POLITECNICO DI TORINO

Master's Degree in Computer Engineering
Cybersecurity Focus

Security automation for web-based attacks

Supervisors
Prof. Fulvio Valenza
Prof. Riccardo Sisto
Dr. Daniele Bringhenti

Candidate
Francesco Grande

Accademic Year 2023/2024

Summary

One of the most emerging innovations of the last years is the security automation of networks. Various technologies have been developed to streamline all security processes within different network environments. Two significant advancements in this field are SDN (Software-Defined Network) and NFV (Network Function Virtualization), which have elevated automation to a central role in cybersecurity. These technologies enable the construction of networks where certain nodes are not tied to specific hardware but are instead virtualized, allowing them to deploy network functions through software. Within this context, the VEREFOO (VErified REfinement and Optimized Orchestration) framework has been developed; it is capable of obtaining an automated and optimal allocation of some NSFs (Network Security Functions), necessary to fulfil a set of NSRs (Network Security Requirements) provided as input, starting from a logical description of the network topology. VEREFOO ensures optimality and formal correctness of its solutions through the formulation of a MaxSMT problem solved efficiently by z3, a theorem prover developed by Microsoft Research. This thesis goal is to enhance VEREFOO by designing and implementing a solution capable of automatically configuring and deploying Web Application Firewalls within a network. Web Application Firewalls are firewalls which, in addition to regular filtering rules, specialize in web traffic and web applications. This thesis specifically focuses on the defense against web-based attacks, which are becoming increasingly frequent and sophisticated. By utilizing a Web Application Firewall as virtual network function, and exploiting the ModSecurity Core Rule Set, the proposed solution aims to protect web applications from a wide range of threats such as SQL injection and XSS.

Acknowledgements

First and foremost, I would like to thank my supervisors, Professor Fulvio Valenza and Professor Riccardo Sisto, who gave me the opportunity to work on this thesis project and were always available to help me solve any problems that arose. I also extend my gratitude to Dr. Daniele Bringhenti, whose advice enabled me to carry out my work in the best possible way.

A huge thanks goes to my parents, who have supported me from the very beginning in my decision to study in another city, despite the emotional toll they would experience and the financial burden it entailed. I still remember my mother's tears on the day I moved to Turin. I hope I have made you proud by reaching this milestone.

A heartfelt thanks also goes to the rest of my family: my grandmother, my uncles and aunts, my brother, and my sisters, who have always supported me despite the distance. Every time I came back home for holidays, they made me feel as though I had never left.

A special thanks to my uncles and cousins in Turin, who welcomed me as an additional family member during the abundant Sunday lunches and were always available for any need. In particular, I would like to thank my aunt Paola, who treated me like a son, never hesitating to offer her help with laundry, drying, and ironing, despite her busy work schedule.

I would also like to thank my friends, with whom I shared wonderful moments of leisure during these university years away from home. First of all, my university colleagues (now former colleagues), with whom I bonded immediately and had wild times from the very first day during the memorable "cyber-secure" beer, thanks to Luca, Peppe, Lorenzo, and Ben. Thanks to you, I also got to see all of Italy, since you come from such different places. A special thanks also to the recent additions, Flavia and Lorenza, who brought some liveliness and a feminine touch to the group, particularly to Flavia, who quickly became my most loyal confidante. I must also thank my friends from the residence, who were like real brothers during our time together, making even the days spent at home enjoyable, sometimes even making

me stay up very late. Special thanks to Denis, Riccio, Zak, Fil, Momo, Candy, and The Judge. A heartfelt thanks also goes to my friends back home, with whom I never lost touch, a sign of a strong friendship. They made my time in Aversa equally memorable, and all of them tried their best to come to my graduation in Turin, despite their busy schedules and the long journey from across Italy. Among them, a special thanks goes to Luca, who has been like a brother to me since my first year of undergraduate studies, as well as a mentor in the field of computer science. We have grown together and matured in so many ways over the years. A big thanks also to everyone else I haven't mentioned but who has been part of this journey.

Finally, but no less important, I would like to express my deepest gratitude to my girlfriend, Samantha. Over this past year, she has shown me what it truly means to have someone by your side who supports you in every aspect of your life. She has been a wise advisor, a companion on countless trips and dinners, and a person who is kind, respectful, sincere, and beautiful. Despite the distance between us, she has always been ready to jump on the first train or flight to see me, even if she had been in the hospital that morning for her internship or had an upcoming exam. She has always put me first, showing me the depth of her feelings. Thank you for loving me so genuinely.

Contents

List of Figures

List of Tables

Listings

Acronyms

ADP Allocation, Distribution, and Placement

AP Allocation Place

API Application Programming Interface

CRS Core Rule Set

CWE Common Weakness Enumeration

GUI Graphical User Interface

HLP High-Level Policies

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

ID Identification

IIS Microsoft Internet Information Services

IP Internet Protocol

JSON JavaScript Object Notation

LCP Longest Common Prefix

MaxSMT MAX-Satisfiability modulo theories

MLP Medium-Level Policies

NAT Network Address Translation

NFV Network Function Virtualization

NP Nondeterministic Polynomial-Time

NSF Network Security Function

NSR Network Security Requirement

OSI Open Systems Interconnection

OWASP Open Web Application Security Project

P Polynomial Time

PL Paranoia Level

RAM Random Access Memory

SAT Boolean Satisfiability Problem

SDN Software-Defined Network

SMT Satisfiability Modulo Theories

SNMP Simple Network Management Protocol

SQL Structured Query Language

TCP Transmission Control Protocol

UDP User Datagram Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

UTF Unicode Transformation Format

VEREFOO VErified REfinement and Optimized Orchestration

VNE Virtual Network Embedding

VPN Virtual Private Network

XML Extensible Markup Language

XSD XML Schema Definition

XSS Cross-Site Scripting

WAF Web Application Firewall

Chapter 1

Introduction

1.1 Objectives

In recent years new network technologies have emerged and among these the NFV (Network Function Virtualization) and SDN (Software-Defined Network) paradigms stand out for innovation. NFV is a network architecture concept that leverages virtualization technology to manage and deploy NSFs (Network Security Functions), such as firewalls, load balancers or VPNs (Virtual Private Networks). Traditionally, NSFs have been implemented on dedicated hardware devices, while now NFV transforms these network functions into software-based components that can run on standard, off-the-shelf hardware, such as servers or virtual machines. This leads to costs reduction by eliminating the need to purchase a dedicated physical device for each required network function, as well as a better flexibility because NSFs can be deployed, managed, and scaled more easily since they are software-based.

SDN, that comes across NFV, is a network architecture approach that separates the control plane from the data plane in network devices; the control plane is centralized and managed by a software-based controller, while the data plane remains on the network devices. SDN introduces the possibility to create a forwarding path by means of a software process.

Starting from these innovative technologies, the thesis objective was to extend and enhance the implementation of VEREFOO (VErified REfinement and Optimized Orchestration), a framework primarily designed to obtain an automated and optimal allocation of the NSFs necessary for satisfy NSRs (Network Security Requirements) within an indicated topology. The framework, in fact, receives as input a logical description of the network and a set of security policies.

To achieve these results, VEREFOO solves a MaxSMT (MAX-Satisfiability modulo theories) problem using z3, a theorem prover developed by Microsoft Research, and Verigraph, a framework developed by the Polytechnic of Turin, specialized in the verification of requirements for VNE (Virtual Network Embedding) scenarios.

This thesis aimed to allow VEREFOO to support a WAF (Web Application Firewall), that is a firewall which, in addition to regular filtering rules, specializes in web traffic and web applications. In particular, it has been implemented the WAF functionality capable of activating the OWASP (Open Web Application Security Project) rules from NSRs, exploiting the ModSecurity CRS (Core Rule Set), and it has been performed the integration with the already existing firewall module.

1.2 Outline

After a brief introduction and description of the thesis presented in Chapter [1], the remainder of the paper is structured as follows:

- **Chapter [2]**: This chapter, therefore, delves into the functionality of WAFs and their role in mitigating various attack vectors. It specifically highlights ModSecurity, an open-source WAF, and the OWASP CRS, a collection of pre-defined rules developed by OWASP for configuring ModSecurity and many other WAFs to defend against prevalent threats.
- **Chapter [3]**:This chapter first introduces the technologies that form the foundation of VEREFOO, namely SDN and NFV. Then the chapter focuses on VEREFOO itself, describing its core mechanisms, including the ADP module and the z3 theorem prover which are the main objectives of this thesis work.
- **Chapter [4]**: This is a transition chapter that outlines the main objectives of this thesis work, defining the correspondent approach adopted and the choices maded.
- **Chapter [5]**: This chapter addresses the analysis and modeling of Network Security Requirements concerning OWASP rules. In the first part, the model of the requirements is illustrated along with its corresponding XML schemas. In the second part, the MaxSMT problem is formulated by adapting the firewall constraints to the WAF model and defining additional constraints.
- **Chapter [6]**: This chapter is dedicated to the description of the implementation and validation of the adopted model. The first part presents the actual implementation in the context of both manual and automatic configuration,

ensuring that the outputs produced by the framework are correct. The second part is focused on the validation of the model through performance and scalability tests, highlighting the distinctive features of the solution in network topologies of varying sizes and with a variable number of NSRs.

- **Chapter** [7]: This chapter is dedicated to the conclusions and a reflection on future work that could stem from the results obtained in this thesis.

Chapter 2

Web Application Firewalls: ModSecurity and OWASP Core Rule Set

The aim of this chapter is to provide a comprehensive explanation of what a Web Application Firewall is and the mechanisms by which it operates. Additionally, this chapter seeks to explore the Open Web Application Security Project in detail and critically examine the potential application of a commercially available WAF, specifically ModSecurity. Through this analysis, we will delve into the technical aspects of WAF functionality, investigate the contributions of OWASP to web security, and assess the practical implementation and effectiveness of ModSecurity in safeguarding web applications.

2.1 Web Application Firewall

A WAF represents a critical layer of security within modern cybersecurity architectures, specifically designed to safeguard web applications from a spectrum of threats that traditional network firewalls may not adequately address. Unlike conventional firewalls, which focus on controlling traffic at the network perimeter, a WAF is strategically positioned to monitor, filter, and potentially block HTTP/HTTPS traffic that interacts with web applications, thereby defending against a variety of sophisticated attack vectors.

As illustrated in figure 2.1, in fact, a Network Firewall filters network traffic analyzing it up to the transport layer (TCP, UDP), while a WAF is able to analyze the traffic up to application level (HTTP/HTTPS), consequently more rules can

be applied. Furthermore the WAF also has visibility into the transport layer fields, so it also can filter the network traffic like the commmon firewall.

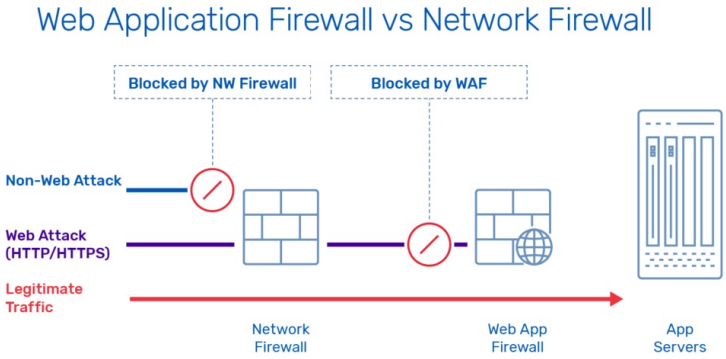

Figure 2.1: Web Application Firewall vs Network Firewall

The WAF is strategically positioned between the clients and the server, functioning as a reverse proxy for the back-end infrastructure. In technical terms, this configuration ensures that all HTTP requests from clients are routed through the WAF before reaching the web server. The WAF meticulously analyzes each request against a predefined set of rules to determine its legitimacy. If a request is identified as malicious, it is promptly blocked; otherwise, it is seamlessly forwarded to the web server for processing.

2.1.1 WAF structure

Let's now examine in greater detail the processes within a WAF, starting from figure 2.2.

A WAF typically involves three stages[1]:

1. **Pre-processor.** The pre-processor stage involves standardizing incoming data to a uniform format and discerning its nature. WAFs handle a variety of traffic types, including HTTP and HTTPS requests, which can encompass methods such as GET and POST, and data formats ranging from JSON to simple parameters. The pre-processor's role is to analyze these requests to identify and isolate user-submitted data for subsequent validation. Additionally, the pre-processor identifies and discards anomalies such as malformed URLs before they proceed to the input validation phase. This preliminary filtering is crucial, as input validation is a resource-intensive procedure that requires significant time and computational power. By ensuring that only relevant and correctly formatted requests undergo validation, the pre-processor

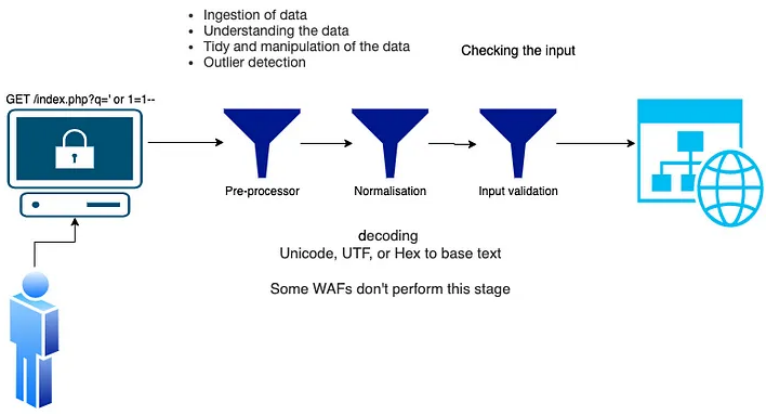

Figure 2.2: Internal workings of a WAF

optimizes the efficiency and effectiveness of the overall security process;

2. **Normalization.** Normalization aims to decode UTF or encoded values into their standard text representations. Research has demonstrated that attackers can circumvent input validation by encoding their attack vectors, such as using base64 encoding. In response, WAF developers have incorporated normalization stages to mitigate these tactics;

3. **Input Validation.** Input validation is the phase where the actual detection of malicious input occurs. This process is heavily influenced by the specific WAF employed, but generally involves assessing incoming requests against the WAF's established policies and rules. The goal is to determine whether a request is safe to forward to the server or if it should be flagged as malicious. For instance, if a policy prohibits the inclusion of the keyword 'code' any request containing this term will be automatically blocked.

