vulnerable web application in place of Honeypots, suiting our intended test scenarios. As this is a beginning approach to the DHIDS prototype, we did not need, for evaluation purposes, a fully automated production mechanism for real-time capturing and publish of events. Instead, we used pre-captured data sets from the aforementioned IDS and honeypot tools, to be "re-played" and published as observed events. This strategy enables us to use available real world sets of events (some,

[1] *ELK* stands for *Elasticsearch*, *Logstash* and *Kibana*. The term was coined by *Elastic*, the company behind these open-source projects, as it is aparently very common for these three components to be deployed together as part of the same log analysis solution.

[2] In https://www.elastic.co/products there is a list of these relevant products, ranging from security enhancements, system-administration facilities to data visualization and management functions (accessed on 2/Feb/2016).

containing attacks) and reproduce them at will in a controlled environment. Please refer to Appendix C for the list of the used data sets.

Generic Agent

The aforementioned approach also saved us the time of implementing several IDS-specific agent adapters for the DHIDS platform. Instead, we developed and use several instances of a Generic Agent, which is a component capable of reading many different types of logs (*PCAP*, *MySQL*, *Apache*, Syslog, any JSON or XML represented events, etc.) and publish them as standardized events, according to the defined interoperability format, in the Event Dissemination Platform. The Generic Agent was fully developed in *Java 1.8*. It consists of about 1900 lines of code and 64 classes. It also makes use of several libraries and open-source projects, as follows:

• ***jNetPcap***	Used to read *PCAP*-formatted files into the Generic Agent. Version used: 1.4 [jNetPcap].
• ***org.json* library for *Java***	Used To manipulate JSON documents in a object-oriented way in Java. Used version: 2015-05-01 [JSON-java].
• ***com.cr_labs.rfc4765* lib**	*Java* implementation of the data structure introduced in RFC4765. We used to help convert event events to the interoperability format specified in chapter 3. Version: not specified. [IDMEF-java]
• ***RabbitMQ* Java Client**	*Java* API to communicate with the *RabbitMQ* server. Used to publish the events. Version used: 3.4.4. [RabbitMQ]

IDS Systems

In our prototype, we used *Snort* (v. 2.9.6) [Snort] and *Suricata* (v. 1.4.7) [Suricata] as NMA agents. We configured both to produce *PCAP*-formatted logs. To obtain HMA-related events, we used *Tripwire* (v. 2.4.2.2) and instances of our earlier mentioned Generic Agent to read collect events directly from application logs. To implement an example of a Honeypot we used *Wackopicko*, a web application with vulnerabilities used to test vulnerability-scanning tools.

Low-cost card-sized computers

One key notion underlying our model is the pervasiveness of agents. It is therefore particularly interesting to implement NMAs and Honeypots as standalone credit-card-sized low-cost autonomous computer nodes, which can be deployed anywhere in the network. For this purpose, we selected *RaspberryPi*'s (both 1B and 2B) [RaspberryPi], running *Raspbian Wheezy* (kernel version 3.18) [Raspbian] to implement NMAs and Honeypots for event collection, as well as Generic Agents to process pre-captured logs in a test environment.

4.3 Event Dissemination Platform - RabbitMQ

RabbitMQ (version 3.4.4) [RabbitMQ] is an open-source distributed and reliable message-queuing system. In this implementation, it was configured to provide publish/subscribe support, for many producers – the agents distributed in the probing environment – and one consumer – the Event Monitoring and Management System (EMMS).

As stated earlier, we implement an interoperability format for event dissemination. We use JSON as encoding format to implement the base RFC4765 message format. This induces a smaller encapsulation overhead in comparison with XML and demands less processing by the Event Manager (*Logstash*), which doesn't need to convert it to JSON (as *Elasticsearch* document storage model is based on JSON).

Another feature provided by *RabbitMQ* is an acknowledgement mechanism – Confirms (aka Publisher Acknowledgements) – as an extension to the AMQP 0-9-1 protocol, that ensures messages are not lost. This slightly impacts the performance, but as we will see in our experimental evaluation in the next chapter, the overhead is almost negligible. This mechanism may be activated or not by the publishers, i.e. the agents. Preferentially, we publish events using Confirms mode.

4.4 Event Monitoring and Management System

The EMMS is the most complex part of the implementation. Here we explain how we handled the challenges of data storage, event management and event analysis.

4.4.1 Data Storage – *Elasticsearch*

We used *Elasticsearch* (version 1.7.1) [Elasticsearch] for event storage. We consider this an adequate choice as it suits the model's requisites, has a good reputation for performance and scalability, and it also enables seamless integration with *Logstash* – an event log processor – and *Kibana* – an analytics and visualization tool, useful to chart and summarize event data.

In this particular implementation, we used it as a schema-free document database (the default) for simplicity. This means that *Elasticsearch* automatically detects the structure and data types of the event's JSON document and indexes it "as is", as the agent sends it, in the IDMEF format. Nonetheless, it is possible (by configuration on *Elasticsearch*) to force the documents to comply with one or many rigidly specified schemas.

4.4.2 Event Management – *Logstash*

Logstash (version 1.5.5) [Logstash] handles the functions of the Event Manager and Event Pre-processor. It is a data pipeline for log management, which gathers log events from a variety of sources and parses them into a structured format set by configuration. As mentioned, *Logstash* also provides event parsing and formatting features, which are configured via a sequence of filters. For simplicity, we are not using any relevant filter (besides for timestamp parsing) as we assume the events to be sent in the expected format by the agents.

4.4.3 Event Analysis Module

The Event Analysis Module was the most demanding part in terms of implementation effort. It was developed in *Java* 1.8. It consists of aboud 8000 lines of code and 96 classes. It implements the query processing workflow model presented in section 3.4.3. This specified the four query execution phases and the main components involved in the process. Next, we go through each of these phases and its respective implementation details. To implement the interface with the *Elasticsearch*, we used the *Java* API (version 1.4.4) provided by the *Elasticsearch* platform.

Parsing (Phase 1)

The first is the parsing phase. This is performed by the Query Parser whose job is to receive a user-provided text query and produce a query tree. The query tree is the data structure that represents the query internally. Each node in the tree represents one specific DHIDS-QL syntactic connector. The

Query Parser was built using *Java Compiler Compiler* (a.k.a. *JavaCC*) 6.0 [JavaCC]. This is a tool that given a grammar specification, it converts it to a *Java* program capable of recognizing text and matching it to the grammar. Most of the Query Parser components were generated using this tool, given the DHIDS-QL language specification as presented in 3.5 (converted to the JavaCC input format). The language specification used to create the parser can be found in Appendix B. JavaCC was set to use the default LOOKAHEAD – 1, which means DHIDS-QL is an LL(1) grammar. Once the parsing phase is successfully terminated, then the query is considered syntactically correct and the execution can proceed to the next phase – validation.

Validation (Phase 2)

This phase provides an opportunity to traverse the query tree and perform the necessary checks or modifications in order to assure that every node is valid and to enforce any rules that cannot be verified at the syntactic level. If any inconsistency is detected, the Query Validator should either perform the necessary modifications to the query tree to make it valid, or if not possible, to fail-fast and raise the appropriate exception. This procedure assures that in the following phases, we are only dealing with valid queries. Since this implementation is a prototype, we didn't focus in performing extensive query checking. Nevertheless, we use the validation phase to perform field name translation and other adaptations in order to make our queries more readable and less verbose. For example, our prototype accepts queries written with simplified field names (e.g. "tcp") but it replaces them internally with the full name (e.g. "additionalData.packet.tcp").

Planning (Phase 3)

As explained in chapter 3 (section 3.4.3) each query is executed according to an execution plan specifically tailored to each query, determined before the execution phase. This is performed by the Query Plan Builder, which receives a validated query tree and produces a query execution plan. The latter is simply a data structure consisting of an ordered set of tasks to be carried out during the execution phase. Our system's model does not suggest the specific query execution operations that should be supported (just that there are three kinds of operations: fetch, filter and result-process). These are the query processing algorithms and we see them as an implementation matter. In our prototype, the specific implemented operations that may compose a query execution plan are:

1) Fetch Tasks:

i. ***Single Event Search*** – performs a search for occurrences of a single event, according to a given parameterization.

ii. ***Absent Event Search*** – performs a search for a given event that if not found returns a positive match.

iii. ***Kleene-Closure Search*** – performs a search for an unbounded number of events that share the same key characteristics.

2) Filter Tasks:

i. ***All-to-all Comparison Filter*** – can be used, after a Kleene-Closure Search, and filters the input set according to the evaluation of an all-to-all predicate.

ii. ***Pattern Filter*** – it filters instances of the pattern according to the evaluation of predicate expressed in the HAVING clause.

3) Result-process Tasks:

i. ***Publish Results*** – prepares the output event for each positive instance of the pattern, and indexes it in the Elastic Data Storage.

In our implementation, the query execution plan enforces a priority order on the tasks. That order corresponds precisely to the order in which they are presented here. That means Single Event Search tasks (if any) always happen before than Absent Event Search tasks (if there is any absent event), which precede Kleene-Closure Search, and so forth.

Execution (Phase 4)

The query execution is handled by the Query Runtime component. In concept, the Query Runtime iterates over the query execution plan and executes the tasks in sequence while feeding as input to each task the output of the previous one. Each task receives as input, and produces as output, a set of *cases*. A *case* is a partial event sequence representing a potential pattern match, i.e. a possibility that has not yet been discarded. For each task, the *cases* are divided in a few or many chunks (or subtasks) of configurable size and placed in an execution queue. A pool of threads consumes from this queue and executes the subtasks. The degree of concurrency is determined by the thread pool size also set by configuration.

For each *case,* the fetch-tasks request events from the Elastic Data Storage, via the Data Access Manager (Figure 3.5). We should note that the amount of processing needed to resolve a query depends greatly on the type of pattern we are querying, specifically the number of *cases* generated by each fetch task, that must be considered in the subsequent tasks. The idea is to put the heavy processing work on side of the Elastic Data Storage, and to simplify, as much as possible, the work on side of the Event Analysis Module. This is so, because the former is the component that can be horizontally scaled.

Elasticsearch is able to deal with many queries in the same request and efficiently parallelize the work internally. In our Event Analysis Module implementation, the queries for each chunk of *cases* are generated at the same time and sent to the *Elasticsearch* bulked in the same request. Therefore, the chunk size determines the number of queries per request demanded from *Elasticsearch*. To make full use of the *Elasticsearch* cluster's hardware, this number should be the highest possible just before, we start getting "rejected execution" exceptions from *Elasticsearch*. That means we are asking too many simultaneous queries from it. As we will see later in the next chapter, we found this optimal number to be around forty queries per request, for our test-bench server (a quad-core with 8GB RAM), as we will present in chapter 5.

5 Evaluation

To validate the proposed DHIDS platform we deployed an experimental environment to conduct a set of tests, using two distinct test-bench installations. We start by presenting each of the mentioned test-bench environments in section 5.1. The remaining sections are dedicated to the presentation of a selected set of experiments and representative observations, using those test-bench pilots. Section 5.2 addresses network performance indicators and end-to-end latency conditions related to the DHIDS platform and its components. The section also presents some base validation observations on reliability, efficiency and scalability criteria, considering event-detection, -formatting, -dissemination and -storage. Section 5.3 addresses DHIDS-QL expressiveness and effectiveness criteria, by showing how the proposed query-language can be used to express a set of selected representative attack-patterns (or signatures), queried as searchable evidences of combinations and correlations of stored events. Finally, in section 5.4, we present a summary of the evaluation phase and some additional remarks.

5.1 Evaluation Environment

For the evaluation and experimental observation of the proposed, designed and implemented DHIDS platform, we created two test environments: Test-bench 1 and Test-bench 2. Most of the experimental tests were conducted in the Test-bench 1, designed to operate as a dedicated and isolated network, described in 5.1.1. Some observations also used the Test-bench 2 described in 5.1.2, a setup environment integrated in the FCT/UNL internetwork infrastructure, in an effort to more closely simulate a real-world scenario in terms of latency and network conditions.

5.1.1 Experimental Test-bench 1

In the Test-bench 1, the DHIDS platform is supported by an isolated switched IEEE 802.3 and IEEE 802.11g LAN environment, as represented in the Figure 5.1. Table 5.1 complements the diagram and characterizes each component presented in the figure, regarding its function, software and hardware.