However not all implementations include the same set of stages. Some WAFs, which are old and obsolete, lack a normalization stage, for example, making them susceptible to basic encoding techniques such as base64 or HEX of the payload. Others may even not include the pre-processor stage if they are a bit less advanced.

2.1.2 WAF categorization

WAFs can be classified based on several criteria, depending on the aspects you're focusing on:

- **Stateful or Stateless** based on how they process the traffic;
- **Allowlist or Blacklist** based on how the rules for attack detection are enforced;

- **Signature-Based or Anomaly-Based**, based on their detection methodology;
- **Inline or Out-of-Band** based on their processing location;
- **Network-Based, Host-Based or Cloud-Based**, based on their deployment model.

Stateful or Stateless

A Stateless WAF does not maintain any context or memory of previous interactions. It analyzes each request independently, without considering the state of any ongoing session. Since it does not require tracking session states it is easier to implement and manage, in addition to the fact that can handle high volumes of traffic more efficiently. So Stateless WAFs are suitable for high-traffic environments where simplicity, speed, and scalability are prioritized, and where threats are primarily based on known attack patterns.

A stateful WAF maintains context about ongoing sessions and interactions between clients and servers. It tracks and remembers the state of each session, allowing it to apply security rules based on the context of the entire interaction. It improves the accuracy of blocking and it can offer more sophisticated analysis and detection of threats by understanding the flow and the state of the interactions. Unfortunately managing session state can introduce latency, particularly if the WAF is handling many simultaneous sessions, but overall the stateful WAF is the best solution for security.

Allowlist or Blocklist

In the context of a WAF, the concepts of allowlist and blocklist refer to opposite approaches to traffic filtering[2]. A blocklist approach is based on a negative security model in which all the traffic is allowed, except for the one that matches with at least one of the blocklist rules, while an allowlist approach is based on a positive security model in which all the traffic is blocked except for the one that matches with at least one of the allowlist rules. The allowlist approach is considered more secure because it minimizes the risk of unknown or unexpected threats, since only explicitly allowed traffic is permitted. The drawback is that creating and maintaining an up-to-date allowlist can be complex and time-consuming, especially as applications evolve. The blocklist approach, instead, is generally easier to configure and manage as it focuses on blocking known threats rather than defining all possible good traffic, thus it requires continuous updating of the blocklist to account for new threats and vulnerabilities. Given the advantages and disadvantages

of these two WAF approaches, it's not surprising that many WAFs now operate from a hybrid "allowlist-blocklist" security model.

Signature-Based or Anomaly-Based

Detection methodology in the context of WAFs refers to the techniques and the strategies employed to identify and mitigate malicious activities and attacks targeting web applications. The effectiveness of a WAF depends significantly on its detection methodology, which can be identified into two main approaches: Signature-Based or Anomaly-Based.[3]

Signature-based WAF rules work by scanning incoming traffic for patterns that match a predefined set of known attack signatures. These signatures typically consist of regular expressions or text strings that correspond to malicious code. When the WAF detects a match, it blocks the associated request. Signature-based rules are central to the negative model, where the WAF actively blocks traffic that matches known patterns. Ultimately, while these rules are highly efficient in preventing known attacks, as well as being relatively simple to manage and mantain, they often fall short when it comes to detecting new or unknown threats.

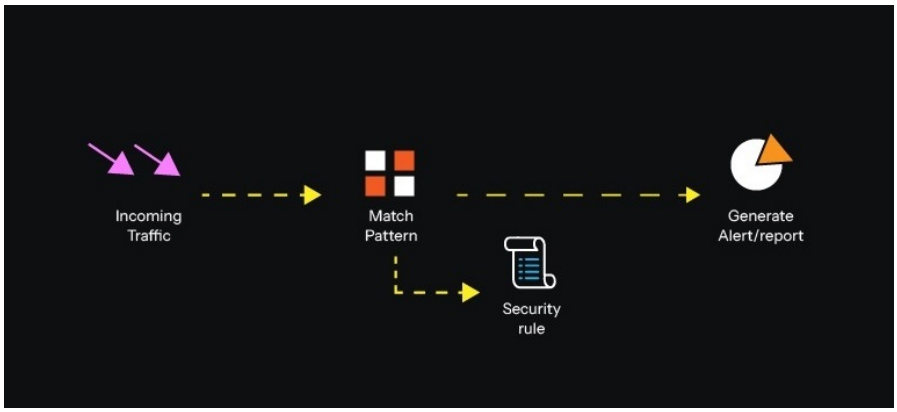

Figure 2.3: Signature-based processing

Anomaly detection WAF rules identify irregularities in traffic behavior, such as unexpected surges in activity, inputs containing abnormal characters or requests originating from unusual geographic regions. When an anomaly is flagged, the WAF may either block the request or record it for further analysis. Unlike signature-based rules, which are confined to recognizing established threats, anomaly detection can more efficiently identify new and emerging attacks. However, this approach carries a higher risk of false positives, potentially mistaking legitimate traffic for malicious activity.

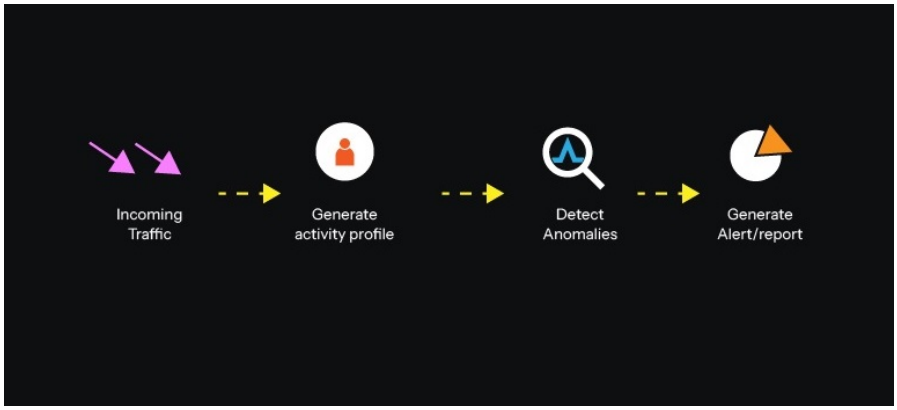

Figure 2.4: Anomaly-based processing

A highly effective strategy for safeguarding a web application or website is to combine signature-based rules with anomaly detection rules in the WAF. Signature-based rules are ideal for preventing recognized threats, while anomaly detection enhances security by identifying previously unknown attacks. This dual-layered approach ensures comprehensive protection, addressing both recognized and novel vulnerabilities.

Inline or Out-of-Band

The processing location of a WAF refers to where and how it inspects and filters incoming web traffic relative to the flow of data between the client and the web server. An Inline WAF is positioned directly in the path of network traffic between the client and the web server. This means that all incoming and outgoing traffic must pass through the WAF before it reaches the server or the client. The advantage is that as the traffic flows through the WAF, it is inspected in real-time according to the configured rules and policies and if it detects any malicious activity, it can immediately block the request, log the event, or perform other actions as configured. Instead, the disadvantage is the added latency in the traffic flow because all the traffic passes through the WAF, especially under heavy traffic loads or if complex rules are applied; furthermore the WAF become a single point of failure.

An Out-of-Band WAF, also known as an *asynchronous* or *monitoring* WAF, does not sit directly in the traffic path. Instead, it monitors the traffic by receiving a mirrored copy of the data flow, typically through a network tap or a span port, and analyzes it in parallel with the actual traffic. Given this decoupling there is no more added latency from the WAF since it does not interfere with the actual traffic flow, moreover it is no longer a single point of failure. However since it does not

operate in the main traffic path, it cannot block malicious requests in real-time, potentially allowing attacks to reach the server. That is why Out-of-Band WAFs are are primarily used for monitoring and logging.

Network-Based, Host-Based or Cloud-Based

In the context of an inline WAF, it can be placed in different spots within the network based on the type of protection desired[4].

Network-based WAFs are installed at the network boundary to protect all web applications within the network. They inspect incoming traffic and reject any that fails to comply with established security policies. Network-based WAFs are deployed either as hardware appliances or as software running on a dedicated server. The strength of this solution is that can protect multiple web applications across different web servers in the network and at the same time it is easier to manage as all traffic passes through the WAF, provided to introduce some latency and generate a single point of failure. This solution is suitable for organizations with extensive web infrastructures where centralized management and robust protection are needed.

Host-based WAFs are installed directly on individual web servers, providing protection specifically for the web applications hosted on them. They examine incoming traffic to the web application and block any that fails to adhere to the defined security policies. Host-based WAFs are deployed as software solutions that run on the web server they are installed on, which may necessitate multiple deployments for large environments, making the management complex and time-consuming. Yet they provide deep integration with the application, allowing for finely tuned security rules specific to the application's needs. This solution is suitable for organizations with a limited number of web applications where deep integration and cost-effectiveness are the key.

Cloud-based WAFs are offered as a service by a cloud provider. They sit outside the organization's infrastructure, and web traffic is routed through the provider's infrastructure for inspection and filtering before reaching the web server. Cloud-hosted WAFs are usually offered as a service, with the provider handling all the hardware and software infrastructure necessary to run the WAFs themselves. This brings a gain regarding the quickness of deployment as this model requires minimal changes to existing infrastructure and a gain regarding the scalability of traffic demands without the need for additional hardware or significant reconfiguration. Though there are limited customization options, as the service provider controls the WAF. This solution is suitable for organizations that need to scale protection quickly as their traffic and business grow.

2.2 Open Web Application Security Project

The OWASP is an open-source project, started in 2001, that aims to formulate guidelines, tools and methodologies to improve the security of web applications. Every component of a web application presents a potential vulnerability and to effectively defend against attacks, it is crucial to understand how various vulnerabilities can emerge and how malicious actors might exploit them.

2.2.1 OWASP Top 10

The OWASP Top 10 is a list of the most critical web application vulnerabilities, compiled by OWASP with input from security experts worldwide. This freely accessible document provides detailed descriptions and suggested remediation for each vulnerability on the list. The risks are prioritized based on factors such as their severity, the potential impact on an application, and the frequency with which these vulnerabilities are encountered. Since 2003, OWASP has maintained the Top 10, updating it every three to four years to reflect the evolving landscape of application security. This project is highly influential, serving as a de facto standard in web application development for many leading companies globally. Auditors may consider an organization to be behind in compliance if it fails to address the Top 10 vulnerabilities. Conversely, integrating this resource into the development lifecycle demonstrates a commitment to best practices in cybersecurity.

Categorization Approach

The identification of the top ten security risks for web applications is accomplished by following a number of distinct stages:

- **Information Gathering.** Information on vulnerabilities found in web applications are collected from a variety of organizations and sources;
- **CWE Mapping.** Each identified vulnerability is associated with a specific CWE (Common Weakness Enumeration). CWEs offer a standardized way to describe weaknesses in systems, which are conditions that could potentially lead to vulnerabilities;
- **Risk Category Grouping.** CWEs with related characteristics are organized into broader risk categories;
- **Ranking.** These risk categories are ranked according to the frequency of the vulnerabilities they contain.

The 2021 OWASP Top 10

The most recent update to the Top 10 was released in 2021, with some changes compared to the older update released in 2017, as shown in figure 2.5.

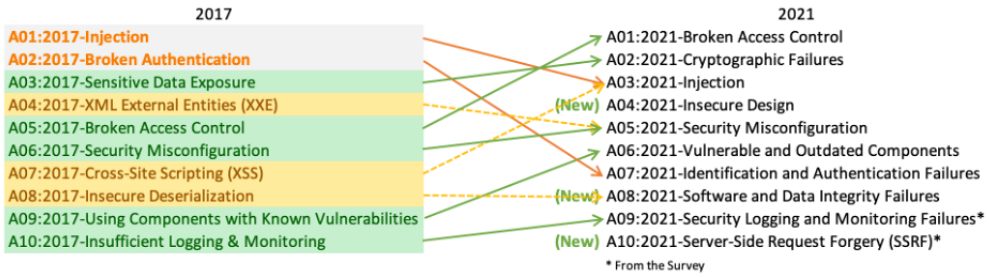

Figure 2.5: 2021 OWASP Top 10

The 2021 OWASP Top 10 categories are:

- **Broken Access Control.** Access control, also known as authorization, manages user permissions to ensure they only access resources they are allowed to. When these controls are flawed, it can lead to unauthorized users gaining access to, modifying, or deleting sensitive information. For example, an attacker might exploit such a flaw to view other users' data or gain entry to administrative sections of the application.
- **Cryptographic Failures.** Cryptography is essential for safeguarding sensitive information from unauthorized access, both when data is being transmitted and when it is stored. Vulnerabilities in cryptographic methods or their implementation can allow attackers to access confidential data, such as user passwords.
- **Injection.** Injection vulnerabilities arise when an application does not properly validate or sanitize user inputs. This oversight can allow attackers to insert malicious commands, like SQL queries (SQL injection), system commands (command injection), or code fragments (code injection), potentially compromising the system.
- **Insecure Design.** This category refers to inherent security weaknesses in the design phase of an application. Such flaws are particularly problematic because they cannot be easily remedied later in the development process. These issues often occur when the security requirements are underestimated during the design stage.
- **Security Misconfiguration.** Security misconfiguration happens when an

application is not securely configured, such as by keeping unnecessary features enabled or failing to change default passwords. When these configurations are exploited, they can become significant vulnerabilities.

- **Vulnerable and Outdated Components.** During development, applications often incorporate third-party components and frameworks. If these external components have security vulnerabilities, they can expose the entire application to risk, making it easier for attackers to exploit.
- **Identification and Authentication Failures.** Authentication mechanisms verify a user's identity. When these mechanisms are weak, attackers may be able to impersonate other users. Such vulnerabilities typically occur when there are insufficient protections against brute force attacks or when the authentication process is poorly designed or implemented.
- **Software and Data Integrity Failures.** Data integrity is vital for ensuring that information is not tampered with. When integrity is compromised, attackers can alter user data or modify the application's source code, leading to unauthorized actions or corrupted data.
- **Security Logging and Monitoring Failures.** Logs capture the actions performed within an application, and proper analysis of these logs is critical for detecting suspicious activities and identifying potential attackers. Without effective logging and monitoring, security breaches may go unnoticed.
- **Server-Side Request Forgery.** This vulnerability occurs when the URL provided by the user is not properly validated before the web application retrieves the remote resource. The vulnerability allows attackers to manipulate the application into sending requests to unintended destinations, which can result in data leaks or other malicious actions.

2.2.2 OWASP Core Rule Set

"The OWASP CRS is a set of generic attack detection rules for use with ModSecurity or compatible WAFs. It aims to protect web applications from a wide range of attacks, including the OWASP Top 10, with a minimum of false alerts."[5] In this section, we will discuss the most significant features of the CRS.

Anomaly Scoring

The key idea this mechanism exploits is the separation of the inspection and detection rules from the blocking functionality. "Anomaly scoring, also known as *collaborative detection*, is a scoring mechanism used in CRS. It assigns a numeric score to HTTP transactions (requests and responses), representing how anomalous

they appear to be. Anomaly scores can then be used to make blocking decisions."[6]

Rules designed to detect particular attack patterns and malicious behavior are executed individually. If a rule matches, it does not immediately cause any blocking action; the transaction continues. Instead, the matched rule contributes to an ongoing anomaly score for the transaction. This score increases based on the number of matched rules. Additionally, each triggered rule typically logs details about the match, such as the rule's ID, the data that triggered the rule, and the requested URI, for future reference. The blocking evaluation phase take place two times in a complete HTTP transaction:

1. **Inbound.** When all the rules examining the request data have been processed, a first blocking evaluation is conducted. If the inbound anomaly score meets or exceeds the inbound anomaly score threshold, the transaction is blocked. Transactions that do not reach this threshold proceed as usual;
2. **Outbound.** When all the rules analyzing the response data have been applied, a secondary blocking evaluation is performed. If the outbound anomaly score meets or exceeds the set outband anomaly threshold, the response is blocked from reaching the user.

By maintaining separate anomaly scores and thresholds for inbound and outbound traffic, request and response data can be inspected and scored independently. Figure 2.6 shows an example of Anomaly Scoring mechanism.

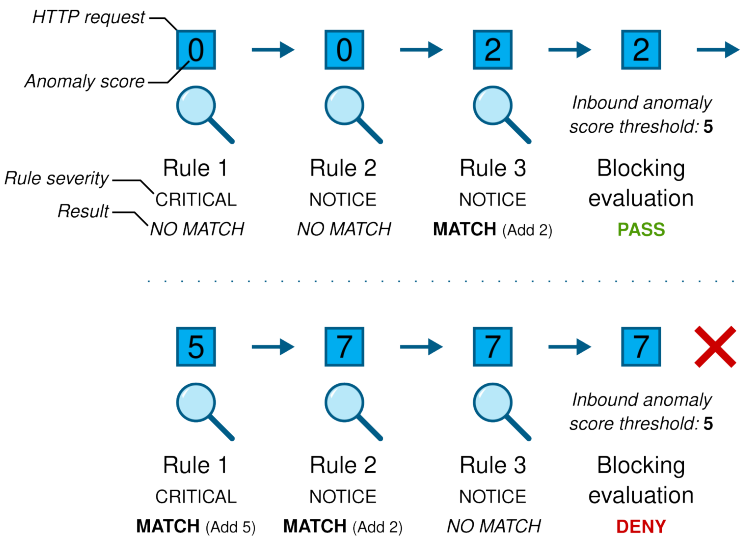

Figure 2.6: Anomaly Scoring Example

Given how this mechanism operates, it's crucial to carefully set the appropriate

values for the thresholds and properly configure the severity levels. To each CRS rule is assigned a severity level, with varying anomaly scores linked to each level. As a result, different rules can increase the anomaly score by different amounts when they are triggered. The four severity levels are:

- **Critical**, with a default anomaly scoring of 5;
- **Error**, with a default anomaly scoring of 4;
- **Warning**, with a default anomaly scoring of 3;
- **Notice**, with a default anomaly scoring of 2.

This means that when a critical severity rule is triggered, the anomaly score increases by 5 points. Modifying the default severity scores or adding new ones is generally not recommended. Lowering the scores can weaken detection capabilities, while raising them might negatively impact application performance.

Once the severity levels are defined, the anomaly score thresholds can be set. It is usually advised to configure this threshold at 5, ensuring that any transaction triggering a single critical severity rule will be blocked. Raising the threshold can reduce the system's sensitivity, possibly allowing some attacks to slip through. Conversely, lowering the threshold may increase the likelihood of blocking legitimate transactions, potentially affecting the application's normal operation.

Paranoia Levels

Another crucial aspect that requires configuration when employing the Core Rule Set is the PL (Paranoia Level), which makes it possible to define how aggressive CRS is in detecting possible attacks.[7]

In practice, the Paranoia Level dictates the number of rules that are enforced: the higher the Paranoia Level, the more rules are executed. PL1 offers a basic set of rules that rarely, if ever, trigger false alarms. PL2 builds on this by adding additional rules designed to catch more types of attacks, but with these added rules comes an increased likelihood of false alarms triggered by legitimate HTTP requests. At PL3, the rule set expands further to cover more specialized attacks, which consequently increases the occurrence of false positives. Finally, at PL4, the rules become extremely stringent, detecting nearly every possible attack, but this heightened sensitivity also results in a significant amount of legitimate traffic being incorrectly flagged as malicious.

An elevated paranoia level makes it more difficult for attackers to evade detection, yet this comes at the cost of more false positives. When false positives arise, they must be addressed through fine-tuning. This involves writing rule exclusions.

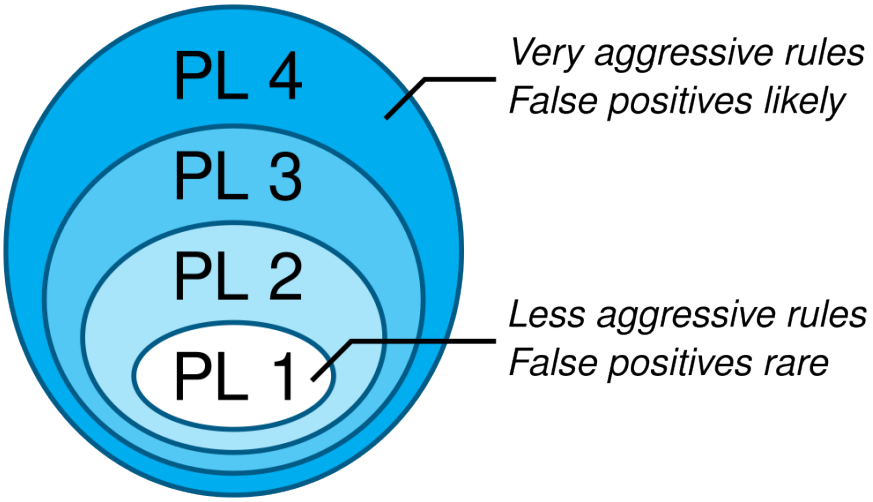

Figure 2.7: Paranoia Levels

A rule exclusion is a directive that either entirely or partially disables another rule, applying the exclusion to specific parameters or particular URIs. This approach allows the rule set to remain fully intact while ensuring that the CRS installation is no longer disrupted by false positives.

Choosing the right paranoia level is primarily influenced by the purpose of the web application and the sensitivity and quantity of the data it processes. The CRS project suggests the following guidelines:

- **PL1.** Baseline security, this is appropriate for anyone running a basic HTTP server on the internet;
- **PL2.** Adequate for applications that handle personal information, such as users' names and addresses;
- **PL3.** Suitable for applications that process sensitive data, such as bank details;
- **PL4.** Necessary only if the application manages secret or restricted data.

It's essential to recognize that Paranoia Levels and Anomaly Scoring are completely separate concepts with no direct link. The paranoia level determines the number of active rules, while the anomaly threshold dictates the number of rules that can be triggered before blocking a request.

2.3 ModSecurity

ModSecurity is an open-source Web Application Firewall that provides a robust layer of protection for web applications. It operates primarily as a module for Apache HTTP Server, but it can also be used with other web servers such as Nginx and IIS through various connectors. ModSecurity is designed to detect and mitigate a wide range of web-based attacks, including the OWASP Top 10, through the exploitation of CRS. In this section Modsecurity will be examined in depth, according to [8]. Among the most important features of ModSecurity there are:

- **HTTP Traffic Logging.** Web servers are typically proficient at logging traffic in a way that supports marketing analysis but often lack the capability to effectively log traffic to web applications, especially when it comes to capturing request bodies. Attackers exploit this weakness by using POST requests, which many systems fail to log properly, leaving them vulnerable. ModSecurity overcomes this limitation by enabling complete HTTP transaction logging, allowing full requests and responses to be recorded. It also offers fine-grained control over what is logged and when, ensuring that only relevant information is captured. Additionally, ModSecurity can be configured to mask sensitive data in certain fields before it is written to the audit log, protecting confidential information while maintaining detailed logs.
- **Real-Time Monitoring and Attack Detection.** Beyond its logging capabilities, ModSecurity can also actively monitor HTTP traffic in real-time to detect potential attacks. In this role, ModSecurity functions as a web intrusion detection system, enabling you to respond swiftly to suspicious activities occurring on your web infrastructure.
- **Flexible Rule Engine.** At the core of ModSecurity lies a versatile rule engine that drives its functionality. This engine utilizes the *ModSecurity Rule Language*, a specialized programming language tailored for handling HTTP transaction data. The Rule Language is designed to be user-friendly, making basic tasks straightforward while still offering the flexibility to perform more complex operations.
- **Flexible Deployment Mode.** ModSecurity is well-suited for deployment as both a network-based WAF or a host-based WAF. This versatility allows it to protect web applications in various environments, whether it's monitoring and filtering traffic at the network level or securing individual servers directly at the host level.
- **Attack Prevention and Virtual Patching.** ModSecurity can promptly act to prevent attacks from reaching web applications, through three possible

modus operandi:

1. **Negative\Positive security model.** ModSecurity can work both with an allowlist or a blocklist approach;
2. **Known weaknesses and vulnerabilities.** ModSecurity's rule language makes it an excellent tool for external patching. External patching, also known as *Virtual Patching*, focuses on minimizing the window of exposure. In many organizations, it can take weeks to apply patches to application vulnerabilities. With ModSecurity, applications can be protected externally, without modifying the application's source code or even accessing it, thus securing the systems until a permanent patch is implemented.

- **Portability.** ModSecurity is recognized for its compatibility with a broad spectrum of operating systems.

2.3.1 Configuration Directives

ModSecurity's configuration typically involves several files, each serving a specific purpose. These are the primary configuration files:

- **modsecurity.conf**: This is the main configuration file for ModSecurity. It contains global settings and directives that control how ModSecurity operates, such as logging, audit configurations, and core settings.
- **crs-setup.conf**: This file is used to set up and customize the OWASP Core Rule Set. It includes settings for paranoia levels, anomaly score thresholds, and other parameters specific to the CRS.
- **rules/*.conf**: These files include rules for detecting and preventing various types of attacks. The OWASP CRS, for instance, includes multiple rule files like 'REQUEST-920-PROTOCOL-ENFORCEMENT.conf', 'REQUEST-934-APPLICATION-ATTACK-GENERIC.conf' etcetera.

The activation process of the CRS is as follows[9]:

1. Download the CRS from the official website;
2. Extract the rule set files to a familiar location on the server, usally within the web server's directory;
3. Set up the main CRS configuration file, the *crs-setup.conf* file, for the settings regarding Anomaly Scoring and Paranoia Levels;
4. Specify the location of the rules to the web server by including the rules configuration files, through the command *Include "/path/to/coreruleset/*.conf"*.

In this section the main directives that can be written inside the *modsecurity.conf* file will be discussed.

SecRuleEngine

This directive provides the ability to configure the rules engine. The possible values are:

- **On.** This means the rules are processed;
- **Off.** This means the rules are not processed;
- **DetectionOnly.** This means the rules are processed but no disruptive action will be executed, so malicious transactions will be logged but not blocked.

The syntax is as follows:

```
SecRuleEngine On|Off|DetectionOnly
```

SecRequestBodyAccess and SecResponseBodyAccess

These two directives control whether ModSecurity is allowed to access the request and response bodies, respectively. The syntax is as follows:

```
SecRequestBodyAccess|SecResponseBodyAccess On|Off
```

Of course if a complete protection is wanted, it's mandatory to enable both features in order to allow ModSecurity to inspect both request and response bodies.

SecArgumentsLimit

This directive is used to define the maximum number of arguments that can be processed, through the follow syntax:

```
SecArgumentsLimit LIMIT
```

When applying this directive, it's advisable to create a rule that blocks any request with more arguments than the specified limit. Failing to do so could allow an attacker to bypass detection by placing their payload in a parameter beyond the last one analyzed.

SecDefaultAction

This directive in ModSecurity is used to define a set of default actions, in a specific phase, that will apply to all rules that follow it unless those rules explicitly specify

different actions. It simplifies rule writing by allowing the user to set common actions once, rather than having to repeat them for each individual rule. The general syntax is:

```
SecDefaultAction "action1,action2,action3..."
```

An example of usage could be:

```
SecDefaultAction "phase:2,deny,log,status:406"
```

In this example, the rule will be executed in the phase 2 and if the rule matches the transaction will be denied and logged, returning the code error 406 (*Not Acceptable*).

All rules following a SecDefaultAction directive within the same configuration context will inherit its settings unless explicitly overridden. Each SecDefaultAction directive must define a disruptive action and a processing phase; *meta-data* actions are not permitted within this directive.