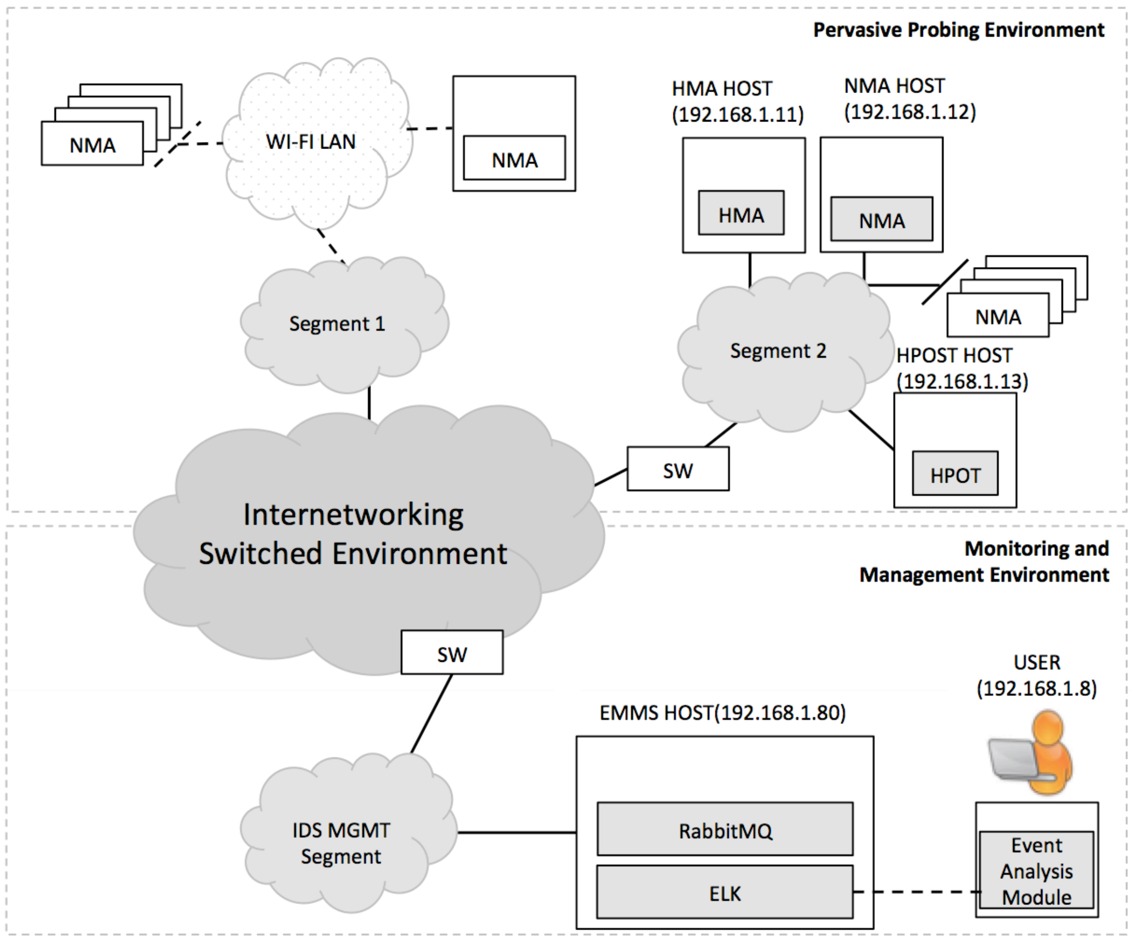


Figure 5.1: Test-bench 1 network diagram

The pervasive probing environment (for NMA, HMA and HPOT components) correspond to a set of devices installed in two switched segments, namely, the wireless IEEE 802.11g (54 Mbps) *Segment 1*, and the wired IEEE 802.3 (100 Mbps) *Segment 2*. The IDS MGMT segment corresponds to the EMMS network and is a wired IEEE 802.3 (100 Mbps) segment. This EMMS instance is composed of just one node (EMMS HOST). Specifically, it hosts a single instance of *Elasticsearch* and the related *ELK* software stack, which includes the *Logstash* and *Kibana*. In this test-bench, this same node simultaneously hosts the Event Dissemination Platform implemented by *RabbitMQ*.

Table 5.1: Components in the Test-bench 1 installation

ID	Description	Hardware	Software
EMMS HOST	Host for both the Event Dissemination Platform instance and the Event and Monitoring and Management System.	AMD Quad-core; 8GB DDR3; 100-BASE-T Ethernet;	Linux Ubuntu 14.04 64bit; JRE 1.8; Elasticsearch 1.7.1; Logstash 1.5.5; Kibana 4.2; RabbitMQ Server 3.4.4;
NMA / NMA HOST	The Network Monitoring Agents. These can either be integrated in an node as a background software, or as an specific independent network monitoring device	RaspberryPi 1B / RaspberryPi 2B	Raspbian Wheezy 3.18; Snort (v xxx); Suricata 2.0.8;
HMA HOST	The Host Monitoring Agent.	RaspberryPi 1B	Raspbian Wheezy 3.18; Apache 2.2.22; Tripwire 2.4.2.2;
SW	Switches 10/100 BASE-T	Micronet SP616EA Ether-Fast 16 ports / SMC 108DT 8 ports	Not-Applicable
HPOT HOST	The honeypot implemented by Wackopicko running on a LAMP server in a RaspberryPi. It is used as the target of attacks.	RaspberryPi 1B	Raspbian Wheezy 3.18; Apache 2.2.22; MySQL 5.5; PHP 5.5; Wackopicko;
USER	User station. Runs the implemented Event Analysis Module. In some tests, it is also used as the attacker station.	MacBook Pro; 2.5GHz Intel Core i5 (dual core); 8GB DDR3	OS X 10.10.5; VirtualBox 4.3.26; Ubuntu 14.04 LTS (virtualized); w3af 1.7.6

The USER station usually connects wirelessly through IEEE 802.11g to the WI-FI LAN. It runs the Event Analysis Module, which in turn is indirectly connected to the *ELK* as it consumes and writes data from and to *Elasticsearch*. Some specific tests may require to not be limited by the IEEE 802.11g wireless network bandwidth in which case we connect the USER node to *Segment 2* via Ethernet cable. In other tests, we also use the USER station to inject the attacks.

The local internetworking and switching environment (represented in figure 5.1) is provided by the interconnection of two local 100 Mbps switches. Additionally, to allow for packet-sniffing functions, a local 10Mbps hub-repeater is available, which may be connected whenever necessary, to each switch, allowing the connection of network-interfaces in promiscuous modes, to inspect traffic observed in the hub ports and in one specific port of each switch.

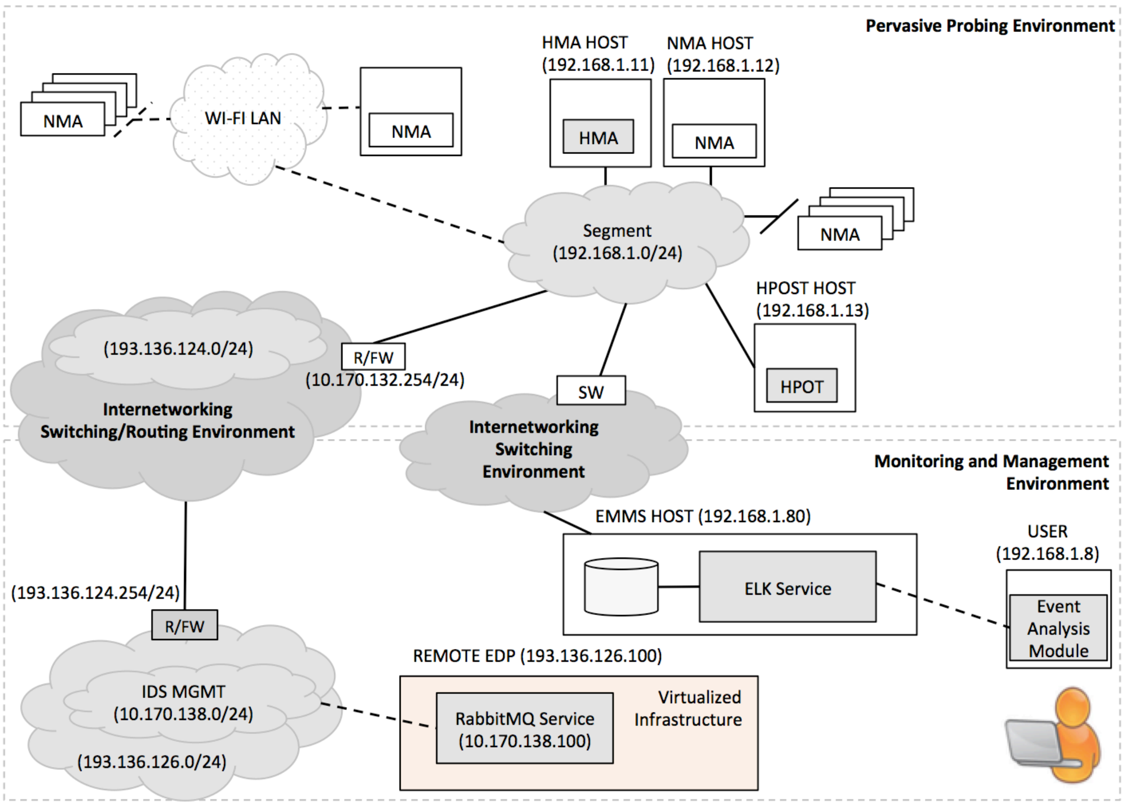
5.1.2 Experimental Test-bench 2

Test-bench 2 is useful for complementary evaluations of quantitative metrics, closer to a real-life environment. This environment is integrated to the internetworking environment and infrastructure of FCT/UNL. Namely, a network-environment interconnecting DI (*Departamento de Informática*) and local teaching and research labs, and remote Virtual LANs installed in the SI FCT/UNL (Serviço de Informática – FCT/UNL), a centralized Sector responsible for all IT installations, as well as system and network administration functions, including the centralized software services provisioning.

The Test-bench 2 setup allows for the observation of differences in testing metrics, some of them primarily obtained in the test-bench 1, but in this case closer to a "real-operation setup environment". For example, we can compare the event-publishing throughput performance in the context of the test-bench 2 and its central management solution, to compare with the equivalent observation in the dedicated local setup of Test-bench 1.

This internetworking environment is represented in the Figure 5.2. The pervasive probing environment (for NMA, HMA and HPOT agents) is enabled by devices installed in a LAN segment. This is located in a local lab and connects to the wired LAN switched infrastructure covering the DI-FCT-UNL through a NAT box (R/FW) connected to wall Ethernet connector. In this case, we use a local switched segment, providing a IEEE 802.3 wired switched access. For the Test-bench 2 local DI-FCT-UNL segments and local addressing, we use the DHCP and DNS services, as centrally managed by the SI-FCT-UNL. The Event Dissemination Platform in this case is located in a remote virtual machine installed in a VLAN at the SI-FCT-UNL, running the *RabbitMQ* platform (REMOTE EDP). The EMMS, similarly to Test-bench 1, runs in the EMMS HOST in our local network. This configuration requires that every published event traverse the FCT-UNL network infrastructure and return to the EMMS HOST, therefore being subjected to the traffic conditions offered by the FCT-UNL network.

We should notice that in the Test-bench 2, we are not able to install pervasive intrusion-detection sensors in the internetworking infrastructure. We can only simulate their operation, by processing and disseminating datasets of pre-captured events, for example, reference datasets with events reflecting attack-patterns. First, we are not authorized to install systems or devices with network interfaces running in promiscuous mode to capture or to sniff traffic though the FCT/UNL network infrastructure. Second, we can't use Honeypot or HIDS elements running in production servers. Third, we can't capture and we are not authorized to publish "real intrusion events" that have been observed in the FCT/UNL infrastructure, through the monitoring, interception or inspection of switch-based traffic or routing-processing traffic crossing the production infrastructure.



Pervasive Probing Environment
HMA HOST (192.168.1.11)
NMA HOST (192.168.1.12)
NMA
WI-FI LAN
NMA
HMA
NMA
Segment (192.168.1.0/24)
NMA
HPOST HOST (192.168.1.13)
HPOT
(193.136.124.0/24)
R/FW
(10.170.132.254/24)
Internetworking Switching/Routing Environment
SW
Internetworking Switching Environment
Monitoring and Management Environment
EMMS HOST (192.168.1.80)
ELK Service
USER (192.168.1.8)
Event Analysis Module
(193.136.124.254/24)
R/FW
REMOTE EDP (193.136.126.100)
IDS MGMT (10.170.138.0/24)
(193.136.126.0/24)
RabbitMQ Service (10.170.138.100)
Virtualized Infrastructure

Figure 5.2: Test-bench 2 network diagram

The characterization of components in the test-bench 2, regarding its function, software and hardware type, is presented in Table 5.2.