SecRule

A SecRule is a directive like any other understood by ModSecurity. The difference is that this directive is way more powerful in what it is capable of representing.[10] This is arguably the most crucial directive, as it allows the user to define rules for analyzing parts of HTTP requests and responses, in this context referred to as variables, using an operator. If the conditions defined by the operator are met, specific actions can be taken. The general syntax is:

```
SecRule VARIABLES OPERATOR [ACTIONS]
```

Each rule must specify one or more variables and the operator for inspecting them. If no actions are defined, the default actions will apply. It's important to note that a default action list is always in place, even if it was not explicitly configured with SecDefaultAction. Instead, if a rule specifies actions, these will override the default actions, resulting in the final set of actions that will be applied.

An example of usage could be:

```
SecRule ARGS "<script>" "phase:1,deny,status:403,log,id
:1002,msg:'XSS Attempt Detected'"
```

In this example ARGS, which define the variable to inspect, checks if all query string and POST parameters contain the keyword '<script>'. The pattern string '<script>' is commonly used in XSS attacks. If the rule, with unique identifier 1002, detects this pattern string, it blocks the request, logs the incident with the indicated message and returns a 403 (*Forbidden*) code error.

2.3.2 Processing Phases

ModSecurity enables rules to be assigned to one of the following five phases of the HTTP request lifecycle:

- **Request Headers**;
- **Request Body**;
- **Response Headers**;
- **Response Body**;
- **Logging**.

It is crucial to keep in mind that data accumulates across phases; therefore, a rule in a particular phase will have access to all the data collected in preceding phases of the transaction. The execution phase of a rule, as we have already seen, can be specified in the actions of the designed rules. Moreover, rules are executed based on their assigned phases, so even if two rules are listed consecutively in a configuration file, they will not necessarily execute in sequence if they are set for different phases.

In this section it will be explained what happens inside each phase.

Phase 1: Request Headers

Rules in this stage are executed immediately after the request headers are received. At this point, the request body hasn't been processed yet, so not all request parameters are available. Rules should be placed in this stage if they need to be executed early, either to take action before reading the request body or to determine how the request body will be managed.

Phase 2: Request Body

This is the main phase for analyzing inputs, where most application-focused rules should be applied. At this stage, it can be certain that the request arguments are available, assuming the request body has been processed. ModSecurity supports four types of request body parsing in this phase:

- **application/x-www-form-urlencoded**;
- **multipart/form-data**;
- **XML**;
- **JSON**.

Phase 3: Response Headers

In this phase the rules are executed prior to sending back the response headers to the client. This allows for observation of the response before it is sent.

Phase 4: Response Body

This phase serves as the general-purpose output analysis stage. Here, you can apply rules to examine the response body, assuming it has been buffered. This is the appropriate phase for checking the outbound HTML for signs of information disclosure, error messages, or text indicating failed authentication.

Phase 5: Logging

This phase is executed right before logging occurs. Rules defined in this phase can only influence how the logging is performed. At this stage, it's too late to deny or block transactions. In fact, this phase is special because it is executed at the end of each transaction, regardless of the outcomes of the previous phases. This ensures that it will be processed regardless of whether the request was intercepted or if an allow action was used to permit the transaction.

2.3.3 Making the rules

As discussed, rules in ModSecurity are created using the SecRule directive, which generally is made up of 4 parts:

- **Variables.** These determine the specific parts of the request or response that the rule will inspect;
- **Operators.** These define the conditions under which the rule will be triggered;
- **Actions.** These specify what should happen when the rule's conditions are met;
- **Transformations.** These instruct ModSecurity how it should normalize variable data, they are specified in the action part of a rule.

Variables

Let's examine now some among the most relevant variables that can be used to write rules within ModSecurity.

ARGS variable is critical when inspecting an HTTP request in ModSecurity because it encompasses all the arguments, including those from a POST request payload and URL arguments. ARGS can be used with a fixed parameter to match arguments with that exact name, or with a regular expression to match any arguments whose names match the regular expression itself. Essentially, most user-controlled input and potential attack payloads are contained within this variable.

If there is need to focus the analysis on a specific subset of parameters, the user can select them by name using the syntax ***ARGS:<parameter_name>***. Conversely, if there are parameters that you want to exclude from analysis, you can do so using the syntax ***ARGS:!<parameter_name>***. However, it's essential to exercise caution when excluding parameters from the analysis, as this could inadvertently exclude critical parameters, allowing an attacker to insert a payload and potentially bypass the WAF.

FILES variable contains the names of files in a request body of type multipart/form-data. This variable is useful for monitoring file uploads, particularly for detecting attempts to exploit file upload functionality by inserting malicious code into the application. For instance, you can use the FILES variable to write rules that block file uploads with potentially dangerous extensions such as .aspx or .php. This can help prevent attackers from uploading files that could be used to compromise your system.

However, it's important to understand that the FILES variable only includes the names of the files in the request body. This means that attackers might bypass such rules by placing malicious file names in other parts of the request, not just in the multipart/form-data section. Therefore, while using the FILES variable is a useful security measure, it should be complemented with additional security practices to ensure comprehensive protection.

RESPONSE_BODY variable includes the content of the web server's response. This can be useful for detecting information disclosures or sensitive data leaks within the response content.

Still, it is essential to remember that this variable will only be populated if ModSecurity has been configured to access the response body. If ModSecurity is not set up to inspect the response body, any rules that rely on this variable will be ineffective.

REMOTE_ADDR variable holds the IP address of the client making the request. This can be useful for creating rules to block known malicious IPs or to enforce IP-based protections against brute-force attacks.

Nevertheless, it's important to keep in mind that IP addresses can be altered to bypass detection. Additionally, if ModSecurity is placed behind a reverse proxy or load balancer, REMOTE_ADDR might show the proxy's IP address rather than the client's true IP.

REQUEST_URI variable represents the URI of the HTTP request, excluding the scheme, such as *http* or *https*, host, and query string. It typically includes the path and any query parameters that are part of the request. This variable is useful for writing rules that need to inspect or act upon the request's target URI. For instance, you can create rules to block requests to certain paths or detect patterns in query parameters.

Inspecting REQUEST_URI is generally efficient as it involves only the URI part of the request. Still, complex patterns or extensive rule sets may impact performance. Furthermore, logging the variable can be useful for auditing and debugging purposes to see which URIs are being accessed or triggering rules.

Operators

After selecting the variables that the rule will apply to, the next step is to define the condition under which the rule will be activated and the associated actions will be executed. This is achieved using operators. ModSecurity offers a wide range of operators, but we will focus on the most pertinent ones for the purposes of this thesis.

@rx operator executes a match based on the regular expression pattern supplied as a parameter in order to decide if the rule should be triggered or not. It is the default operator and the most important one; if a rule does not specify a different operator, it will automatically use @rx.

Regular expressions are a powerful tool because they allow the detection of various potential attack payloads with a single rule. For instance, if we want to block the use of <code> and <script> HTML tags, which could be part of an XSS attack payload, we don't need to write two separate rules. Instead, we can use a single regular expression to match both tags.

```
SecRule VARIABLES "@rx <code>|<script>" ACTIONS
```

This example illustrates the power of regular expressions in identifying potential attack patterns. Despite their versatility, crafting regular expressions can be a challenging task, where errors are common. In the context of a WAF, such mistakes can lead to vulnerabilities and potential security breaches. Another important

point is that ModSecurity performs pattern matching in a case-sensitive manner by default. To prevent attackers from exploiting case sensitivity to bypass rules, it's highly recommended to make the regular expressions case-insensitive. This can be achieved by using the lowercase transformation function, which will be discussed later, or by adding the ***?i*** prefix directly within the pattern.

@ipMatch operator can be used in combination with the REMOTE_ADDR variable to match IP addresses against specified patterns, which can include single IP addresses, ranges of IPs, or even entire subnets. It is useful for implementing security rules based on the client's IP address, such as blocking or allowing traffic from specific IP addresses or ranges. This operator can handle both IPv4 and IPv6 addresses with the follow syntax:

```
SecRule REMOTE_ADDR "@ipMatch 192.168.1.100" ACTIONS
```

@inspectFile operator invokes an external program for each variable specified in the target list. The variable's contents are passed to the script as the first command-line argument, while the program itself must be specified as the operator initial argument. This operator is especially useful for applications with file upload features, as it allows for a thorough examination of uploaded files to assess their safety and identify any malicious content.

All types of script are supported, but the LUA script is preferred as Modsecurity has an internal LUA engine which would process internally the script. Internally processed scripts typically execute faster because they avoid the overhead of process creation and have complete access to ModSecurity's transaction context. The operator syntax is as follows:

```
SecRule FILES_TMPNAMES "@inspectFile <script_name>.<
script_extension>" ACTIONS
```

It's important to note that the @inspectFile operator should be exclusively used with the FILES_TMPNAMES variable. Using it with other variables, such as FULL_REQUEST can expose the server to code execution vulnerabilities.

Numerical operators like @lt, @le, @eq, @ge, and @gt are useful for imposing constraints based on numeric values, such as, for example, the count of HTTP request arguments. An example of rule could be:

```
SecRule &ARGS "@ge 20" ACTIONS
```

This rule would trigger actions if the number of request arguments exceeds or is equal to 50. It's important to be aware that if the specified variable cannot be

interpreted as an integer, these operators will default to treating the variable as 0. This behavior might lead to unintended issues or security vulnerabilities in the protected web application.

Actions

After selecting the variables for a rule and choosing the appropriate operator to evaluate them, the final step in crafting a ModSecurity rule is specifying the actions that will be executed when the rule is triggered. ModSecurity provides five different types of actions:

- **Disruptive actions**, which trigger ModSecurity to perform a specific response, that often involves blocking the transaction. However, not all disruptive actions result in blocking, the ALLOW action, for instance, though considered disruptive, actually permits transactions rather than obstructing them. It's crucial to remember that only one disruptive action can be applied per rule. If multiple disruptive actions are specified or inherited, only the last one will be applied. Also, it's important to keep in mind that disruptive actions will not be carried out if the SecRuleEngine is set to *DetectionOnly*.
- **Non-disruptive actions**, which instruct ModSecurity to perform a task that does not influence the flow of rule processing. For example, setting or modifying a variable is considered a non-disruptive action.
- **Flow actions**, which affect the flow of rule processing. An example could be the skip action that, if activated, will instruct ModSecurity to forego evaluating the subsequent n rules for the ongoing transaction.
- **Meta-data actions**, which are employed to offer additional details about rules, such as assigning them a severity level, an identifier, or a description to be logged when the rule is activated.
- **Data actions**, while not true actions themselves, function as containers that store data utilized by other actions, such as the status action that holds the status that will be used for blocking transactions.

Now let's examine the most relevant actions that ModSecurity offers.

allow is a disruptive action that halts rule processing upon a successful match and permits the transaction to continue. This action can offer a more detailed control over what happens, through three possible options:

1. When used alone the *allow* action impacts the entire transaction. It will stop processing in the current phase and skips all remaining phases, apart for the logging phase which is designed to always execute.

2. When used with the *phase* parameter, *allow* will stop the engine from processing any further in the current phase, but other phases will proceed as usual.
3. When paired with the *request* parameter, *allow* will stop the engine from processing the current phase, and the next phase that will be processed is the RESPONSE_HEADERS phase.

block is a disruptive action that performs the disruptive action defined by the previous SecDefaultAction. This action serves as a placeholder designed for rule creators who want to request a blocking action without defining the exact method of blocking. The purpose is to leave these decisions to the users of the rules, giving them the flexibility to specify how the blocking should be carried out or to override the blocking if they so desire.

The SecRuleUpdateActionById directive enables users to override how a rule manages blocking. This is beneficial in three scenarios:

1. If a rule has blocking hard-coded, but the user wants it to use the policy it determines.
2. If a rule is configured to block, but the user would rather have it issue a warning instead.
3. Conversely, if a rule is set to only issue a warning, but the user needs it to enforce blocking.

deny is a disruptive action used to stop the processing of a rule and to intercept the transaction. It will typically send back an HTTP status code like 403 (*Forbidden*), although this can be customized.

id is a meta-data action used in the SecRule directive to assign a unique numeric identifier to a rule. Each rule added to a ModSecurity instance must have a unique ID.[11] As a result of this the user necessitates careful planning when generating IDs to avoid conflicts with other rules. This becomes particularly important when using rule sets like the OWASP CRS. Since CRS is designed to be used alongside other rule sets, ensuring that ID ranges do not overlap between different rule sets is essential. To help manage this, the OWASP maintains a list of reserved ID ranges.

It is important to note that the ID is an identifier only; it does not affect the relative order in which rules are executed: there is a possibility that rules with inferior IDs will be processed after rules with superior IDs.

phase is a meta-data action that assigns the rule to one of the five available processing phases. Additionally, it can be used within the SecDefaultAction directive to set the default actions for rules in a specific phase.

It's important to understand that if the user assigns a rule to the wrong phase, the variable referenced in the rule might not yet be accessible. This could result in a false negative, where the rule logic is correct, but the malicious data is missed simply because the phase was incorrectly specified.

setvar is a non-disruptive action that offers the possibility to create, remove or update a variable, remembering that variable names are case-insensitive. For creating a variable and initializing it at the same time, the user can use the follow syntax:

```
setvar:tx.<attribute>=<value>
```

Transformation functions

As we observed in subsection 2.1.1, one of the key stages in the ID analysis process is normalization. In ModSecurity, this stage is accomplished using transformation functions. These functions are defined in the action portion of a rule and are structured with the following syntax:

```
SecRule VARIABLES OPERATOR "t:TRANSFORMATION_FUNCTION,
..."
```

Transformation functions are employed to modify input data prior to its use in matching, such as during operator execution. Importantly, the original input data remains unchanged. When a transformation function is requested, ModSecurity generates a copy of the data, applies the transformation to this copy, and then evaluates the operator against the transformed result.