Table 5.2: Components in the Test-bench 2 installation

ID	Description	Hardware	Software
REMOTE EDP	Remote host running RabbitMQ. Virtualized in SI-FCT-UNL infra-structure.	Intel Xeon E5-2660 2.2GHz; 506MB DDR;	Linux Ubuntu 14.04 64bit; JRE (Java Runtime Environment) 1.8; RabbitMQ Server 3.4.4
EMMS HOST	Host for the Event and Monitoring and Management System. Runs the ELK service stack.	AMD Quad-core; 8GB DDR3; 100-BASE-T Ethernet;	Linux Ubuntu 14.04 64bit; JRE 1.8; Elasticsearch 1.7.1; Logstash 1.5.5; Kibana 4.2;
NMA/ NMA HOST	The Network Monitoring Agents. These can either be integrated in an node as a background software, or as an specific independent network monitoring device	RaspberryPi 1B / RaspberryPi 2B	Raspbian Wheezy 3.18; Snort (v xxx); Suricata 2.0.8;
HMA HOST	The Host Monitoring Agent.	RaspberryPi 1B	Raspbian Wheezy 3.18; Apache 2.2.22; Tripwire 2.4.2.2;
SW	Switches 10/100 BASE-T	Micronet SP616EA EtherFast 16 ports / SMC 108DT 8 ports	Not applicable.
R/FW	Router / Firewall	Linksys WRT54GC	Not applicable.
HPOT HOST	The honeypot implemented by Wackopicko running on a LAMP server in a RaspberryPi. It is used as the target of attacks.	RaspberryPi 1B	Raspbian Wheezy 3.18; Apache 2.2.22; MySQL 5.5; PHP 5.5; Wackopicko;

5.2 Event Collection and Processing Tests

The goal for this first series of tests is to evaluate how reliable, efficient and scalable is the process of capturing, formatting, disseminating and storage of events. Specifically, we aim to answer the following questions:

- Are the events detected by the agents and reliably transmitted to the Event Dissemination Platform? What is the effective publishing throughput?
- What is the expectable performance and local processing capabilities of the specific probing devices (namely the Raspberry Pi nodes)?
- Does the implemented EMMS correctly process and store the events? What is the event-processing throughput?

5.2.1 Network Performance

We began by measuring the effective network performance on both test-benches. This measure will serve as an upper-bound benchmark to be compared against the following performance observations. For that, we used *iPerf 2.0.5* [iPerf] – a network performance active measuring tool – to obtain the measurement of the maximum achievable bandwidth in a TCP connection from the agents to the EMMS HOST, and to the EDP REMOTE in the case of Test-bench 2. The measurements considered in both cases are the ones seen by the receiver. Each of the following results is an average of ten consecutive observations. The variables in this tests are the test-benches: both 1 and 2; and the device running the agent: *RaspberryPi 1 Model B*, *RaspberryPi 2 Model B* and MacBook Pro (associated to the USER station – see Table 5.1).

Table 5.3: *iPerf* bandwidth measurements

	RaspberryPi 1B	*RaspberryPi 2B*	*MacBook Pro*
Test-bench 1	57,2 Mbps	94,0 Mbps	94,0 Mbps
Test-bench 2	46,2 Mbps	46,1 Mbps	46,4 Mbps

Table 5.3 presents the obtained measurements for both test-benches and all the available agent hardware devices. As expected, *RaspberryPi 2B* is much faster than its predecessor dealing with network traffic, as the results from Test-bench 1 show. Regarding *test-bench 2*, the bottleneck seems to be imposed by the

network conditions, every device showed similar bandwidth measurements. In addition, we measured the network's end-to-end latency, in the same circumstances as the bandwidth measurements (Table 5.4). The presented results are the average of one hundred measurments.

Table 5.4: Round-trip-time measurements

	RaspberryPi 1B	*RaspberryPi 2B*	*MacBook Pro*
Test-bench 1	0,903 ms	0,589 ms	0,504 ms
Test-bench 2	2,987 ms	2,826 ms	2,737 ms

5.2.2 End-to-end Event Processing Throughput

The goal for this test is to evaluate the overall system's event-processing capacity. For this purpose, we used an event log of measurable size in *PCAP* format from the DARPA data set – named "outside.tcpdump" – (see Appendix C for more details about the DARPA data set) containing 233428 events. We set up one Generic Agent to extract, format and publish events into the Event Dissemination Platform implemented by *RabbitMQ*. We set up the EMMS on the other side, implemented by *Logstash* and *Elasticsearch*, to consume, process and store the events. The throughput is calculated based on the difference between the receiving time of the first and last events of the dataset. As mentioned in last chapter, *RabbitMQ* offers an acknowledge-based reliable publishing mode. We tested both publishing conditions: the default (without Confirms) and with reliable publishing (Confirms). Event persistency was not enforced in *RabbitMQ*. This means that messages were not necessarily written to disk when they reached the queue.

There are three variables to consider in this test:

- Event-publishing mode: reliable (using the Confirms mechanism) vs. unreliable (default);
- Agent host device: *RaspberryPi 1B*, *RaspberryPi 2B* and a laptop (*MacBook Pro*).
- Test-benches (both 1 and 2)

The presented results (Figure 5.3) are the aggregated values of twenty independent observations for each test instance in Test-bench 1. Regarding the previously obtained network performance measurements, it seems reasonable to expect a very significant decline on the effective throughput. This is due to the overheads imposed by (1) the *RabbitMQ*, an effect inflated by the fact that the events are being published one-by-one and, (2) the event processing and indexing operation on the EMMS side (*Logstash* and *Elasticsearch*). The *RaspberryPi*'s hardware limitations should also likely contribute to the deterioration of the event-publishing performance.

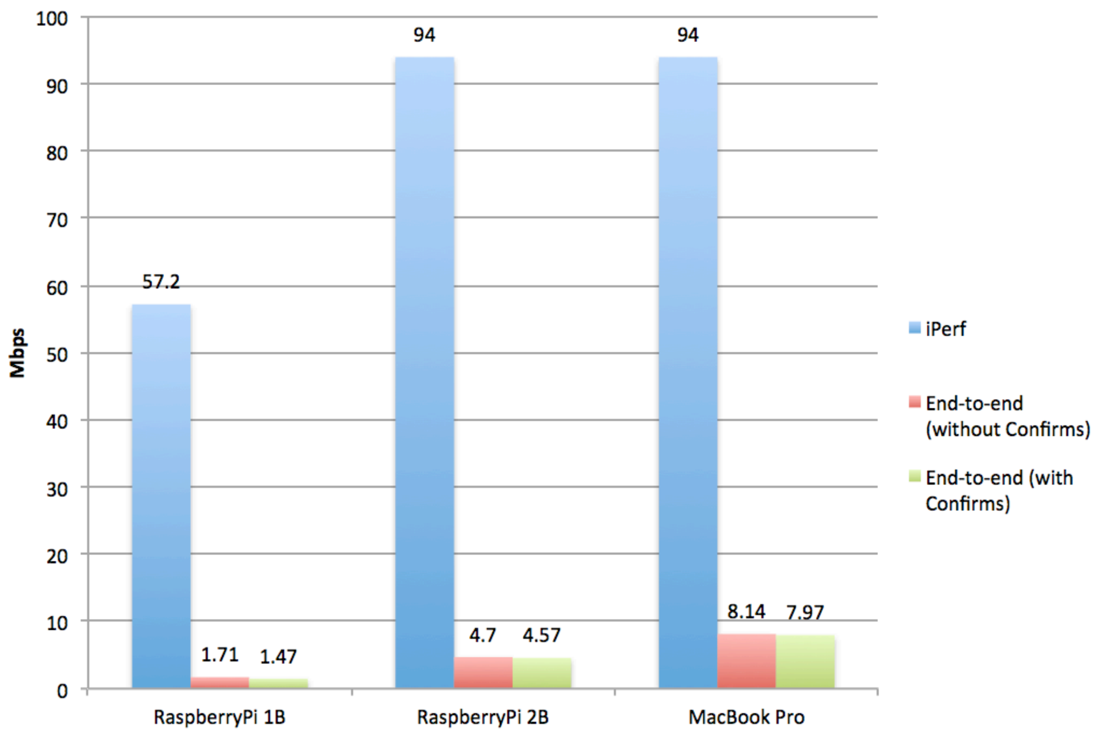


Figure 5.3: End-to-end event-processing throughput (Testbench 1)

As expected, *RaspberryPi 1B* was the slowest tested hardware option, achieving an effective publishing throughput of around 1,5 Mbps. That corresponds to an observed average of 250 to 300 events per second, using the mentioned data set. This was outperformed by far by its successor, which achieved a throughput of up to 4,7Mbps. That equates to about 800 events per second (the event average size in this particular data set is 767 bytes).

We didn't find a very significant variation in performance when using *RabbitMQ*'s Confirms mechanism. It's also worth noting the fact that there was no event loss in these test conditions, even when not using reliable publishing.

Next, we present the results obtained in Test-bench 2 for the exact same tests (Figure 5.4). As we can recall from the previously presented network performance tests (5.2.1), the achievable bandwidth measured for this network conditions was significantly less than the observed in Test-bench 1. That fact seems to have had no impact in the observed end-to-end event-processing throughput. Moreover, we even noticed a slight positive variation in performance (more than 10%) in one of the tests – the one using the laptop. The most likely explanation for that is that, in this test-bench, both *Elasticsearch* and *Logtash* were not running alongside with *RabbitMQ*. The latter was now running in the virtual machine, freeing some resources for event processing on the EMMS side. The following tests will probably reveal more details on this issue.

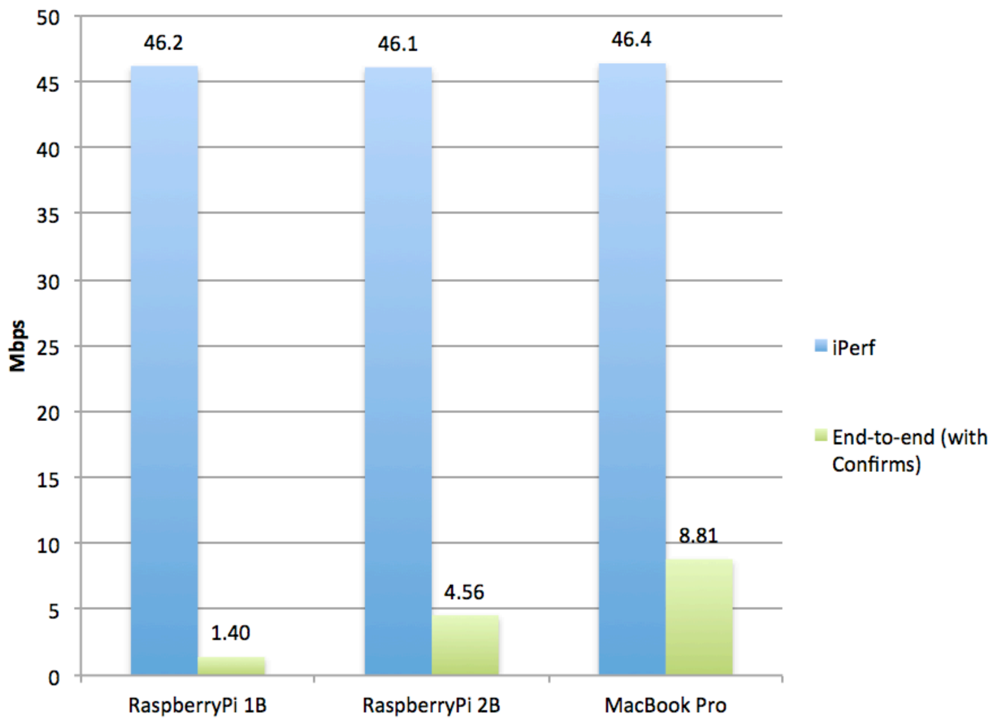


Figure 5.4: End-to-end event-processing throughput (Test-bench 2)

As anticipated, the "end-to-end" throughput test revealed a very significant decrease in performance when compared with the available bandwidth. However, it didn't reveal which component or components are the limiting factors. This is an important question because it has a direct reflection in the system scalability. The results from the following test will certainly shed more light on this matter.

5.2.3 *RabbitMQ* Event Publish-Subscribe Throughput

In this test, we evaluate the performance of the Event Dissemination Platform implemented by *RabbitMQ* in isolation. The goal is to estimate what is the actual overhead imposed by this component and consequently, to what extent does it affect the event-processing throughput. For that purpose, we set up the same Generic Agent, publishing events in same conditions as the previous test. On the server side, instead of the EMMS, we set up a simple process just consuming events and measuring the receiving rate. The events were published in a reliable channel, using the Confirms mechanism.

The variables in this test are the following:

- Agent host device: *RaspberryPi 1B*, *RaspberryPi 2B*, *MacBook Pro*.
- Test-benches (both 1 and 2)

The presented results represent the aggregated measurements from twenty independent observations in both Test-bench 1 (Figure 5.5) and Test-bench 2 (Figure 5.6).

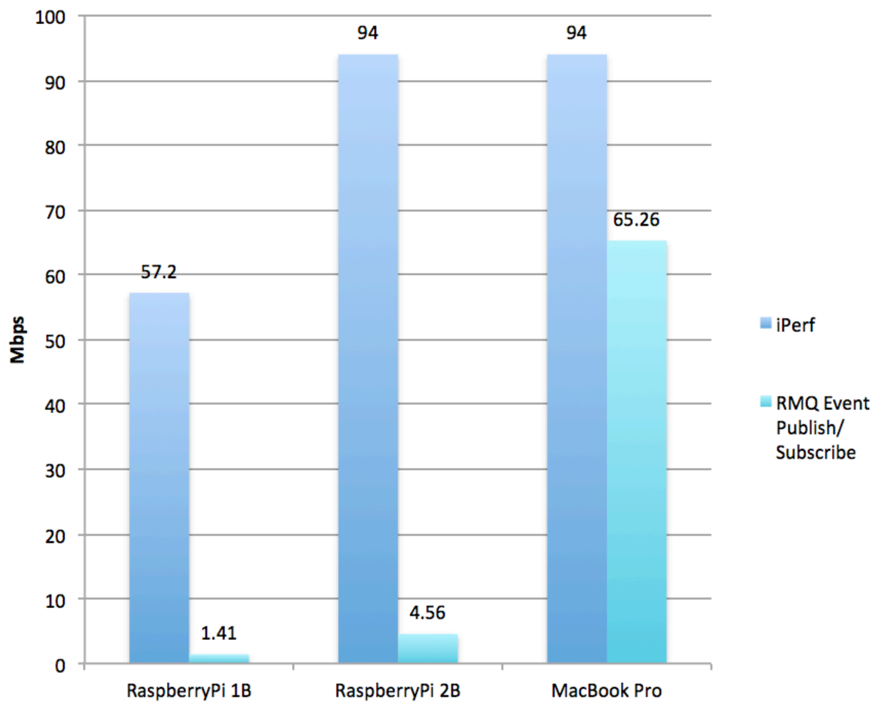


Figure 5.5: *RabbitMQ* publish-subscribe throughput (Test-bench 1)

Considering both the results from this test and the previous one, there seem to be two distinct bottlenecks. The first bottleneck concerns the *RaspberryPi* devices. The effective throughput in Mbps obtained in this test - simple publish and subscribe of events without any further processing – is, almost exactly, the same as in the first "end-to-end" test. It seems reasonable to assume that the limitation is actually in the publishing capacity of both devices, imposed by their hardware limitations, which in turn sets a ceiling for the event-processing rate for the whole system. This theory is reinforced when we look at the results for the laptop agent, where that limitation seems to not be there anymore.

The second bottleneck becomes clear when we compare the results obtained on the laptop on both tests. When the events were being consumed by *Logstash* and indexed in *Elasticsearch*, we achieved a throughput of about 8Mbps. With those two components out of the equation, we observe a throughput increase to around 65 Mbps. This leads us to believe that the 8 Mbps seen before is in fact the upper bound of *Logstash*/*Elasticsearch* event-processing throughput in this specific setup, in which these components are operating in a single node. Although we don't have at this moment, data regarding the performance of *Elasticsearch* and *Logstash* in a distributed environment due to limitations in time and resources, we don't believe this bottleneck to be a serious limitation. As mentioned, *Elasticsearch* is built to scale horizontally according to the needs, simply by adding new nodes to the cluster. *Logstash* is a single process pipeline and not intrinsically scalable, but multiple instances could be set up in multiple nodes. In that scenario, *RabbitMQ* would be configured as a "work queue" to distribute the events among the various *Logstash* instances, for example following a "round-robin" policy. This way, both *Logstash* and *Elasticsearch* could be easily scaled. *RabbitMQ* is also built to operate in clustered environments. The next section presents a similar experiment with *RabbitMQ* in a distributed environment.

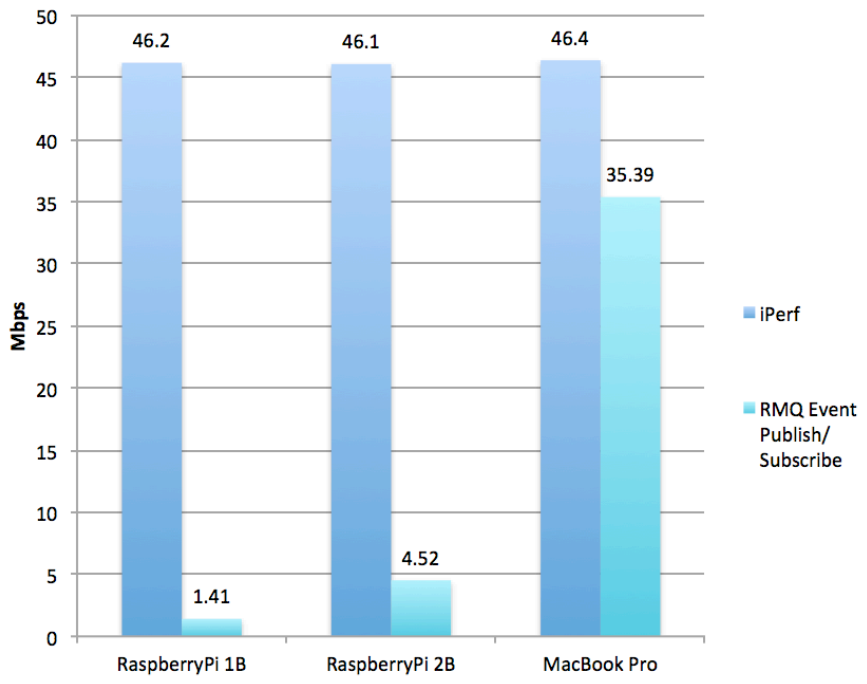


Figure 5.6: *RabbitMQ* publish-subscribe throughput (Test-bench 2)

5.2.4 *RabbitMQ* Throughput in Scalable Conditions

In this test, we evaluated the performance and scale capability of the Event Dissemination Platform implemented by *RabbitMQ*. This was conducted in a different test-bench (neither of the previously presented ones), as a possibility offered by PT Portugal SGPS, S.A. especially for this test. This test-bench consists of a clustered *RabbitMQ* pre-production environment in PT's infrastructure.

This test features multiple Generic Agent processes running in multiple hosts (simulating multiplicity of agents), a decent size *RabbitMQ* cluster and multiple event consumer processes running in another (single) machine. On the agents' side there were 24 Generic Agent processes running simultaneously, evenly distributed by four machines (6 agents per host). The *RabbitMQ* cluster was composed of six machines interconnected by a 1 Gbps backbone. The hardware present in each machine (both in the agent's and in the *RabbitMQ* cluster) was the following: 8 cores *AMD Opteron 6000* and 128 GB RAM DDR3. The consumer was running in another more powerful host with the following hardware characteristics: 12 cores *AMD Opteron 6000* and 256 GB RAM DDR3. This cluster (including the agents' and consumer's machines) is interconnected via a 1 Gbps end-to-end backbone.

For this test, the 24 agent processes were executed simultaneously, all publishing events through the same queue. Similarly to the previous *RabbitMQ* test, this was also performed using reliable publishing (using the Confirms mechanism) and the test was composed of twenty repetitions, each followed by a 10 seconds interval to allow for network stabilization. For simplicity, we considered the results of just one agent, since we have found no relevant differences between the various agents.

The average throughput achieved by one agent was 194.4 Mbps. Since there were 24 agents executing simultaneously the aggregated throughput is calculated to be 4.56 Gbps. This equates to an average rate of about 771 thousand events per second that can be dispatched by the Event Dissemination Platform regarding the described circumstances and equipment.[5]

[5] The event dataset used was the same used in the previous tests, whose average size per event is 767 bytes after encapsulation in the interoperability format.

This test demonstrated that the Event Dissemination Platform wouldn't pose a problem regarding the overall scalability of our system. PT's technicians also provided further information about the *RabbitMQ* cluster's typical performance that supports this conclusion. Their pre-production environment typically achieves between 160 and 400 Mbps per client in similar tests (depending on the number of clients running simultaneously), consistent with what we observed in this particular test. Their more powerful production infrastructure, interconnected by 10Gbps backbone and using superior hardware, they say one publishing client can achieve in average 1,2Gbps throughput.

5.3 DHIDS-QL Expressiveness and Effectiveness

The goal for this series of tests is to answer the following questions:

- Does the Event Analysis Module provide the querying capabilities specified in the system model?
- How efficient is the runtime support to process queries, in face of scale conditions and heterogeneity of events stored in the EMMS?
- Regarding specific "real-life" attack scenarios, how effective is the detection, and how appropriate is the language expressiveness to evidences of relevant and recent security exploits? Namely: SQL-injection, SYN-flood DoS and SSL Heartbleed attack?
- Is it expectable that the designed solution presents extensibility support to evolve and to allow the expression of new attacks?

To evaluate the expressiveness of the proposed DHIDS-QL language for auditing specific attacks and the implemented system's detection effectiveness in the face of realistic and some of the security threats most prominent today, we selected the following auditing cases:

- SQL-Injection Attack
- SYN-Flood DoS Attack
- SSL Heartbleed Attack

For each attack scenario, we recreated the attacks in our test environment using specific attack tools and target applications. Then we collected the traces left by the attack in the form of events, which were eventually stored in the EMMS. Finally, we looked for the telltale signs of the attack by formulating specific DHIDS-QL queries. All the following tests were performed in *Test-bench 1*.

5.3.1 SQL-Injection Attack

According to OWASP's *"The Ten Most Critical Web Applications Security Risks"* report [OWASP13], injection vulnerabilities are considered the most common type of security weaknesses in web applications today. These can occur when non-sanitized user input data is passed to an interpreter as part of a command or query. This way, an attacker can trick the interpreter to execute his malicious instructions. Here we focus specifically on SQL-Injection, likely the most popular kind of injection attack, to recreate an attack scenario where a vulnerable application is targeted in a monitored environment. SQLi is, as the name suggests, an injection technique that attempts to execute malicious SQL statements on a database, taking advantage of a user entry field where the input is not appropriately sanitized.

For this test scenario, we set up a vulnerable web application, *Wackopicko* [Wackopicko], running on a LAMP server on a HPOT (running on a *RaspberryPi 1B* node), as represented in the figure 5.1. Logs from the *Apache* server and the *MySQL* database (as components of such "Honeypot") were recorded and published as events on the EMMS server. We used *w3af* [w3af] (running on the USER station) – a vulnerability scanning and exploiting tool for web applications. With *w3af*, we perform an SQLi scanning and intrusion attack targeting *Wackopico*. Additionally, logs of network events were also collected in *PCAP* format by a *Suricata* instance (used here in the role of a NMA). *Suricata* also runs on the HPOT node, the target of the attack (but it could run on any other node in the same segment). The events were published by the probing agents involved and stored on the EMMS system.

This test intends to illustrate a typical use case scenario of auditing a potential injection attack, in this case SQLi, in combination with possible related evidences captured by intrusion-detection agents. Even that we studied in this case a SQLi attack, the underlying principles and methodology could be extrapolated to other kinds of web injection attacks such as cross-site scripting - XSS). This test should provide a concrete idea of how our system performs when dealing with that kind of attack detection.

One of the telltale signs of a consummated SQLi attack can be found on the database log. We know, from the study of *w3af*, that its SQL injection plugin tries to inject the string *a'b"c'd"* in every injection point (in HTML forms) and searches for SQL errors in the HTTP response body. Assuming the attack was successful, we began by looking for that kind of string pattern occurring at the database level. We did this by executing the following query (Listing 5.1) on the Event Analysis Module.