Now let's examine the most relevant transformation functions that ModSecurity offers.

htmlEntityDecode transformation function is used to convert characters encoded as HTML entities back into their original one-byte form. This is particularly effective in defending against XSS attacks. For instance, an attacker might encode the <script> tag payload using HTML entities like this:

```
\&lt;\&\#115;\&\#99;\&\#114;\&\#105;\&\#112;\&\#116;\&gt
;
```

If the htmlEntityDecode transformation function isn't applied, the WAF might not detect this payload as malicious. When the browser interprets the content, it would convert the entities back to their original characters, allowing the script to execute.

lowercase transformation function is used to convert all characters to their lowercase equivalents. This transformation is particularly useful in scenarios where case sensitivity can lead to bypassing security rules. For instance, if a rule is designed to detect certain keywords or patterns in an HTTP request, and an attacker alters the case of these keywords, the rule might not trigger if case sensitivity is not addressed; for example if the attacker uses a payload like <ScrIpT> instead of <script>.

urlDecodeUni transformation function is used to decode encoded characters in a URL, back to their original form, including Unicode characters that may have been encoded using multiple bytes. Similar to the htmlEntityDecode function, this is vital for preventing bypasses that exploit encoding techniques to disguise malicious content. Without this function, an attacker could obscure a <script> tag by encoding it as *%3Cscript%3E*, allowing it to bypass detection mechanisms. Once the encoded payload reaches the web server, it would be decoded back to its original form, potentially leading to an attack.

utf8toUnicode transformation function is used to convert UTF-8 encoded characters into Unicode. This function is particularly useful for ensuring that all characters are properly normalized and understood in their Unicode form, which can help in detecting and mitigating attacks that exploit character encoding issues, minimizing the amount of false positives and negatives.

Multiple transformation actions can be applied to the same rule, creating a transformation pipeline. The transformations are executed sequentially, following the order they appear in the rule.

Typically, the sequence of transformations is very important. For example, when dealing with evasion tactics, performing transformations in the wrong order could allow a savvy attacker to bypass detection. Here's an illustration of a scenario where the correct sequence of transformations is fundamental to preventing evasion:

```
SecRule ARGS "@rx <script>" "id:1000, deny, t:
htmlEntityDecode, t:urlDecodeUni"
```

Furthermore, let's suppose to have the follow payload, which is the <script> payload first HTML-encoded and then URL-encoded:

```
%26lt%3Bscript%26gt%3B
```

With this payload, the transformation functions will result to be in an incorrect order, leading to the subsequent string analyzed by the WAF:

```
&lt;script&gt;
```

The rule will not detect this string, causing a security bypass.

As we approach the conclusion of this chapter, we have gained an in-depth understanding of the operational mechanics of a WAF and explored the key features for configuring both ModSecurity and the CRS. We have also observed that setting up a WAF is a complex process that demands meticulous attention and considerable time to prevent misconfigurations that could potentially be exploited. Regrettably, developers often lack either the time or expertise required for optimal WAF configuration, which can leave systems vulnerable and allow attackers to circumvent defenses and exploit application weaknesses. In the next chapter we will examine VEREFOO, the framework which is the main topic of this thesis, together the technologies it makes use of.

Chapter 3

VEREFOO

The rising incidence of cybersecurity threats highlights the urgent need to address misconfigurations in NSFs (Network Security Functions) such as firewalls and VPNs. When network administrators rely on manual methods, the allocation of filtering or protection rules across NSFs often ends up being inefficient, creating vulnerabilities within the system.

To overcome this challenge, there is a rising necessity for automated policy-based network security management tools. These tools are intended to assist human operators by streamlining the creation and configuration of security services through automation. This automated process involves setting policies for each NSF, ensuring that they meet the required security standards or intentions. The advantages of *Security Automation* are significant, including the elimination of human error, the automatic analysis of policy conflicts, and the formal verification of policy correctness.

The introduction of advanced technologies such as SDN (Software-Defined Network) and NFV (Network Function Virtualization) have elevated automation to a central role in cybersecurity. These technologies enable a more sophisticated and comprehensive application of automation, leading to the deployment of security mechanisms that are not only more resilient but also highly efficient. However, despite the promising potential, research in this field remains in its nascent stages. Recently, a range of innovative approaches have been published, exploring how these technological advancements can be strategically leveraged to enhance cybersecurity.

This chapter will first introduce the foundational technologies of SDN and NFV, which are essential for comprehending the underlying principles of VEREFOO. The work done for this thesis, as previously mentioned, aims to be a contribution

and extension of this tool. Then, with this foundational knowledge in place, the chapter will delve into the core principles and functionalities of VEREFOO.

3.1 Software Defined Network

SDN, as described in [12], represents a networking paradigm that utilizes software-based controllers or APIs to interface with the underlying hardware infrastructure and manage traffic flow across the network. In contrast to conventional architectures that depend on specialized hardware devices like routers and switches, SDN facilitates the creation and management of virtual networks and allows control over traditional hardware through software-driven methods. This paradigm introduces a centralized server to handle the routing of data packets, signifying a substantial change in how network traffic is orchestrated and managed compared to traditional approaches.

Computer networks can be divided in three planes of functionality[13]:

- **Data Plane.** This plane controls the real-time flow of network traffic, encompassing both the physical infrastructure and the virtual components involved. It relies on rules and forwarding tables to ensure efficient packet delivery to the desired destination.
- **Control Plane.** This plane is responsible for making decisions about the most efficient paths for network traffic. It encompasses sophisticated functions such as network management strategies, routing algorithms and switching mechanisms. Furthermore, it maintains continuous communication with the data plane, delivering detailed instructions on how to handle, route, and prioritize traffic effectively.
- **Management Plane.** This plane comprises software services, including tools based on the SNMP, which are utilized for remote monitoring and configuration of control functionalities.

In traditional networks, the control and data planes were tightly coupled, embedded in the same networking devices, and the whole structure was highly decentralized, reducing flexibility and increasing management complexity.

SDN creates a distinction between the data plane and the control plane, enabling a centralized controller to oversee network strategy without being directly involved in data forwarding. This feature offer potential advantages such as boosted performance, streamlined configuration, and fostering innovation in network design and operations. In fact, SDN's capability to obtain real-time network status allows for centralized control of the network, driven by both current network conditions

and user-defined policies. This real-time oversight enables the optimization of network configurations and enhances overall network performance. The advantages of SDN are further highlighted by its ability to serve as a versatile platform for experimenting with new techniques and fostering innovative network designs. This is largely due to SDN's programmability and its capability to define isolated virtual networks through the control plane.

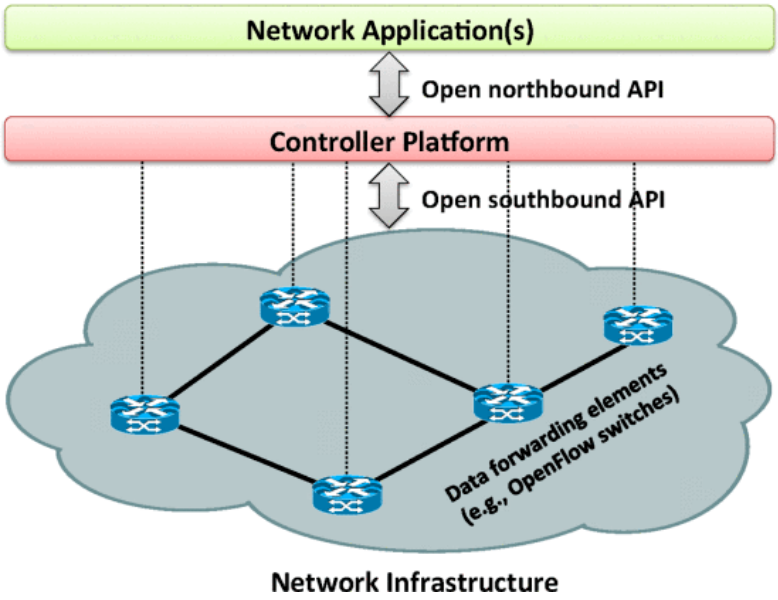

Figure 3.1: SDN architecture

Northbound and Southbound interfaces serve as the connection points between the different layers of SDN. They drive communication between the control plane and the data plane, while also bridging the control plane with external applications. These APIs empower the controller to command network devices programmatically, updating forwarding tables, directing traffic, and dynamically responding to evolving network conditions.

3.2 Network Function Virtualization

The central concept of this technology is that each function operates as a software process, eliminating the need for dedicated hardware and allowing it to run on general-purpose servers. While SDN is concerned with creating forwarding paths through software, NFV focuses on the virtualization of computing resources.

There are several ways to implement network functions in a virtualized manner. One approach is based on the use of virtual machines, which offer strong isolation.

Each time a new virtual machine is created, a separate hardware profile is defined, along with a distinct operating system that can be installed, independent of the hypervisor. As a result, this method is particularly well-suited for scenarios where safeguarding access to services is of critical importance. However, this advantage comes at the cost of significant memory and disk space requirements.

The modern approach to virtualization is based on Docker. Docker's success is largely due to its portability. Once a Docker container is created, it can easily be moved to another machine without issues. The potential problem of independent file systems consuming large amounts of disk space can be addressed through layered file systems. This type of file system allows multiple Docker containers to share parts of the file system with each other and the hypervisor. The main limitation is the lack of isolation between containers, which can lead to security vulnerabilities. For this reason, using Docker in environments where each container is owned by a different user may not be the best solution.

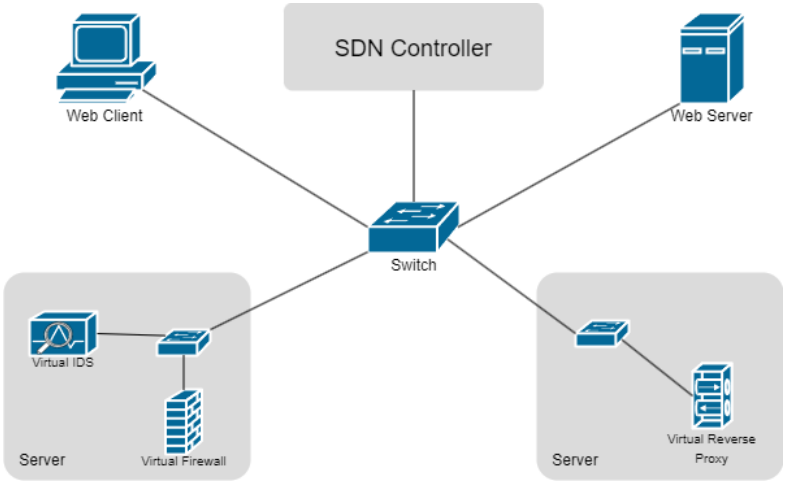

Figure 3.2: NFV technology applied to SDN architecture

NFV and SDN are intricately linked and mutually complementary technologies. NFV enhances SDN by virtualizing the SDN controller itself, enabling it to operate on cloud infrastructure. This virtualization allows for the dynamic relocation of controllers to optimal positions based on current network demands and conditions. Conversely, SDN supports NFV by offering programmable network connectivity between virtualized network functions. This programmability facilitates optimized traffic management and steering, ensuring efficient network operations and performance.[14]

The limitations of the SDN paradigm would become more evident without NFV, given that the hardware it relies on lacks adaptability and flexibility. This hardware

is designed for the static execution of specific functions. In contrast, NFV brings a transformative element by virtualizing network functions traditionally tied to hardware, effectively converting them into flexible software entities.

Ultimately, the benefits that NFV technology adds to SDN are multiples:

- There is significant flexibility in deploying new functions, as it only requires the creation of a new Virtual Machine or Docker container on the host Operating System, rather than the need to purchase and configure dedicated hardware equipment;
- It enables dynamic connectivity between functions within the same server through a software switch, which can potentially be configured as an SDN switch;
- The virtualized functions on the same server share computational resources, which can be dynamically allocated based on their specific needs;
- Forwarding rules can be easily created on the switch proactively, rather than reactively, with minimal communication between the switch and the SDN controller.

3.3 VEREFOO Framework

The primary goals of the framework known as VEREFOO (VErified REfinement and Optimized Orchestration) are to refine high-level NSRs (Network Security Requirements), optimally allocate, and automatically configure the selected NSFs (Network Security Functions) to meet security constraints. It also ensures the proper placement of each virtual network function (in this thesis, Web Application Firewalls) within the *Allocation Graph*, a topology derived from the *Service Graph* (both concepts will be explained soon).

To accomplish these outcomes, VEREFOO addresses a MaxSMT (MAX-Satisfiability modulo theories) problem using the z3 solver, a theorem prover developed by Microsoft Research, along with Verigraph, a framework created by the Polytechnic of Turin that specializes in verifying requirements for VNE (Virtual Network Embedding) scenarios.

Before exploring the details of VEREFOO, it is useful to provide the concepts of Service Graph and Allocation Graph, needed for a better comprehension of this section.

Service Graph. A Service Graph is a logical network topology composed of various interconnected network functions, such as load balancers, web caches, NATs,

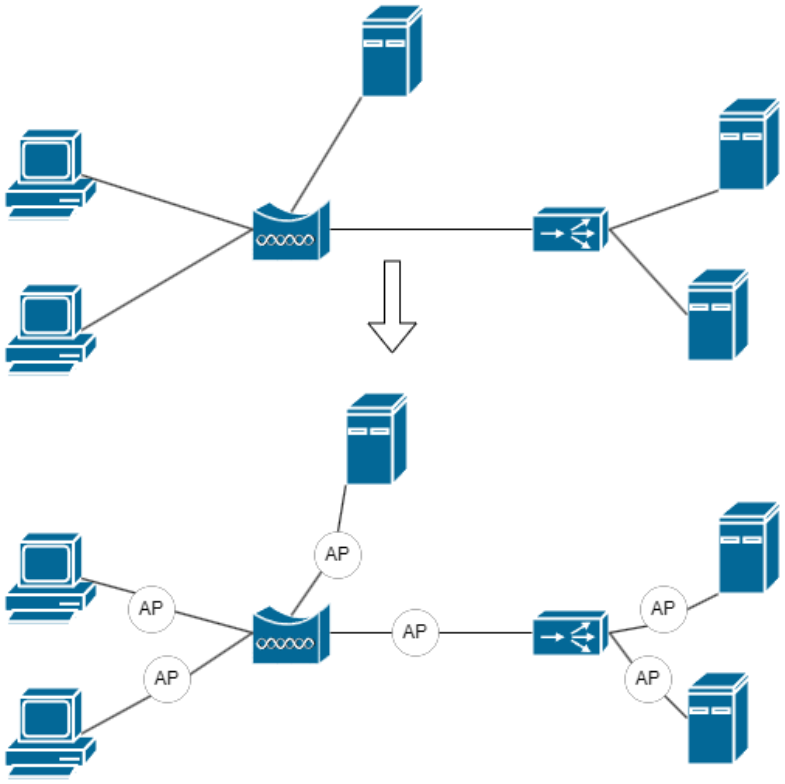

Figure 3.3: Example of Allocation Graph derived from a Service Graph

and forwarders. It is typically designed by a network architect tasked with defining a complete end-to-end service. Unlike the more linear structure of a Service Function Chain, which represents an ordered sequence of service functions and constraints applied to specific packets or flows, a Service Graph offers greater flexibility. In a Service Graph, the functions can be arranged in a more complex, non-linear structure, supporting multiple paths between endpoints and potentially incorporating loops.