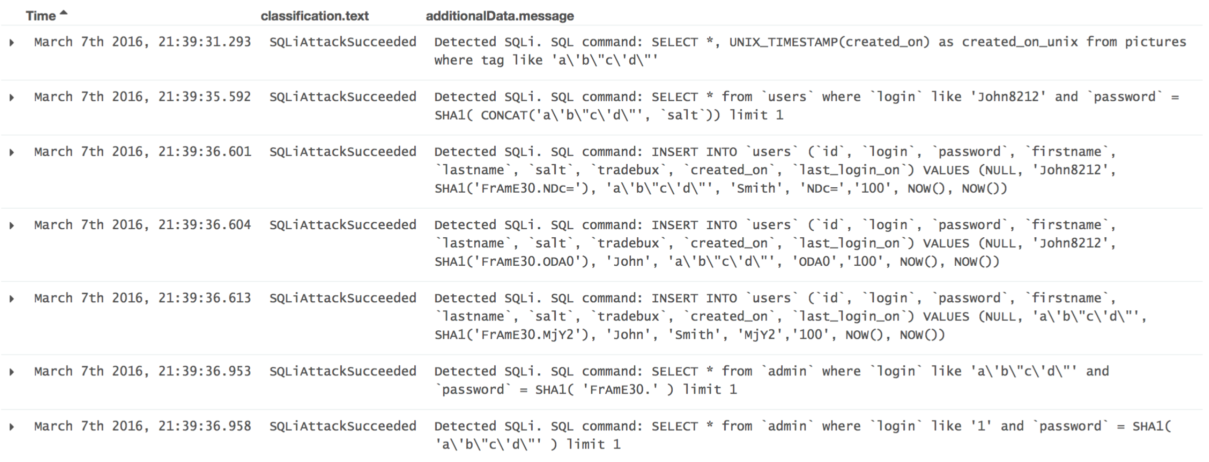
```
PATTERN SEQ (MYSQL e1)
WHERE e1.command = "Query"
AND e1.argument CONTAINS "a\\'b\\\"c\\'d\\\""
PUBLISH "SQLiAttackSucceeded" IN "attacks"
DESCRIPTION "Detected SQLi. SQL command:" + e1.argument;
```

Listing 5.1: *MySQL* event analysis (DHIDS-QL)

We must notice that the string we are in fact looking for is not the originally injected string but its equivalent considering the escape characters as it will appear in the *MySQL* log: *a\'b\"c\'d\"*. The string presented in the query is the equivalent to this last one considering the escape characters for our parser, which is implemented in Java[6].

The query in the listing 5.1 returned seven matches. These can be seen on Figure 5.7, which shows a partial screen-shot from *Kibana* (running on the EMMS system) containing every document in the recently created "attacks" index. As mentioned earlier, *Kibana* is a powerful data navigation and visualization tool integrated with *Elasticsearch*. In the following tests, we use it to observe the output events created by the queries. We will be focusing on three event fields: *Time,* which represents the detection timestamp; *classification.text,* which means the event "type" defined by the query; and *additionalData.message,* which is an event description also specified by the query.

[6] The quotation marks (") and apostrophes (') are escape characters in *MySQL,* so they are represented in string preceded by the backslash (\) as in \" and \'. Similarly, in *Java,* the backslash (\) and the quotation marks (") are escape characters themselves and are also represented in a string preceded by backslash (\) as in \\ and \".



```
Time ^                         classification.text    additionalData.message
▸ March 7th 2016, 21:39:31.293  SQLiAttackSucceeded  Detected SQLi. SQL command: SELECT *, UNIX_TIMESTAMP(created_on) as created_on_unix from pictures
                                                     where tag like 'a\'b\"c\'d\"'
▸ March 7th 2016, 21:39:35.592  SQLiAttackSucceeded  Detected SQLi. SQL command: SELECT * from `users` where `login` like 'John8212' and `password` =
                                                     SHA1( CONCAT('a\'b\"c\'d\"', `salt`)) limit 1
▸ March 7th 2016, 21:39:36.601  SQLiAttackSucceeded  Detected SQLi. SQL command: INSERT INTO `users` (`id`, `login`, `password`, `firstname`,
                                                     `lastname`, `salt`, `tradebux`, `created_on`, `last_login_on`) VALUES (NULL, 'John8212',
                                                     SHA1('FrAmE30.NDc='), 'a\'b\"c\'d\"', 'Smith', 'NDc=','100', NOW(), NOW())
▸ March 7th 2016, 21:39:36.604  SQLiAttackSucceeded  Detected SQLi. SQL command: INSERT INTO `users` (`id`, `login`, `password`, `firstname`,
                                                     `lastname`, `salt`, `tradebux`, `created_on`, `last_login_on`) VALUES (NULL, 'John8212',
                                                     SHA1('FrAmE30.ODA0'), 'John', 'a\'b\"c\'d\"', 'ODA0','100', NOW(), NOW())
▸ March 7th 2016, 21:39:36.613  SQLiAttackSucceeded  Detected SQLi. SQL command: INSERT INTO `users` (`id`, `login`, `password`, `firstname`,
                                                     `lastname`, `salt`, `tradebux`, `created_on`, `last_login_on`) VALUES (NULL, 'a\'b\"c\'d\"',
                                                     SHA1('FrAmE30.MjY2'), 'John', 'Smith', 'MjY2','100', NOW(), NOW())
▸ March 7th 2016, 21:39:36.953  SQLiAttackSucceeded  Detected SQLi. SQL command: SELECT * from `admin` where `login` like 'a\'b\"c\'d\"' and
                                                     `password` = SHA1( 'FrAmE30.' ) limit 1
▸ March 7th 2016, 21:39:36.958  SQLiAttackSucceeded  Detected SQLi. SQL command: SELECT * from `admin` where `login` like '1' and `password` = SHA1(
                                                     'a\'b\"c\'d\"' ) limit 1
```

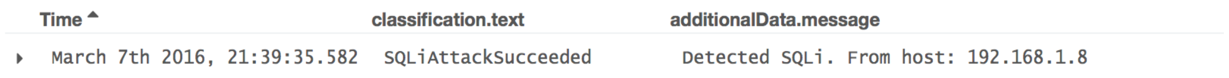
Figure 5.7: Results of *MySQL* event analysis query (*Kibana*)

The last query still doesn't reveal where the attack came from. For that purpose, we can now try to correlate the MySQL events (which reveal that there was an attack) with the preceding POST request found in the Apache log (which reveal the origin of the HTTP requests that most likely carried the attack). We do so with the following query (Listing 5.2):

```
PATTERN SEQ (APACHE e1, MYSQL e2)
WHERE e1.verb = "POST"
AND e1.request = "/users/login.php"
AND e1.response = 200
AND e2.command = "Query"
AND e2.argument CONTAINS "SELECT*users*a\\'b\\\"c\\'d\\\""
WITHIN 30ms
PUBLISH "SQLiAttackSucceeded" IN "attacks2"
DESCRIPTION "Detected SQLi. From host: " + e1.clientip;
```

Listing 5.2: Correlating *MySQL* with *Apache* events (DHIDS-QL)

The previous query (Listing 5.1) returned several matches. One of them (second row in Figure 5.7) is related to an SQLi attempt on the users' login form. This query (Listing 5.2) targets specifically that occurrence. It correlates both the *MySQL* event and the related HTTP POST request, by relating the request URI (/users/login.php) with the manipulated SQL query that should validate the login (SELECT … FROM users … <SQLi string>). As expected, it returned exactly one match (shown in Figure 5.8). This query, as opposed to the previous one, not only tells us that there was a consummated SQLi attack but also reveals the origin of the potential malicious request.



Time	classification.text	additionalData.message
March 7th 2016, 21:39:35.582	SQLiAttackSucceeded	Detected SQLi. From host: 192.168.1.8

Figure 5.8: Results of the correlation between *MySQL* and *Apache* events (*Kibana*)

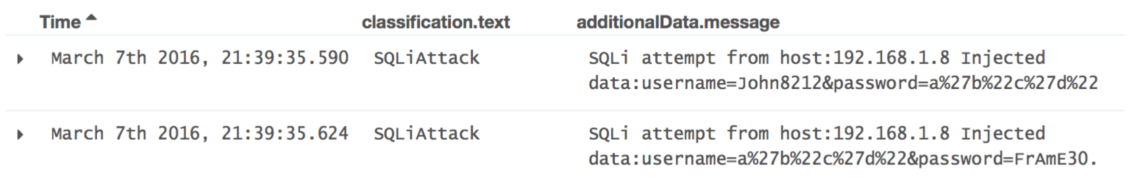
Another variant of this scenario would be if we wanted to know if there was an SQLi attempt regardless of whether it was successful or not. In that case, it's not enough to look only at the database and *Apache* events. In fact, *Apache* by default doesn't log the data in POST requests. Therefore, in this case we had to look also at the network events. Specifically, we looked at the HTTP messages captured on the network (by the *Suricata* based agent). We did so with the following query (Listing 5.3).

```
PATTERN SEQ (HTTP pkt)
WHERE pkt.http.request.method = "POST"
AND pkt.http.request.uri = "/users/login.php"
AND pkt.payload.data CONTAINS "a%27b%22c%27d%22"
PUBLISH "SQLiAttack" IN "attacks"
DESCRIPTION "SQLi attempt from host:"+pkt.ip4.source
+ " Injected data:" + pkt.payload.data;
```

Listing 5.3: Looking for SQLi evidences in HTTP messages

We should notice that the string pattern we are looking for in this query is the same (*a'b"c'd"*), only this time it is its HTML-encoding equivalent.

This query looks for HTTP POST requests directed to *Wackopicko*'s login page, carrying the typical *w3af*'s SQLi pattern. It produced two output events as shown in Figure 5.9. They are two variants of the same attack injected by w3af. One targets the "username" field and other targets the "password" field.



Time	classification.text	additionalData.message
March 7th 2016, 21:39:35.590	SQLiAttack	SQLi attempt from host:192.168.1.8 Injected data:username=John8212&password=a%27b%22c%27d%22
March 7th 2016, 21:39:35.624	SQLiAttack	SQLi attempt from host:192.168.1.8 Injected data:username=a%27b%22c%27d%22&password=FrAmE30.

Figure 5.9: Results of the SQLi evidences found in HTTP messages (*Kibana*)

5.3.2 SYN-Flood DoS and DDoS Attacks

The SYN-Flood is a form of denial-of-service in which the target is hit with a barrage of TCP SYN requests, in an attempt to exhaust its resources and impair its ability to answer legitimate requests. This happens because the receiver to initiate a connection has to store some state while it waits for the expected acknowledgement. When that acknowledgement doesn't arrive, those resources are only liberated much later, after a time-out. This form of DoS is considered outdated nowadays, as it can be defeated by some mechanisms like the use of SYN-cookies, which prevent the receiver to store any state before the three-way handshake is completed. Nevertheless, many systems don't implement such new features, and the principle underlying the SYN-flood attack maintains its relevance today and it can still be effective. The danger is particularly relevant in the context of DDoS (Distributed Denial of Service) attacks. In addition, it is a very illustrative example of a DoS attack evidence, and an interesting example to demonstrate the capabilities of the pattern-detection algorithm and language.

For this test scenario, first we simulated a SYN flood attack and captured its evidences in the form of related events. For that purpose, we set up an Apache server, in the HPOT, and used *hping3* [hping] – a command-line oriented TCP/IP packet assembler and analyzer tool – to send a profusion of SYN packets to the HPOT from the USER station, using random fake source IP addresses. Meanwhile, we had a NMA (implemented by a *Suricata* instance running in a *RaspberryPi*) sniffing packets on the same switch port as the HPOT (using a hub/repeater). The packets were then sent in the form of events to the EMMS, as usual, using the Generic Agent implementing the NMA.

Using *hping3* (USER station), we sent 10.000 SYN packets to the target (HPOT running in *RaspberryPi 1B*) at a rate of 1.000 packets per second. Meanwhile, the *Suricata* instance (NMA running in a *RaspberryPi 1B*) sniffed traffic on the same Ethernet segment. All the packets observed by the NMA were then published to the EMMS. Using the Event Analysis Module prototype running on the USER station, we executed the following query (Listing 5.4), to find out how many of the 10.000 SYN packets sent, were actually detected by the NMA.

```
PATTERN SEQ (TCP e1)
WHERE e1.tcp.flags = "syn"
AND e1.ip4.destination = "192.168.1.13"
PUBLISH "SynRequest" IN "tcpevents";
```

Listing 5.4: SYN requests query

This query returned 3.117 matches, which means a very significant event loss rate of around 69%. We believe, the most likely cause for this loss of packets is attributed to the hardware limitations of the *RaspberryPi 1B*. Although out of scope of this test, we performed a series of quick informal tests to provide a general idea of what could be expected from this kind of devices operating as NMA's, as we briefly describe:

- First, we repeated the same procedure under the same circumstances using *RaspberryPi 2B*, but it did not prove to perform any better than *RaspberryPi 1B*.
- Then, we repeated the same test, only varying the pace at which we sent the packets while maintaining the total of packets sent: 10.000. At half of the initial rate, 500 packets/sec, we found a slight improvement with *RaspberryPi 1B*, collecting 4.271 of the total 10.000 SYN packets sent, which equates to about 57% event loss. At 200 packets/sec, it collected 7.613 packets (24% loss). Finally, at 100 packets/sec, it managed to capture all 10.000 (0% loss).