It is important to note that in a Service Graph, NSFs such as firewalls are typically not included. The primary focus of the Service Graph is to deliver a fully functional network service to the user, rather than addressing security concerns.

Allocation Graph. An Allocation Graph is a logical topology that can either be built from scratch or derived from a Service Graph. While it retains the same network functions as the Service Graph, it introduces additional nodes, which are called *Allocation Places* (APs), on each link connecting consecutive nodes. These APs serve as potential spots where NSFs, such as WAFs, can be deployed. In the Allocation Graph, NSFs configuration is handled automatically. If no NSF is

placed at an AP, but the AP is involved in at least one input requirement, the AP will be filled by a forwarder. The forwarder's role is to ensure that packets are routed along their intended path without interruption.

An important point to mention is that, although the transformation from a Service Graph to an Allocation Graph is automated, the service designer retains control by imposing constraints on the process. This includes the ability to mandate the allocation of a specific NSF to a particular AP or to prohibit the placement of a new AP in certain locations. While this feature enhances flexibility and reduces computation time by narrowing the solution space, it can also result in less optimized outcomes.

The figure 3.3 shows an example of Allocation Graph derived from a Service Graph.

3.3.1 VEREFOO architecture

VEREFOO architecture, illustrated in figure 3.4 and described in [15], requires two essential input elements:

- a Service Graph, from which an Allocation Graph will be generated, or alternatively, directly an Allocation Graph through a Service GUI, which offers access to a *Network Functions Catalog*. From this catalog, the user can select which functions (either basic network functions or NSFs) to allocate directly onto their graph. This approach is useful when the user feels confident in specifying the placement of allocation points from the outset, where the functions should be installed.
- a set of NSRs to outline the security constraints that need to be fulfilled. These constraints can be defined using either a high-level or medium-level language, depending on the user's expertise. This is facilitated through a Policy GUI, which is designed to streamline and simplify the process of creating and specifying these requirements. The basic NSRs are two:
  1. **Reachability Property:** It indicates that a destination node Y MUST be reachable from a source node X along at least one path within the topology;
  2. **Isolation Property:** It specifies that a destination node Y MUST NOT be reachable from a source node X in any of the possible paths within the topology.

  The contribution of this thesis will introduce new NSRs as input of VEREFOO, as we will see later.

After receiving these inputs, VEREFOO will progress through multiple phases

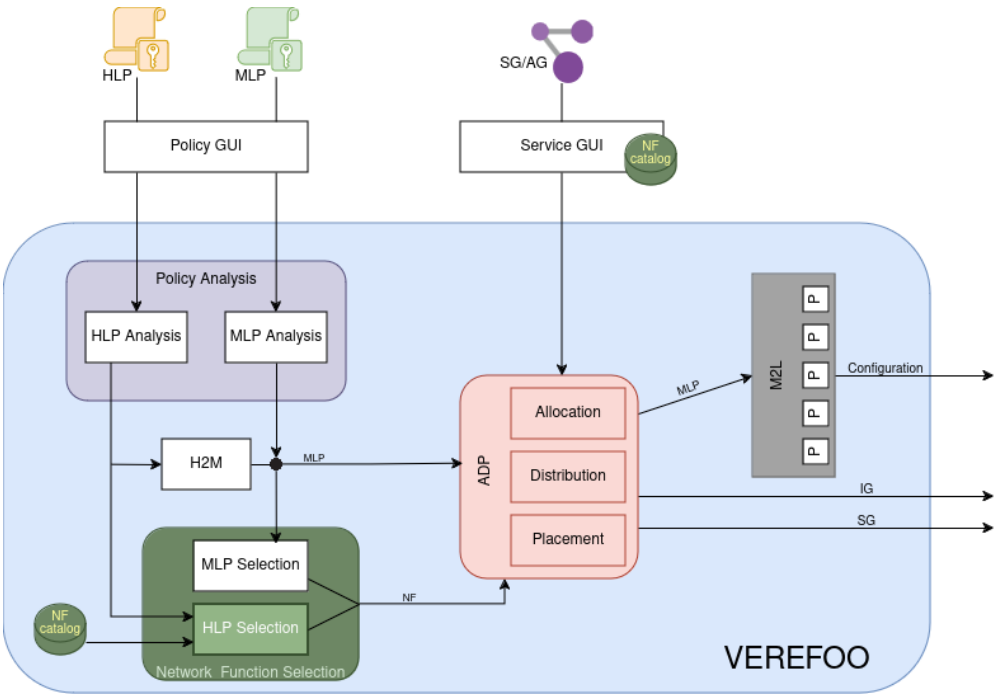

Figure 3.4: VEREFOO architecture

before arriving at the final result. A preliminary phase is represented by the *Policy Analysis* module. This module ensures the consistency and absence of conflicts among the requirements, detecting any potential errors. Once the verification is successful, it provides the minimal set of requirements that need to be fulfilled. If any errors cannot be automatically resolved, the module generates a detailed report outlining these issues.

If certain NSRs are expressed using the high-level language, these requirements are then converted into the medium-level language by the *High-To-Medium* module. The medium-level language includes all the detailed information needed for the framework to subsequently generate the corresponding NSFs that will fulfill the specified requirements.

Let's now examine in greater detail these two types of languages:

- Based on a subject-verb-object-parameters paradigm, the high-level language is used for designing and abstracting HLP (High-Level Policies). Each statement is characterized by a subject that represents the entity responsible for enforcing the NSRs, a verb that designates the required action, and a target object of the action, followed by additional arguments to provide more details. The key feature of the high-level language is its ability to formulate

simple expressions that end-users can understand and use without specialized or in-depth knowledge. The aim is to offer an optional auxiliary tool, such as a predetermined list of security policies that users can select from.

- The medium-level language is designed to provide all the necessary data for configuring network functions in the subsequent phase, as form of MLP (Medium-Level Policies). Consequently, its target audience is the technical community, such as security administrators. To avoid being tied to specific implementations, MLP must be presented in an abstract manner, independent of their particularities.

The next step involves the *Network Function Selection* Module, which is responsible for reviewing the requirements and selecting the most appropriate NSFs to meet them. These functions are chosen from the Network Function Catalog that includes all the functions managed by the framework.

The *ADP (Allocation, Distribution, and Placement)* module serves as the core component of the architecture. Its purpose is to compute a Service Graph enhanced with NSFs. The input consists of medium-level NSRs, the chosen list of NSFs, and either the original Service Graph or the pre-processed Allocation Graph. Furthermore, the output of the ADP module comprehends the list of medium-level policy rules used to configure each instance of a network function. The specific low-level configuration, which depends on the implementation of the deployed function, is then generated by the *Medium-To-Low* module. This module translates the vendor-independent expressions into the specific rules that must be applied to the appropriate network function.

3.4 Satisfiability

3.4.1 Boolean Satisfiability Problem

In the fields of logic and computer science, the SAT (Boolean Satisfiability Problem)[16] represents a foundational challenge which tries to understand if a valid interpretation can be found for a given Boolean formula. The crux of this problem lies in determining whether it is possible to assign TRUE or FALSE values to the variables in the formula in such a way that the entire expression evaluates to TRUE. If such an assignment exists, the formula is said to be satisfiable; if no such assignment can be made, it is deemed unsatisfiable. Importantly, the SAT problem is not concerned with finding the most optimal solution but merely with identifying whether there is any combination of variable assignments that renders the formula TRUE. The primary objective is simply to ascertain whether or not the formula can be satisfied.

For instance, let's consider the formula $a \wedge \neg b$. This formula is satisfiable because assigning $a = true$ and $b = false$ causes the formula to evaluate as $true$. On the other hand, consider the formula $a \wedge \neg a$. This formula is unsatisfiable because no assignment of values for a would make the formula $true$ under any circumstances, as the two conditions contradict each other.

The importance of SAT extends beyond its intrinsic complexity; it was the first problem shown to be NP-complete, a result formalized by the Cook–Levin theorem. This designation means that solving any problem within the NP class, which encompasses a vast array of optimization and decision problems, is as difficult as solving a SAT problem. Despite its critical role in computational theory, no efficient algorithm currently exists that can solve all instances of SAT problems. The search for such an algorithm remains a formidable challenge, intricately connected to the unresolved P versus NP problem, one of the most significant open questions in computer science.

3.4.2 Satisfiability Modulo Theories

SMT (Satisfiability Modulo Theories)[17] extends the concept of SAT by incorporating more complex logical constructs. While SAT deals exclusively with formulas expressed in basic Boolean logic, SMT extends basic logical formulas to include more complex domains such as integers or real numbers, and sophisticated data structures like arrays or strings. The term *modulo* refers to the fact that these formulas are interpreted with respect to a particular formal theory in first-order logic, typically without the use of quantifiers, enabling SMT to solve problems that go beyond the scope of pure Boolean logic.

SMT solvers, such as z3, are powerful tools designed to address the SMT problem for a practical range of inputs. These solvers form the foundation for a variety of applications in computer science, including automated theorem proving, program analysis, software verification, and testing. To address SMT problems, z3 leverages a specialized SMT solver that integrates search pruning techniques along with heuristic combinations of algorithmic proof methods known as *tactics*; these tactics are defined by various parameters that require fine-tuning to achieve optimal performance.

Given that SAT is already classified as NP-complete, the SMT problem, due to its more expressive and complex language, typically results in even more intricate challenges. That is why SMT problems are generally NP-hard and, in many cases, they can be undecidable. An undecidable problem is one for which it is impossible to design an algorithm that consistently produces a correct yes-or-no answer. Researchers focus on identifying specific theories or subsets of theories that render

SMT problems decidable and investigate the computational complexity of these decidable instances. The decision procedures derived from this research are often integrated into SMT solvers, enhancing their ability to handle particular problem sets.

3.4.3 Maximum Satisfiability Modulo Theories

MaxSMT (MAX-Satisfiability modulo theories) is an extension of SMT designed to handle optimization problems where determining the best possible assignment of truth values is essential for decision-making. This approach is especially valuable in fields like formal verification of hardware and software, automated theorem proving, and constraint optimization problems.

While SMT and SAT aim to check whether a logical formula can be satisfied, MaxSMT goes beyond that by seeking an assignment that maximizes the number of satisfied clauses or constraints in the problem. In a MaxSMT problem, the core elements include a set of logical constraints (often representing real-world conditions) and an objective function that defines the goal to be optimized. MaxSMT explores the space of potential solutions, not only ensuring that the logical constraints are satisfied but also maximizing the outcome defined by the objective function, leading to the most optimal configuration.

For instance, let's consider the following conjunctive formula composed of 2 variables and using the basic Boolean operators[18]:

$$(x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1)$$

This formula is intrinsically unsatisfiable. Regardless of how truth values are assigned to the two variables, at least one of the four clauses will inescapably evaluate to false. However, when framed as a MaxSMT problem, the goal shifts to maximizing the number of clauses that can be satisfied. In this case, the maximum number of claused that can be satisfied at the same time is three. Thus, satisfying three clauses for this formula is the optimal solution to the MaxSMT problem.

To enhance the flexibility of formulating a MaxSMT problem, several variants exist. The core idea is that in some scenarios, not all constraints can be met. To accommodate this, the problem is divided into two categories: hard constraints and soft constraints. Hard constraints are those that must be satisfied and represent critical requirements such as firewall placements or specific allocation restrictions set by the service designer. On the other hand, soft constraints are used to capture optimization goals and are not strictly necessary for the solution but are desirable to maximize. In VEREFOO, the solver's objective is to find a solution that meets all hard constraints while optimizing the satisfaction of the soft constraints. This

approach allows for a more flexible handling of constraints and optimization goals, ensuring that essential requirements are fulfilled while still aiming to achieve the best possible outcome for the optional constraints.

When dealing with soft constraints in a MaxSMT problem, varying weights can be assigned to different clauses to reflect the penalty associated with falsifying a clause, thereby indicating its relative importance. Introducing these weights transforms the instance into a weighted MaxSMT problem, where the constraints are classified into hard and soft, making the problem partial. In a weighted partial MaxSMT problem, the goal is to find an assignment that satisfies all the hard constraints while minimizing the total weight of any soft constraints that aren't met. This methodology is employed in VEREFOO to optimize resource utilization, specifically focusing on the total number of NSFs allocated and the rules configured, as described in [19]. This approach in VEREFOO not only facilitates automation and optimization but also ensures formal correctness. The soft constraints delineate the optimization goals, while the hard constraints represent the essential requirements. This structure allows for minimal human intervention, as the primary task for users is to provide the NSRs, with the system handling the rest.

3.5 Z3 Theorem Prover

Z3 is a cutting-edge theorem prover developed by Microsoft Research, designed to tackle SMT problems. SMT extends the SAT problem by incorporating additional theories such as equality reasoning, arithmetic, fixed-size bit-vectors, and quantifiers. This powerful tool is widely used in software analysis and verification contexts to address complex logical and computational challenges.[20] It offers APIs in different high-level programming languages, like C, C++, python and Java.

3.5.1 Z3 Architecture

The top-left of figure 3.5 provides an overview of the interfaces available for interacting with z3. Users can interface with z3 via SMT-LIB2 scripts, which serve as the standard input format for the solver. These scripts can either be provided as text files or piped directly into z3. Alternatively, one can interact with z3 through API calls in high-level programming languages (in VEREFOO, for instance, Java APIs are utilized). These high-level API calls essentially act as proxies for functions based on a C-based API, offering a flexible integration with z3 in various development environments.

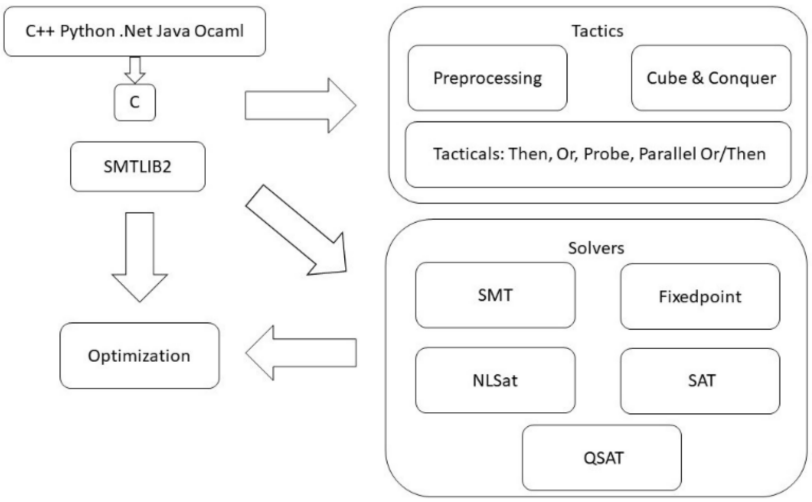

Figure 3.5: Z3 architecture

Unlike solvers that focus on verifying the satisfiability of a given set of assertions, tactics work by transforming assertions into new sets of assertions. This process generates a proof tree where nodes correspond to goals, and the branches or children of each node represent subgoals. Many valuable pre-processing steps can be expressed as tactics, which take an original goal and break it down into smaller, more manageable subgoals. This decomposition is essential for simplifying and organizing the solving process, often improving the overall efficiency of the solution strategy.

Z3 extends its capabilities beyond basic satisfiability checking by offering support for optimizing objective functions. This feature is particularly useful when the goal is to find the best solutions based on specified criteria. The optimization module allows users to define objectives, such as maximizing or minimizing a particular arithmetic term t. For example, using the command $maximize(t)$ directs the solver to seek solutions that increase the value of t. Additionally, the weighted partial MaxSMT method discussed in subsection 3.4.3, provides another way to define optimization objectives by incorporating weights for soft constraints, ensuring a balance between constraint satisfaction and optimality.

Z3 Example

Here's an example illustrating how z3 works in practice, using a trivial MaxSMT problem. Let's say we want to solve a system of linear inequalities, whose hard

constraints are:

$$x + y <= 4$$

$$x >= 0$$

$$y >= 0$$

While the soft constraints are the following:

$$x >= 2 \quad \text{with weight } 2$$

$$y >= 3 \quad \text{with weight } 5$$

We can use z3 to determine whether these inequalities are satisfiable, and if so, to find values for x and y that satisfy the hard constraints, meanwhile trying to satasfy as many soft constraints as possible. In the SMT-LIB2 format, the input for z3 would look like the listing 3.1:

```
1  (set-option :opt.priority box) ; Consider the weights
2  (set-logic QF_LIA)
3
4  ;Declare integer variables
5  (declare-fun x () Int)
6  (declare-fun y () Int)
7
8  ;Hard constraints
9  (assert (<= (+ x y) 4))
10 (assert (>= x 0))
11 (assert (>= y 0))
12
13 ;Soft constraints with weights
14 (assert-soft (>= x 2) :weight 2)
15 (assert-soft (>= y 3) :weight 5)
16
17 ;Check satisfiability and maximize soft constraint
   satisfaction
18 (check-sat)
19 (get-model)
```

Listing 3.1: Example of a SMT problem expressed in z3 language.

The output might be the one shown in listing 3.2:

```
sat
(model
  (define-fun y () Int 3)
  (define-fun x () Int 1)
)
```

Listing 3.2: Example of a SAT solution expressed in z3 language.

As we already mentioned in subsection 3.4.3, the weights help the solver prioritize which soft constraints to satisfy. The higher the weight, the more important that soft constraint is. If we invert the weights, for example, the solution that the z3 solver will provide us will be different.

This chapter explored the pivotal roles of SDN and NFV in modern network management. SDN improves control and flexibility, while NFV enables cost-effective deployment of network functions. We also introduced VEREFOO, a framework that refines NSRs, optimally allocates NSFs, and ensures their proper placement within virtualized network environments. By leveraging MaxSMT and the z3 solver, VEREFOO addresses complex security and orchestration challenges in virtual networks. In the following chapter, we will outline the primary objectives of this thesis.

Chapter 4

Thesis approach

This chapter outlines the approach adopted for the future work. The thesis objectives will be presented and explained. The final objective is broken down into a series of smaller goals that must be achieved in order to meet the overall aim.

As discussed in chapter [3], network virtualization has radically transformed the approaches used to manage and configure network security. The most prominent technologies in this field are NFV and SDN. Together, these technologies provide a high degree of flexibility and agility, as new software simplifies the tasks of network administrators.

In this context, security configurations must evolve as quickly as the network itself, making manual configuration increasingly challenging. Based on this premise, and following the analysis conducted in the previous chapters regarding web application firewalls and their configuration, the goal of this work is to design and implement a solution capable of automatically configuring and deploying WAFs within a network. Whether or not a WAF is pre-assigned, this configuration must, starting from the initial service graph, meet a set of requirements defined by the service designer.

This work specifically focuses on the defense against web-based attacks, which are becoming increasingly frequent and sophisticated. By utilizing a WAF as virtual network function, and exploiting the ModSecurity CRS, the proposed solution aims to protect web applications from a wide range of threats, including SQL injection, XSS, and other vulnerabilities exploited by attackers. The automatic configuration and deployment of WAFs across the network ensures that security policies are consistently applied, reducing the risk of misconfigurations and enhancing the overall security posture of the network. This automated approach helps to maintain up-to-date defenses in the face of evolving attack vectors, providing a

dynamic and responsive security solution.

To achieve this objective, the ADP module of the VEREFOO framework, examined in chapter [3], has been extended. The extension of the module was carried out step by step.

- The first objective was to design and implement a new type of NSR in order to allow service designers to configure OWASP rules of WAFs using a high-level or mid-level language. The corresponding outputs were also modeled and implemented accordingly. To achieve this first objective, WAFs technology has been studied, in particular it has been examined in detail ModSecurity, along with the OWASP CRS from which the model took shape.
- The second objective was to develop and implement a new feature in VEREFOO that would allow the creation of a WAF. This feature can be used to automatically assign and/or configure a WAF based on the requirements defined by the administrator. This second objective was also broken down into three smaller tasks to achieve the final goal:
  1. Manual configuration of OWASP rules;
  2. Automatic configuration and allocation of OWASP rules;
  3. Integration with the firewall implementation.

To ensure optimality in the automatic configuration and allocation of the WAF, the problem was approached by formulating it as a MaxSMT problem. The solution was obtained using the z3 solver, which allowed to meet the defined requirements while optimizing the network security configurations.

Chapter 5

Modeling of Network Security Requirements

The following chapter will provide a description of the Network Security Requirements model. NSRs represent the second input to the VEREFOO framework, in addition to the Service Graph. Specifically, NSRs define connectivity constraints between two endpoints, that can be can be classified into two different types, as already mentioned in chapter [3]:

- **Reachability Property:** It indicates that a destination node Y MUST be reachable from a source node X along at least one path within the topology;
- **Isolation Property:** It specifies that a destination node Y MUST NOT be reachable from a source node X in any of the possible paths within the topology.

The goal of this chapter is to provide a model for creating security rules that manage the category of threats regarding web-based attacks, such as SQL injection and XSS. Before moving on to the next section, which analyzes the developed model in detail, it is important to make a preliminary note. The model has been designed to be an extension of the firewall model already existing in VEREFOO. The model decouples firewall security rules from those specifically designed to counter web-based attacks, thereby rendering their application independent. It is feasible to activate solely the functionalities for defending against web attacks while deactivating the isolation and reachability constraints, or as an alternative, to enable only the the isolation and reachability constraints while leaving the web attack defenses inactive. Both decisions will lead to the placement of a WAF, if necessary to meet the NSRs.

5.1 The model

As described in previous chapters, WAFs are defined by specific parameters that can be configured to create rules. Let us now examine the characteristics that have been selected and made available to a user. Each rule $r \in R$ is represented by the following format:

[*Action, IPsrc, IPdst, PORTsrc, PORTdst, transportProto, HTTPmethod, URL, Domain*]

- **Action** $\in \{Allow, Deny\}$
  1. **Allow** corresponds to the reachability constraint. It permits the flow of network traffic from one endpoint to another;
  2. **Deny** corresponds to the isolation constraint. It blocks the flow of network traffic from one endpoint to another.
- **IPsrc** is the source IP address of the communication for which the rule is defined;
- **IPdst** is the destination IP address of the communication for which the rule is defined;
- **PORTsrc** is the source port number or range of ports for the communication for which the rule is defined;
- **PORTdst** is the destination port number or range of ports for the communication for which the rule is defined;
- **transportProto** is the transport-level protocol of the communication for which the rule is defined;
- **HTTPmethod** $\in \{GET, HEAD, DELETE, POST, PUT, PATCH\}$ is the HTTP request method used in the communication for which the rule is defined;
- **URL** is the string that represents the URL for which the rule is defined;
- **Domain** is the string that represents the domain for which the rule is defined.

For representing IP addresses, such as IPsrc and IPdst, it has been utilized the conventional dot-decimal notation:

$$ip_1.ip_2.ip_3.ip_4$$

where $ip_i, \forall i \in \{1,2,3,4\}$ can be an integer in the range from 0 to 255, inclusive, or alternatively, a wildcard symbol denoted as $*$. This symbol allows for a unified

declaration of both a network address and its corresponding netmask, instead of using two separate elements. For instance, the representation 20.0.0.∗ can be used to express endpoints within the network 20.0.0.0/24, while 30.0.∗.∗ characterizes the network 30.0.0.0/16. In the z3 context the wildcard symbol is represented by the -1 number, so, for instance, instead of 20.0.0.∗ will be used 20.0.0.-1.

The source and destination ports, PORTsrc and PORTdst, can be specified using either a single number or a range of numbers, within the interval $[0, 65535]$. To illustrate, if it is required that the source 10.0.0.1 must not be able to reach the destination 20.0.0.1 when the port numbers fall within the range $[1000, 2000]$, the traffic flow between the two endpoints must be blocked if the packets have a source port number within this interval. This ensures that the isolation requirement is met.

Finally, the transportProto element represents the layer-4 protocol used above the IP layer. The possible values are TCP and UDP, or, alternatively, the wildcard ∗. In this case, ∗ means that the property's satisfiability must be ensured, accounting for the possibility that the source may send both TCP and UDP packets.

Concerning the defense against web attacks using the OWASP CRS, in chapter [2] it was explained the activation process of owasp rules in ModSecurity. In our model each OWASP rule $o \in O$ is represented by the following format:

$$[IPsrc, IPdst, OWASPrule]$$

The user can activate one or more OWASP configuration files using the following syntax:

```
ACTIVATE <IPsrc> <IPdst> <OWASPrule>
```

where $OWASPrule \in \{$*METHOD-ENFORCEMENT*, *SCANNER-DETECTION*, *PROTOCOL-ENFORCEMENT*, *PROTOCOL-ATTACK*, *MULTIPART-ATTACK*, *APPLICATION-ATTACK-LFI*, *APPLICATION-ATTACK-RFI*, *APPLICATION-ATTACK-RCE*, *APPLICATION-ATTACK-PHP*, *APPLICATION-ATTACK-GENERIC*, *APPLICATION-ATTACK-XSS*, *APPLICATION-ATTACK-SQLI*, *APPLICATION-ATTACK-SESSION-FIXATION*, *APPLICATION-ATTACK-JAVA*, *BLOCKING-EVALUATION*, *DATA-LEAKAGES*, *DATA-LEAKAGES-SQL*, *DATA-LEAKAGES-JAVA*, *DATA-LEAKAGES-PHP*, *DATA-LEAKAGES-IIS*, *WEB-SHELLS*, *BLOCKING-EVALUATION*$\}$.

This means that if multiple configuration files need to be activated, the keyword ACTIVATE must be repeated. The following listing shows an example.

```
ACTIVATE METHOD-ENFORCEMENT 0 10.0.0.1 30.0.5.2
ACTIVATE DATA-LEAKAGES-SQL 0 10.0.0.2 30.0.5.2
```

These instructions will be first translated into the medium-level language and then, during the translation phase from the medium-level language to the low-level language, the filenames specified after the keyword ACTIVATE will be translated in the z3 language, evaluated from the z3 solver and used as configuration of the WAF.

5.2 XML representation

This section outlines how the XML model for configuring the input parameters of the framework has been updated.

5.2.1 Input schemas

In this paragraph, we analyze the XSD schema of the security requirements provided as input.

```
1  <xsd:element name="PropertyDefinition">
2   <xsd:complexType>
3    <xsd:sequence>
4     <xsd:element name="Property" type="Property"
5     minOccurs="0" maxOccurs="unbounded"/>
6     <xsd:element name="OWASPprop" minOccurs="0"
7     maxOccurs="unbounded">
8      <xsd:complexType>
9        <xsd:attribute name="value" type="OWASP"
10       use="required"/>
11       <xsd:attribute name="graph" type="xsd:long"
12       use="required"/>
13       <xsd:attribute name="src" type="xsd:string"
14       use="required"/>
15       <xsd:attribute name="dst" type="xsd:string"
16       use="required"/>
17       <xsd:attribute name="isSat" type="xsd:boolean"/>
18      </xsd:complexType>
19     </xsd:element>
20   </xsd:sequence>
21  </xsd:complexType>
22 </xsd:element>
```

Listing 5.1: Property Definition XML schema.