As a final consideration, it is not much relevant for the problem in hand to capture every single incomplete connection attempt. It should be possible to detect the presence of a SYN-Flood attack by just looking at a representative sample of all the SYN packets sent. For the remaining of this experiment, we consider the original results obtained with *RaspberryPi 1B* (3.117 TCP SYN packets observed out of 10.000 sent).

Then we ran another query (Listing 5.5), to observe how many of these SYN requests were followed by a typical SYN flood attack pattern – a SYN-ACK response from the target followed by the absence of the expected handshake-completing ACK. This returned 2.828 positive matches. The fact that they are less than the preceding SYN's could be caused by one, or both, of two reasons: either the HPOT lost the SYN packets and didn't answer them or it did and the NMA didn't capture those events.

```
PATTERN SEQ (TCP e1, TCP+ e2[], ~(TCP e3))
WHERE e2[i].ip4.destination = e1.ip4.source
AND e1.ip4.destination = e2[i].ip4.source
AND e3.ip4.source = e1.ip4.source
AND e3.ip4.destination = e1.ip4.destination
AND e1.tcp.flags = "syn"
AND e2[i].tcp.flags = "syn-ack"
AND e3.tcp.flags = "ack"
AND e2[i].tcp.ack = e1.tcp.seq+1
AND e3.tcp.seq = e2[i].tcp.ack
AND e3.tcp.ack = e2[i].tcp.seq+1
PUBLISH "IncompleteTCPConnection" IN "threats";
```

Listing 5.5: Incomplete TCP connections query

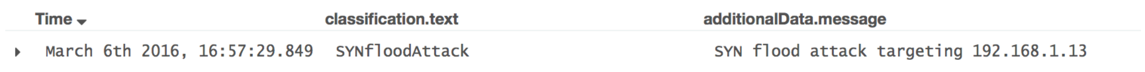
The middle event – *e2* – is represented as a Kleene closure because as it is never acknowledged, it is usually retransmitted by the target, which originates multiple matches for the same event. That, in turn would potentially generate multiple matches for the same handshake, if *e2* had been declared as a single event. This way, multiple matches for the same handshake are always identified as one.

Finally, we ran the following query (Listing 5.6), which looks for the occurrence of more than 1.000 failed TCP handshakes, targeting the same host. It then publishes each pattern match with an appropriate message including the target IP address.

```
PATTERN SEQ(IncompleteTCPConnection+ events[] IN "threats")
WHERE events[i].e1.ip4.destination =
events[j].e1.ip4.destination
HAVING count(events) > 1000
PUBLISH "SYNfloodAttack" IN "attacks"
DESCRIPTION "SYN flood attack targeting "+
first(events).e1.ip4.destination;
```

Listing 5.6: SYN flood attack query

Our initial idea was to design a query correlating the quantity of observed events and the observation time frame. For example, we wanted to detect any occurrence of more than 1.000 *IncompleteTCPConnection* events in a 10 seconds interval. However, our implemented prototype exhibited some particular issues when we attributed a WITHIN clause to a Kleene closure. The problem was that the algorithm, would have to consider every possible set of events that occurred within any possible 10 seconds window. That was not computationally feasible, which led us to exclude the WITHIN clause. This query, as expected, returned one positive match (Figure 5.10).



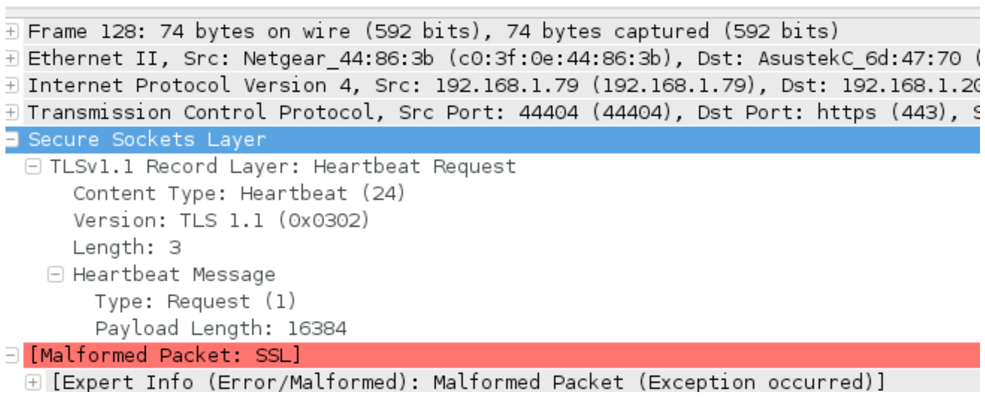
```
Time ▾                         classification.text           additionalData.message
▸ March 6th 2016, 16:57:29.849  SYNfloodAttack                SYN flood attack targeting 192.168.1.13
```

Figure 5.10: SYN flood results (Kibana)

5.3.3 SSL Heartbleed

The Heartbleed [CVE-2014-0160, Heartbleed] is a devastating vulnerability recently found (April 2014) in some older versions of the *OpenSSL* library – a commonly used implementation of the SSL/TLS protocol. It exploits a bug, found in *OpenSSL* versions 1.01 and 1.02 beta, related to the Heartbeat Extension for TLS. The Hearbeat mechanism provides a way to test or keep a secure connection alive, by refreshing the session security association parameters, instead of having to reestablish it with a new TLS handshake, after a certain period of inactivity. This is done by sending a message known as a Heartbeat Request (a standardized message type in the Record Layer Protocol (RLP) of the TLS protocol stack. This message consists of two components: an arbitrary payload and an explicit indication about the size of that same payload.

Figure 5.11 presents a typical traffic flow of an attacker sending a heartbeat request, in this case detected with the *Wireshark* protocol analyzer tool[7].



```
⊞ Frame 128: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
⊞ Ethernet II, Src: Netgear_44:86:3b (c0:3f:0e:44:86:3b), Dst: AsustekC_6d:47:70 (
⊞ Internet Protocol Version 4, Src: 192.168.1.79 (192.168.1.79), Dst: 192.168.1.20
⊞ Transmission Control Protocol, Src Port: 44404 (44404), Dst Port: https (443), S
⊟ Secure Sockets Layer
  ⊟ TLSv1.1 Record Layer: Heartbeat Request
      Content Type: Heartbeat (24)
      Version: TLS 1.1 (0x0302)
      Length: 3
    ⊟ Heartbeat Message
        Type: Request (1)
        Payload Length: 16384
⊟ [Malformed Packet: SSL]
  ⊞ [Expert Info (Error/Malformed): Malformed Packet (Exception occurred)]
```

Figure 5.11: Hearbeat attack

[7] Available on https://www.wireshark.org (accessed on 27/Mar/2016).

This tool can detect malformed heartbeat requests (as showed in the "Malformed Packet: SSL" line) but the packet capture is typical when a heatbleed attack is launched by an originator. As we can see in figure 5.11, the length of the TLS 1.1 record layer packet (carrying the Hearbeat Request message) has a length of 3 bytes, but Hearbeat's payload has a field Payload Length with 16K bytes.

The response to Hearbeat request messages should contain the exact same payload initially sent by the instigator in the Payload Length field. So, the way this mechanism can be exploited is by sending a maliciously crafted Heartbeat Request containing a short payload together with a fake much larger size indication. The receiver of the Heartbeat Request will eventually store the payload in memory while it prepares the response. The above-mentioned versions of *OpenSSL*, handle the Heartbeat Request blindly trusting the size provided by the other party and failing to verify the actual size of the message and do the appropriate bounds checking. In this case, the result is that the response message contains the payload the sent by the attacker followed by what happened to be next to it in the victim's memory. Since *OpenSSL* is meant to provide security to sensitive data, it will most likely have, in its memory, sensitive data like for instance the X.509 certificates' private keys, names and password of users, session-cookies, etc.

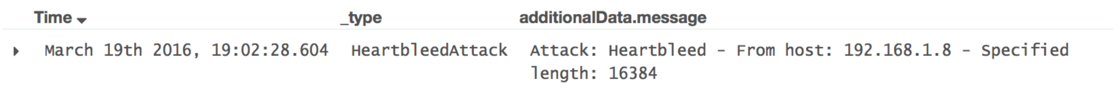
To show how a Heartbleed attack is detected by our DHIDS platform, we set up a basic SSL Honeypot script running on the HPOT. It simply accepts SSL/TLS connections and notifies on the console about the attacker address. On the client side (USER station), we used a Python script to inject the Heartbleed attack. For event capturing purposes, we used two techniques. First, we had an instance of *Suricata* running alongside with the SSL Honeypot (on the HPOT), logging all network traffic. Second, the SSL Honeypot script prints the source addresses of suspected Heartbleed scans to the console. To capture these messages as events into our EMMS, we redirected its standard output to a Generic Agent process, which interprets each line and encapsulates it into an event according to the interoperability format, and publishes it on the Event Dissemination System.

From the server point of view, this attack is known for not leaving any traceable occurrence. However, this is not the case with network events, which usually tell a more complete story. The following query (Listing 5.7) looks for SSL packets with content type "Heartbeat" (24) and heartbeat type "Request"

(1), according to the Heartbeat extension protocol specification [RFC6520]. It also looks at the specified payload length, which we know that in normal cir- cum- stances should always be less than the actual frame size. When this is not the case (and the other criteria is also met), a positive match for a Heartbleed attack is generated. This query returned one positive match, as shown in next partial screen-shot from *Kibana* (Figure 5.11).

```
PATTERN SEQ (SSL e1)
WHERE e1.ssl.record_type = 24
AND e1.ssl.heartbeat_type = 1
AND e1.ssl.heartbeat_payload_length > e1.frame.lenght
PUBLISH "HeartbleedAttack" IN "attacks"
MESSAGE "Attack: Heartbleed - From host:
"+e1.ip4.source+" - Specified length: " +
e1.ssl.heartbeat_payload_length;
```

Listing 5.7: Heartbleed attack query based on network traffic



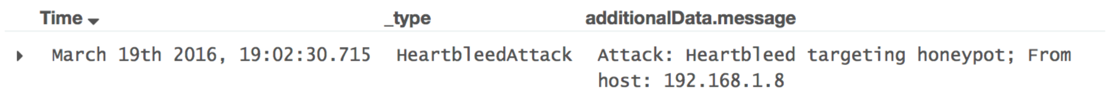
```
March 19th 2016, 19:02:28.604   HeartbleedAttack   Attack: Heartbleed - From host: 192.168.1.8 - Specified
                                                   length: 16384
```

Figure 5.12: Heartbleed attack event based on network traffic (*Kibana*)

A simpler way of detecting this attack is to use the events produced by the Honeypot itself. We did this with the following query (Listing 5.8), which simp- ly looks for any occurrence of a “HeartbleedHoneypot” event (this event type was set by configuration on the Generic Agent that is listening to the Honeypot) and notifies so, producing an event in the “attacks” index. As expected, this query produced one match, as shown in Figure 5.12.

```
PATTERN SEQ (HeartbleedHoneypot hb)
PUBLISH "HeartbleedAttack" IN "attacks"
MESSAGE "Attack: Heartbleed targeting honeypot;
From host: "+hb.source;
```

Listing 5.8: Heartbleed attack query based on Honeypot events



```
March 19th 2016, 19:02:30.715   HeartbleedAttack   Attack: Heartbleed targeting honeypot; From
                                                   host: 192.168.1.8
```

Figure 5.13: Heartbleed attack event based on Honeypot events (*Kibana*)

In this particular case, the Heartbleed attack was not successful because we were using a vulnerable *OpenSSL* version. However, we could formulate a query to detect if there occurred an accomplished Heartbleed attack by comparing the Heartbeat Request size against the Heartbeat Response (type 2, according to [RFC6520]) size, as suggested in Heartbleed official website [Heartbleed]. We can do so with DHIDS-QL as Listing 5.9 exemplifies. Predictably, this query returned no matches.

```
PATTERN SEQ (SSL e1, SSL e2)
WHERE e1.ssl.record_type = 24
AND e1.ssl.heartbeat_type = 1
AND e1.ssl.heartbeat_payload_length > e1.frame.lenght
AND e2.ip4.source = e1.ip4.destination
AND e2.ip4.destination = e1.ip4.source
AND e2.ssl.record_type = 24
AND e2.ssl.heartbeat_type = 2
AND e2.frame.lenght > e1.frame.lenght
PUBLISH "AccomplishedHeartbleedAttack" IN "attacks"
MESSAGE "Attack: Accomplished Heartbleed - From host:
"+e1.ip4.source+" - Specified length: " +
e1.ssl.heartbeat_payload_length;
```

Listing 5.9: Accomplished Heartbleed query example

5.4 Evaluation Summary

In this chapter, we demonstrated how the DHIDS prototype effectively collects, processes and stores intrusion-events, detected by possible multiple and heterogeneous distributed probes. The reported experiments, among other conducted evaluations, are representative of the potential of the DHIDS platform design and implementation. From our experimental observations we detected which components in our implementation, and to what extent, could become bottlenecks, and discussed what possible solutions are available to prevent it. The message-queuing component has demonstrated to be easily scalable and a reliable substrate for intrusion-detection event dissemination from multiple probes to the central management solution built on top of the *Elasticsearch* (or ELK) software stack. Then, we demonstrated that the system is capable of detecting "real-life" intrusion occurrences (namely: SQLi, SYN-Flood, and Heartbleed attacks) relying on a variety of event sources.

Although we didn't present query execution performance tests to provide measurable data about expectable query execution times, we have observed, during the previous tests and other experiments, some typical behaviors regarding the query execution performance. Based on that, we can make the following considerations:

- The execution time depends on the type of events and pattern we are querying. Querying events that are very specific or rare is usually very fast.
- The opposite is also true. Querying common events may take a long time to execute. Furthermore, if the pattern is composed of many different events and they are not rare, then the query may take a very long time to execute or ultimately fail.
- As an example, the query that looks for incomplete TCP connections took a long time to execute (about 2 minutes for the used data set), as TCP handshakes, and therefore SYN packets, are very common occurrences in the network traffic. On the other hand, all the other queries used in this chapter executed quickly (typically, in a few milliseconds).
- As a corollary, even that the DHIDS platform was not designed as a real-time-intrusion detection system, in the sense that time-bound metrics for event-detection always obey to real-time threshold guarantees, the practical observation show, however, that the system exhibits an interesting potential for soft-real-time guarantees. This can be provided namely by using more powerful probing devices, a cluster-based solution for the *RabbitMQ* based event dissemination and a powerful deployment of a cluster-ELK infrastructure.

We should note that the above considerations are conditioned by the specific test environments used. It is expectable that if *Elasticsearch* were running on bigger cluster with more powerful nodes, the execution of queries would also be faster.

6

Conclusions

In this thesis, we proposed a Distributed and Hybrid Intrusion Detection System (DHIDS), addressed as an auditing platform that leverages on different intrusion detection components, synergistically combined in a pervasive monitoring system. Our objectives addressed the identified shortcomings on current IDS technology, building the DHIDS platform as a proposed solution to large-scale monitoring environments.

Our proposal combines the following relevant contributions:

1. It supports a pervasive and diverse environment of probing agents spread all across the network, leveraging from the diversity of different technological options and specializations, including NIDS, HIDS and Honeypot solutions.
2. It offers scalability and extendibility by means of a distributed publish/subscribe middleware and an interoperability format, decoupling the event capturing and the event analysis process.
3. It features a scalable Event Monitoring and Management System, as an auditing platform built on top of an elastic data repository, to store and to manage events for audit-trail correlation analysis.
4. It provides a query-based language (DHIDS-QL) to express attack or signatures for querying security audit-trails, as well as a runtime implementation to interpret and execute queries over the data-repository.

Some issues can be emphasized, considering the contribution, namely comparing with other intrusion detection approaches. First, by adopting different base technology to build a diverse probing environment, the platform over-comes effectiveness drawbacks of conventional IDS technology, according to its specialization and limitations, and when performing in isolation. This complementarity is an open way to deal with well-known problems of such IDS technology, namely considering scalability, reliability and effectiveness issues (i.e., tradeoffs between false-positive and false-negative ratios).

Second, in the DHIDS platform, security events are detected by multiple probes (possibly using specific local capture formats), filtered and converted to a canonical JSON-based message format representation, inspired by the RFC4765 XML-message format. This standardized approach allows that events are uniformly conveyed through a generic publish/subscribe middleware to a dedicated logging and auditing system, the EMMS. In EMMS, all observed events can be aggregated, correlated and queried, against suspicious intrusion attack signature patterns involving heterogeneous evidences. This approach provides some relevant advantages: (1) it is a much more extensible architecture, with the possibility to integrate new pervasive intrusion detection probes, decoupling their internal functionality in the global DHIDS architecture, and (2) the integration of existent technology that adheres to the RFC4765 as a supported native format is almost immediate[8].

Finally, in the proposed DHIDS platform, attack signatures are expressed by means of a declarative pattern query-language (DHIDS-QL), providing event-correlation semantics, using events as base elements for extensible possible rich-aggregations and correlations, modeling possible sophisticated attack evidences. This allows for a global intrusion detection and monitoring environment when compared with specific management functions on conventional IDS technology, which tend to consider sub-sets of discrete events.

We implemented the DHIDS platform as a proof-of-concept prototype, with a particular contribution and emphasis on the design and development of the proposed query language model and query execution runtime for the EMMS. Then, we experimentally evaluated the prototyped platform. For this purpose, we built two different test-bench environments and conducted different observations, regarding event processing throughput, expectable perfor-

[8] In that case, there would be a need for direct conversion from XML to JSON. As a prototype implementation choice, we adopted JSON equivalent to XML model suggested in RFC4765, as JSON imposes significantly lighter communication overhead than XML and it also simplified the implementation effort.

mance and scalability properties of the various adopted devices and platforms, expressiveness and effectiveness of the conceived attack signature expressing language. The results support our belief that the DHIDS platform is a realistic and interesting approach to implement a pervasive, diverse and effective Intrusion Detection System with centralized management and high-level event correlation semantics. The experimental evaluation also demonstrates that the designed platform and its prototype exhibit interesting indicators of potential for scalability, reliability, extensibility and availability.

6.1 Future Work Directions

Despite the results obtained by the prototype developed in the context of the thesis elaboration, many relevant work directions must be addressed as future work, in order to arrive to a final deployed solution, eventually used in a real-monitoring environment. We advance some open issues that could be addressed as next steps, starting from our initial implementation and proof-of-concept evaluations.

A more extensive evaluation is certainly needed to better validate the effectiveness of the proposed solution. Namely:

- Tests regarding the potential for scalability of the EMMS in a clustered environment, targeting both, event capturing and processing, and query processing performance. In addition to these, experiments exploring replication, load balancing and clustering conditions regarding *Elasticsearch* would also be of interest.
- Experiments targeting the system's "real-time" capabilities, when capturing and analyzing events live, possibly in a real pre-production environment. This could include the further development of the agent devices, specifically the development of embedded agents with specialized, native and optimized functions for event-capture, local filtering, event conversion and event publishing.

Another interesting research and development direction would be the further exploration of the software ecosystem around the ELK stack; namely, the use of *Kibana* as a real-time event monitoring visualization dashboard.

We recognize that there is also a space for improvement regarding DHIDS-QL, namely in the expressiveness of the proposed language. This expressiveness could be expanded, for example with the addition of more pattern structures. Not all patterns can be expressed by a sequence. A pattern could be such that looks for the occurrence of a set of events regardless of the order. Also, pattern structures could be embedded in other pattern structures. This effort for the language enrichment could begin by analyzing different kinds of attacks and evolve the language while developing new attack signatures, taking advantage of the extensibility possibilities in the declarative language behavior and the way it is supported by the ELK/*Logstash* component.

Finally, a revision of the current implementation with possible refinement and optimizations of the developed software, as well as, new selected tests, can open the possibility to address the dissemination of the DHIDS platform as an open-source platform. In this direction, an open task is to consider a publication to a national conference, or to an international conference or workshop.

References

[Abadi03] Daniel J. Abadi, et al. “Aurora: A new model and architecture for data stream management”, *The VLDB Journal*, 12:120–139, August 2003.

[Abadi05] Daniel J. Abadi, et al. “The design of the borealis stream processing engine”, *CIDR*, pages 277–289, 2005.

[Agrawal08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, Neil Immerman. “Efficient pattern matching over event streams”, *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD ’08, pages 147–160, 2008.

[AIDE] “AIDE – Advanced Intrusion Detection Environment”, http://aide.sourceforge.net (accessed on 02/Feb/2014).

[AmazonSQS] “Amazon SQS – Message Queuing Service - AWS”, https://aws.amazon.com/sqs/ (accessed on 19/Feb/2016).

[AMQP] “AMQP – Advanced Message Queuing Protocol Specification, Version 0.9.1, AMQP Working Group”, http://www.amqp.org/specification/0-9-1/amqp-org-download (accessed on 15/Jan/2015), November 2008.

[Anderson08] Ross Anderson, “Security Engineering: A Guide to Building Dependable Distributed Systems”, 2nd Edition, Wiley, April 2008.

[Axelsson00a] Stefan Axelsson. “Intrusion Detection Systems: A Survey and Taxonomy”, Technical Report, Dep. Of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, March 2000.

[Axelsson00] Stefan Axelsson, “The Base-Rate Fallacy and the Difficulty of Intrusion Detection”, *ACM Transactions on Information and Systems Security (TISSEC)*, pp. 186-205, Aug 2000.

[Birman05] Kenneth P. Birman. "Reliable Distributed Systems: Technologies, Web Services and Applications”, Springer, 2005.

[Blankstein11] Aaron Blankstein. “Analyzing Audit Trails in the Aeolus Security Platform.” Technical Report, Massachusetts Institute of Technology (MIT), 2011.

[Bruening03] Derek Bruening, Timothy Garnett, Saman Amarasinghe. “An infrastructure for adaptive dynamic optimization”, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO ’03, pages 265–275, 2003. IEEE Computer Society.

[Cantrill04] Bryan M. Cantrill, Michael W. Shapiro, Adam H. Leventhal. “Dynamic instrumentation of production systems”, *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC ’04, pages 2–2, 2004. USENIX Association.

[Costa14] Nuno Costa, Pedro Tibocco, Mauro Cruz, Henrique Domingos. “MOM Comparative evaluation for the Pulse Pilot”, *Techical Report TR-PULSE-2-0017-C-2015, EF Tecnologias S.A - PT Comunicações, Pulse2 Pilot Project Documentation*, Sep 2014

[Demers07] Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker White. “Cayuga: A general purpose event monitoring system”, *Conference on Innovative Data Systems Research*, pages 412–422, 2007.

[Diao07] Yanlei Diao, Neil Immerman, Daniel Gyllstrom. “SASE+: An Agile Language for Kleene Closure over Event Streams”, 2007.

[CVE-2014-0160] “Common Vulnerabilities Exposures – CVE-2014-0160”, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 (accessed on 19/Feb/2016).

[Dionaea] "GitHub – rep/dionaea: dionaea low interation honeypot (forked from: dionaea.carnivore.it) " https://github.com/rep/dionaea (accessed on 19/Feb/2016)

[Elasticsearch] "Elasticsearch" – and End-to-End search and analytics platform, http://www.elasticsearch.org/overview (accessed on. 08/Feb/2014)

[Eugster03] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. In ACM Computing Surveys, Volume 35, Issue 2, pp. 114-131, June 2003.

[Gollman11] Dieter Gollman. "Computer Security", J. Wiley & Sons, 3rd Edition, 2011

[Heartbleed] "Heartbleed Bug", http://heartbleed.com (accessed on 19/Feb/2016).

[HornetQ] "HornetQ – putting the buzz on messaging – JBoss Community", http://hornetq.jboss.org (accessed on 19/Feb/2016).

[hping] "Hping – Active Network Security Tool", http://www.hping.org (accessed on 01(Mar/2016)

[Huang09] P-Sheng Huang, C. Yang, T. Ahn, "Design and implementation of a distributed early warning system combined with intrusion detection system and honeypot", *Proceedings of the 2009 International Conference on Hybrid Information Technology*, ACM, Daejeon, Korea, August 2009.

[IDMEF-Java] "A Java Implementation of IETF RFC4765: Intrusion Detection Message Exchange Format (IDMEF) Experimental Protocol", *https://github.com/cr-labs/RFC4765* (accessed on 02/Feb/2015).

[iPerf] "iPerf – The TCP, UDP and SCTP network bandwidth measurement tool", https://iperf.fr (accessed on 20/Fev/2016).

[JavaCC] "Java Compiler Compiler tm (JavaCC tm) - The Java Parser Generator", https://javacc.java.net (accessed on 15/Jan/2016).

[jNetPcap] "jNetPcap Open-source | Protocol Analysis SDK", http://jnetpcap.com (accessed on 20/Feb/2016).

[Johnson14] Thienne Johnson, Loukas Lazos. "Network Anomaly Detection Using Autonomous System Flow Aggregates", *Proc. of IEEE GLOBECOM 2014*, Austin, Texas, 2014.

[JSON-java] "GitHub - stleary/JSON-java: A reference implementation of a JSON package in Java.", https://github.com/stleary/JSON-java (accessed on 20/Feb/2016)

[Kaufman02] Charlie Kaufman, R. Perlman, Mike Speciner. "Network Security – Private Communication in a Public World", 2nd Edition, Prentice Hall, 2002.

[Kibana] "Kibana" – Open-source data visualization platform, https://www.elastic.co/products/kibana (accessed on 15/Jan/2016)

[Kiczales01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. "An overview of AspectJ." *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, 2001. Springer-Verlag.

[Kim10] Taesoo Kim, Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek. "Intrusion recovery using selective re-execution." *Proc. of the 9th USENIX OSDI Conference, Operating Systems Design and Implementation*, pages 1-9, 2010.

[King05] Samuel T. King, Peter M. Chen. "Backtracking Intrusions." In *ACM Trans. on Computer Sytsems*, Vol. 23(1), pages 51–76, February 2005.

[Kippo] "kippo SSH Honeypot" *https://code.google.com/p/kippo* (accessed on 8/Feb/2015).

[Kothari02] Pravin Khotari. "Intrusion Detection Interoperability and Standradization", GSEC, SANS Institure, InfoSec Reading Room Series, Feb 2002.

[Kreibich04] Christian Kreibich, J. Crowcroft. "Honeycomb: Creating Intrusion Detection Signatures using Honeypots", *SIGCOMM Computer Commiunication Review*, ACM, Vol 34 Issue 1, January 2004.

[Kreps11] Jay Kreps, Neha Narkhede, Jun Rao. "Kafka: a Distributed Message System for Log Processing". In Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece (2011).

[Kumar14] M. Kumar. "Distributed Intrusion Detection System Scalability Enhancement using Cloud Computing", GESJ: Computer Science and Telecommunications 2014 | No.1(41).

[Lazarevic05] A. Lazarevic, V. Kumar, J. Srivastava. "Managing Cyber Threats: Issues, Approaches and Challenges" Chap. 2 – Intrusion Detection: A Survey Springer Series: Massive Computing Vol.5, May 2005

[Lewis99] Ryhs Lewis. "Advanced Messaging Applications with MSMQ and MQSeries". Que; 1st edition (December 17, 1999).

[Logstash] "Logstash | Elastic", https://www.elastic.co/products/logstash (accessed on 20/Feb/2016).

[Luk05] Chi-Keung Luk, et al. "Pin: building customized program analysis tools with dynamic instrumentation." *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference, on Programming language design and implementation*, pages 190–200, 2005.

[Mairh11] A. Maihr, D. Barik, K. Verma, D. Jena. "Honeypot in Network Security", *ICCCS 11 – Proc. of Intl Conference on Communication, Computing & Security*, ACM, Rourkela-India, Feb 2011.

[Mitchell14] Robert Mitchell, Ing-Ray Chen. "A Survey of Intrusion Detection Techniques for Cyber-Physical Systems", *ACM Computing Surveys*, Vol.46, No. 4, March 2014.

[MongoDB] "MongoDB for GIANT Ideas | MongoDB", https://www.mongodb.org (accessed on 19/Feb/2016).

[MSMQ] "Message Queuing (MSMQ) - MSDN - Microsoft", https://msdn.microsoft.com/enus/library/ms711472(v=vs.85).aspx (accessed on 19/Feb/2016).

[Oki93] Brian Oki, Manfred Pfluegl, Alex Siegel, Dale Skeen. "The Information Bus – An Architecture for Extensible Distributed System". In 14th ACM Symposium on Operating System Principals, (Asheville, NC), 1993.

[OracleAQ] "Oracle Advanced Queuing", https://docs.oracle.com/cd/B28359_01/java.111/b31224/streamsaq.htm (accessed on 19/Feb/2016).

[OSSEC] "OSSEC – Open Source Host Based Intrusion Detection System", *http://www.ossec.net* (accessed on 02/Feb/2015).

[OWASP13] "OWASP Top 10 – 2013. The Ten Most Critical Web Application Security Risks", http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf (accessed on 01/Mar/2016)

[Porras02] P. A. Porras, M. W. Fong, A. Valdes. "A Mission-Impact-Based Approach to INFOSEC Alarm Correlation". *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*. 2002

[Qiao13] Peili Qiao, Shan-Shan Hu, Ji-Qiang Zhai. "Design and implementation of dynamic hybrid Honeypot network", *Proc. SPIE 8752, Modeling and Simulation for Defense Systems and Applications VIII*, May 2013.

[RabbitMQ] "RabbitMQ - Messaging that just works", http://www.rabbitmq.com (accessed on 08/Feb/2015).

[RaspberryPi] "Raspberry Pi - Teach, Learn, and Make with Raspberry Pi", https://www.raspberrypi.org (accessed on 20/Feb/2016).

[Raspbian] "Download Raspbian for Raspberry Pi", https://www.raspberrypi.org/downloads/raspbian/ (accessed on 20/Feb/2016)

[RFC4765] H. Debar, D. Curry, B. Feinstein. "The Intrusion Detection Message Exchange Format (IDMEF)" *IETF, Network Working Group*, March 2007.

[RFC4766] M. Wood, M. Erlinger. "Intrusion Detection Message Exchange Requirements" *IETF, Network Working Group*, March 2007.

[RFC4767] B. Feinstein, G. Matthews. "The Intrusion Detection Exchange Protocol (IDXP)" *IETF, Network Working Group*, March 2007.

[RFC6520] R. Seggelmann, M. Tuexen, M. Williams. "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension". IETF, Standards Track, February 2012.

[SAMHAIN] "The SAMHAIN File Integrity / Host Based Intrusion Detection System", *http://la-samhna.de/samhain/* (accessed on 02/Feb/2014).

[Scarfone07] Karen Scarfone, P. Mall. "Guide to Intrusion Detection and Prevention Systems (IDPS)", NIST – National Institute of Standards and Technology, Special Pub. 800-94, Feb 2007.

[Schupp00] Steve Schupp, "Limitations of Network Intrusion Detection", *Global Information Assurance Certification*, SANS Institute, December 2000.

[Snort] "Snort", *http://www.snort.org* (accessed on 02/Feb/2014).

[Stallings14a] William Stallings, "Network Security Essentials: Applications and Standards", 5th Edition, Prentice Hall, 2014.

[Stallings14] William Stallings, Lawrie Brown. "Computer Security: Principles and Practice", Prentice Hall – Pearson, 3rd Edition, August 2014.

[Suricata] "Suricata – Open Source IDS/IPS/NSM Engine", *http://suricata-ids.org* (accessed on 02/Feb/2014).

[Swift07] David Swift, "A Practical Application of SIM/SEM/SIEM Automating Threat Identification", Technical Report, SANS Institute – Infosec Reading Room Series, 2007.

[Tripwire] Open Source Tripwire, http://www.tripwire.org (accessed on 08/Feb/2015)

[w3af] "w3af – Open Source Web Application Security Scanner", http://w3af.org (accessed on 1/Mar/2016).

[Wackopicko] "GitHub - adamdoupe/WackoPicko: WackoPicko is a vulnerable web application used to test web application vulnerability scanners.", https://github.com/adamdoupe/WackoPicko (accessed on 20/Feb/2016).

[Wang10] H. Wang, Q. Chen. "Design of Cooperative Deployment in Distributed Honeynet System", *Proc. of CSCWD 2010 - 14th Intl. IEEE Conference on Computer Supported Cooperative Work in Design*, 2010.

Appendix A: IDMEF XML Document Type Definition

```
   <!ELEMENT Alert                        (
          Analyzer, CreateTime, DetectTime?, AnalyzerTime?,
          Source*, Target*, Classification, Assessment?, (Tool-
   Alert | OverflowAlert | CorrelationAlert)?, AdditionalData*)>
      <!ATTLIST Alert
          messageid          CDATA                  '0'
          %attlist.global;

        >



<!ELEMENT Analyzer                        (
       Node?, Process?, Analyzer?
     )>
   <!ATTLIST Analyzer
       analyzerid          CDATA                  '0'
       name                CDATA                  #IMPLIED
       manufacturer        CDATA                  #IMPLIED
       model               CDATA                  #IMPLIED
       version             CDATA                  #IMPLIED
       class               CDATA                  #IMPLIED
       ostype              CDATA                  #IMPLIED
       osversion           CDATA                  #IMPLIED
       %attlist.global;

        >



<!ELEMENT Classification (
           Reference*
     )>
   <!ATTLIST Classification
       ident               CDATA                  '0'
       text                CDATA                  #REQUIRED

        >
```

```
<!ENTITY % attvals.adtype                 "
       ( boolean | byte | character | date-time | integer |
ntpstamp |
         portlist | real | string | byte-string | xmltext )
     ">



   <!ELEMENT AdditionalData           (
     (boolean | byte         | character | date-time |
      integer | ntpstamp     | portlist  | real      |
      string  | byte-string | xmltext  )
    )>

   <!ATTLIST AdditionalData
       type               %attvals.adtype;        'string'
       meaning            CDATA                   #IMPLIED
       %attlist.global;

        >



  <!ELEMENT CreateTime           (#PCDATA) >
   <!ATTLIST CreateTime
       ntpstamp           CDATA                   #REQUIRED
       %attlist.global;
     >



   <!ELEMENT DetectTime           (#PCDATA) >
   <!ATTLIST DetectTime
       ntpstamp           CDATA                   #REQUIRED
       %attlist.global;

        >
```

Appendix B: DHIDS-QL Abstract Syntax

DHIDS-QL is defined by the following BNF grammar. Reserved words and terminal symbols are presented in boldface font.

```
<query>          ::= PATTERN <pattern>
                     [WHERE <logical_exp>]
                     [WITHIN <time_window>]
                     [HAVING <logical_exp>]
                     ( PUBLISH <publish> | RETURN <expression> )

<pattern>        ::= <pattern_type> ( <element> [, <element>]* )

<pattern_type>   ::= SEQ

<element>        ::= <event> | ~(<event>)

<event>          ::= <type> ( [+] <id> [] | <id> ) [ IN <id> ]

<logical_exp>    ::= <logic_term> [OR <logical_exp>]

<logical_term>   ::= <logical_fact> [AND <logical_term>]

<logical_fact>   ::= <comparison> | NOT <logical_fact>

<comparison>     ::= <expression>
                     [( > | < | = | <> | >= | <= | CONTAINS )
                     <expression> ]

<expression>     ::= <term> [ ( + | - ) <expression> ]

<term>           ::= <fact> [ ( * | / ) <term> | . <field> ]

<fact>           ::= <simple_fact> | - <fact>
```

```
<simple_fact>     ::= integer | string | <id> [<id>] |
                  ( [<expression>]* ) | <id> | ( <logical_exp> )

<time_window>     ::= integer <letter>*

<publish>         ::= <expression> IN <expression>
                  [ DESCRIPTION <expression> ]

<id>              ::= <letter> <letter_digit>*

<type>            ::= <letter> <letter>*

<letter>          ::= [a-z] | [A-Z]

<letter_digit>    ::= letter | [0-9]
```

Appendix C: Pre-captured Data Sets

Among the representative log datasets available on the Internet that can be considered for our experimental purposes, we studied the following ones:

- **DARPA Intrusion Detection Data Sets**

  Available on: http://www.ll.mit.edu/ideval/data/ (accessed on 20/Feb/2016).

  These data sets were created in 1998/1999 for research purposes, specifically for IDS evaluation. These include several gigabytes of data captured in a simulated network in *PCAP* format including several attack occurrences alongside with normal traffic.

- **Public *PCAP* files list from NETRESEC**

  Available on: http://www.netresec.com/?page=PcapFiles (accessed on 20/Feb/2016).

  This is a list of publicly available data sets in *PCAP* format from various sources.

- **USA Military Academy Datasets**

  Available on:
  http://www.westpoint.edu/crc/SitePages/DataSets.aspx (accessed on 20/Feb/2016).

  These are real data sets recorded in the internal USMA network. These include: Network traffic in the *PCAP* format DNS message logs, Apache web server logs (both access and error) and Snort alert logs.