As we can see in the listing 5.1, *PropertyDefinition* is the main element that allows the insertion of a security requirement. It can either be of type *Property* or *OWASPprop*.

- **Property** is the element that allows for the insertion of filtering requirements;
- **OWASPprop** is the element that allows for the insertion of an OWASP rule for protection against specific web-based attacks; it contains the following attributes:
  1. **graph** is the number of the graph in which the OWASP rule is desired;
  2. **src** is the source IP address of the communication for which the OWASP rule is desired;
  3. **dst** is the destination IP address of the communication for which the OWASP rule is desired;
  4. **isSat** is a boolean value that will be true if the MaxSMT problem is satisfiable, and false otherwise;
  5. **value** is the OWASP rule that is desired to activate in the communication. The listing 5.2 provides the list of available OWASP rules, as enums.

```
<xsd:simpleType name="OWASP">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="METHOD-ENFORCEMENT"/>
  <xsd:enumeration value="SCANNER-DETECTION"/>
  <xsd:enumeration value="PROTOCOL-ENFORCEMENT"/>
  <xsd:enumeration value="PROTOCOL-ATTACK"/>
  <xsd:enumeration value="MULTIPART-ATTACK"/>
  <xsd:enumeration value="APPLICATION-ATTACK-LFI"/>
  <xsd:enumeration value="APPLICATION-ATTACK-RFI"/>
  <xsd:enumeration value="APPLICATION-ATTACK-RCE"/>
  <xsd:enumeration value="APPLICATION-ATTACK-PHP"/>
  <xsd:enumeration value="APPLICATION-ATTACK-GENERIC"/>
  <xsd:enumeration value="APPLICATION-ATTACK-XSS"/>
  <xsd:enumeration value="APPLICATION-ATTACK-SQLI"/>
  <xsd:enumeration
  value="APPLICATION-ATTACK-SESSION-FIXATION"/>
  <xsd:enumeration value="APPLICATION-ATTACK-JAVA"/>
  <xsd:enumeration value="BLOCKING-EVALUATION"/>
  <xsd:enumeration value="DATA-LEAKAGES"/>
  <xsd:enumeration value="DATA-LEAKAGES-SQL"/>
```

```
21    <xsd:enumeration value="DATA-LEAKAGES-JAVA"/>
22    <xsd:enumeration value="DATA-LEAKAGES-PHP"/>
23    <xsd:enumeration value="DATA-LEAKAGES-IIS"/>
24    <xsd:enumeration value="WEB-SHELLS"/>
25    <xsd:enumeration value="BLOCKING-EVALUATION"/>
26   </xsd:restriction>
27  </xsd:simpleType>
```

Listing 5.2: OWASP list XML schema.

As we can understand from the precedent XSD schemas, to insert OWASP rules in the form of NSRs we must instantiate an OWASPprop for each desired rule in a specific communication.

Concerning the Property element, the XSD schema is shown below:

```
1  <xsd:complexType name="Property">
2   <xsd:choice>
3    <xsd:element name="HTTPDefinition" type="HTTPDefinition"
4    minOccurs="0"/>
5    <xsd:element name="POP3Definition" type="POP3Definition"
6    minOccurs="0"/>
7   </xsd:choice>
8   <xsd:attribute name="name" type="P-Name" use="required"/>
9   <xsd:attribute name="graph" type="xsd:long"
10  use="required"/>
11  <xsd:attribute name="src" type="xsd:string"
12  use="required"/>
13  <xsd:attribute name="dst" type="xsd:string"
14  use="required"/>
15  <xsd:attribute name="lv4proto" type="L4ProtocolTypes"
16  default="ANY"/>
17  <xsd:attribute name="src_port" type="xsd:string"/>
18  <xsd:attribute name="dst_port" type="xsd:string"/>
19  <xsd:attribute name="isSat" type="xsd:boolean"/>
20  <xsd:attribute name="body" type="xsd:string"/>
21 </xsd:complexType>
22 <xsd:complexType name="HTTPDefinition">
23  <xsd:attribute name="httpmethod" type="xsd:string"/>
24  <xsd:attribute name="url" type="xsd:string"/>
25  <xsd:attribute name="domain" type="xsd:string"/>
26 </xsd:complexType>
```

Listing 5.3: Property XML schema.

The Property element as we already mentioned contains fields needed for the insertion of filtering requirements. The choice command is present in the schema, ensuring the exclusivity between *HTTPDefinition* and *POP3Definition* elements. The first is a structure that contains fields for defining web-based rule, that operates at the HTTP protocol level; the second follows the same concept as HTTPDefinition, but for the POP3 protocol. This schema has not been modified in this thesis work, therefore no further information will be provided about filtering.

5.2.2 Output schemas

Based on the inputs, the framework calculates the result, which, if necessary, will include one or more WAFs to be integrated into the network topology. In this paragraph, we will analyze the XSD schema of a Web Application Firewall.

```
<xsd:element name="firewall">
 <xsd:complexType>
  <xsd:sequence>
    <xsd:element ref="elements" minOccurs="0"
    maxOccurs="unbounded"/>
    <xsd:element ref="owasp_rules" minOccurs="0"
    maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="defaultAction" type="ActionTypes"/>
 </xsd:complexType>
</xsd:element>
```

Listing 5.4: WAF XML schema.

The *elements* definition contains all the fields described in section 5.1. A new element has been added to the schema for the management of OWASP rules, the *owasp_rules* element. The following listing shows the definition of this new element:

```
<xsd:element name="owasp_rules">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="owasp_rule" type="OWASP" minOccurs="0"
   maxOccurs="unbounded"/>
   <xsd:element name="source" type="xsd:string"/>
   <xsd:element name="destination" type="xsd:string"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:unique name="unique_owasp_constraint">
```

```
11      <xsd:selector xpath="owasp_rule"/>
12      <xsd:field xpath="."/>
13    </xsd:unique>
14  </xsd:element>
```

Listing 5.5: OWASP rules output XML schema.

As shown in listing 5.5 the *owasp_rules* element is composed of three additional nested elements:

- **owasp_rule** is the OWASP rule that has been activated in this communication. The *maxOccurs* field is set to unbounded, this means that more istances of this element can be generated, effectively making it possible to activate multiple rules for a given communication;
- **source** is the source IP address of the communication for which the OWASP rule is activated;
- **destination** is the destination IP address of the communication for which the OWASP rule is activated.

The *unique* constraint on the *owasp_rule* attribute is useful for the manual configuration of OWASP rules. It ensures that a WAF can't be configured to activate two equal rules.

5.3 MaxSMT Problem Modeling

In section 3.4 the satisfiability problems were introduced, including the MaxSMT problem. Now it will be presented the adaptation of the MaxSMT problem reported in [21] to the objectives of this thesis.

5.3.1 MaxSMT problem objectives

The MaxSMT problem, in relation to a WAF defending against web-based attacks, has two key objectives that can safeguard the network topology:

1. Minimizing the number of WAFs allocated within the Allocation Graph, ensuring optimal resource utilization when deploying NSFs that implement WAFs on the substrate network;
2. Minimizing the number of OWASP rules in each WAF policy, enabling a more readable configuration for the service designer and simplifying management or modifications, thereby reducing the risk of human error.

Although these two objectives are theoretically independent, they are intertwined

in the MaxSMT problem through shared variables, ensuring for both of them simultaneously an optimal solution. Nevertheless, greater emphasis is placed on reducing the number of deployed WAFs, as the cost of introducing extra WAF instances far outweighs the expense of configuring additional rules within an existing WAF.

5.3.2 Match function

Before showing the constraints defined in the MaxSMT problem for an OWASP requirement, it is important to introduce the match function. The match function checks whether a packet matches a specific rule. Given a generic rule r and a generic packet pk_0 in a given point of analysis, the match function returns TRUE only if all conditions outlined in the following formula are satisfied:

$$\begin{aligned} r.match(pk_0) \iff & pk_0.IPsrc \subseteq r.IPsrc \\ & \wedge\, pk_0.IPdst \subseteq r.IPdst \\ & \wedge\, pk_0.OWASPrule \subseteq r.OWASPrule \end{aligned}$$

When a WAF receives a packet, the conditions of each NSR, i.e. the OWASP rules, are applied to the packet's fields. If a rule matches the packet, it will be subsequently executed the default action, which in the case of OWASP rules it is always to discard the packet.

5.3.3 OWASP NSRs model

Let O_s be defined as the set of OWASP rules that can potentially be activated in the context of a given flow. Each $o \in O_s$ consists of the pair $(O_i,\, a)$, where O_i with $i \in [0, 22)$ represents the specific OWASP rule and a is the action to be executed, which can be *true* if the rule should be activated, or *false* if the rule should not be activated.

$$O_i = [OWASPrule_i, IPsrc, IPdst]$$
$$a \in \{true, false\}$$

Let $f = [n_s, t_{s,a}, n_a, t_{a,b}, n_b \ldots, n_k, t_{k,d}, n_d] \in F$ be a generic traffic flow, where $n_s \in N_a$ represents the source endpoint, $t_{s,a}$ represents the traffic flow between the source endpoint and the second node $n_a \in N_a$ directly connected to it, follows an intermediate ordered list of nodes and associated traffic flows to connect them, $n_d \in N_a$ represents the destination endpoint of the entire traffic flow, and $t_{k,d}$ represents the traffic flow from the penultimate node to the final endpoint.

The following conditions must be met for the flow f to satisfy requirement O_i, assuming that a is true:

(1) $\alpha(n_s) \subseteq O_i.IPsrc \wedge \alpha(n_d) \subseteq O_i.IPdst$

(2) $t_{s,a} \subseteq (O_i.IPsrc, *, *)$

(3) $t_{k,d} \subseteq (*, O_i.IPdst, O_i.OWASPrule_i)$

α is the function that, given an endpoint as a parameter, returns its IP address or the range of addresses, if a subnet is involved. Assuming this, the previous three formulas aim to:

(1) verify that the source and destination endpoints IP addresses correspond respectively to IPsrc and IPdst of the rule.

(2) and (3) verify, respectively, that the traffic flows from the source endpoint to the destination endpoint contain the specific OWASP rule.

Aggregation

As we mentioned in subsection 5.3.1, one of the objectives pursued from this thesis is the minimization of OWASP rules. To optimize the process of allocating OWASP rules, it is necessary to aggregate all flows with different source or destination addresses that have the same rules and whose sources belong to the same LCP (Longest Common Prefix), ensuring that no flow is excluded. Let's consider the basic network topology shown in figure 5.1, in which a WAF has been already allocated in the optimal AP:

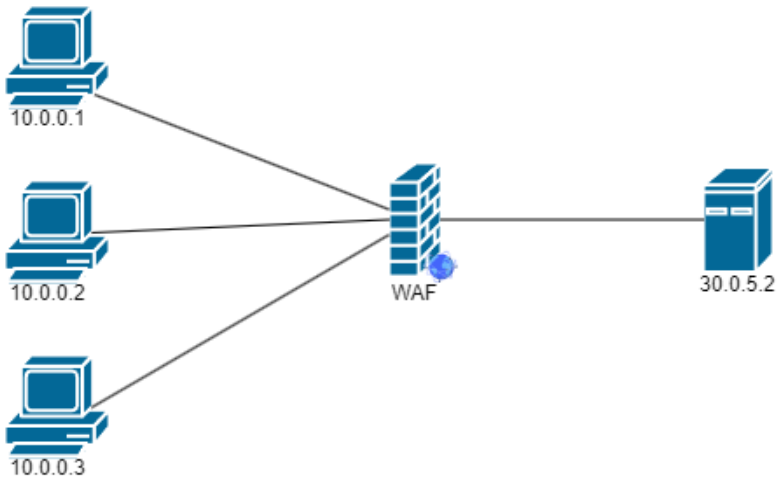

Figure 5.1: Basic network topology example

Let's suppose to have the following OWASP requirements:

```
<OWASPprop value="METHOD-ENFORCEMENT" graph="0"
    src="10.0.0.1" dst="30.0.5.2"/>
<OWASPprop value="WEB-SHELLS" graph="0"
    src="10.0.0.2" dst="30.0.5.2"/>
<OWASPprop value="WEB-SHELLS" graph="0"
    src="10.0.0.3" dst="30.0.5.2"/>
```

Listing 5.6: OWASP aggregation first example input.

There is no possibility for an aggregation in this case, therefore the output configuration of the WAF will show three distinct flows.

```
<owasp_rules>
    <owasp_rule>METHOD-ENFORCEMENT</owasp_rule>
    <source>10.0.0.1</source>
    <destination>30.0.5.2</destination>
</owasp_rules>
<owasp_rules>
    <owasp_rule>WEB-SHELLS</owasp_rule>
    <source>10.0.0.2</source>
    <destination>30.0.5.2</destination>
</owasp_rules>
<owasp_rules>
    <owasp_rule>WEB-SHELLS</owasp_rule>
    <source>10.0.0.3</source>
    <destination>30.0.5.2</destination>
</owasp_rules>
```

Listing 5.7: OWASP aggregation first example output.

It cannot be used 10.0.0.* as source for the WEB-SHELLS OWASP enforcement because it would imply web protection for the first flow also, i.e. 10.0.0.1-30.0.5.2; this would cause the WAF to consume more resources than necessary, leading to a suboptimal configuration. Now let's do another example regarding the network topology shown in figure 5.1 with different requirements:

```
<OWASPprop value="METHOD-ENFORCEMENT" graph="0"
    src="10.0.0.1" dst="30.0.5.2"/>
<OWASPprop value="MULTIPART-ATTACK" graph="0"
    src="10.0.0.1" dst="30.0.5.2"/>
<OWASPprop value="METHOD-ENFORCEMENT" graph="0"
    src="10.0.0.2" dst="30.0.5.2"/>
```

```
<OWASPprop value="MULTIPART-ATTACK" graph="0"
    src="10.0.0.2" dst="30.0.5.2"/>
<OWASPprop value="METHOD-ENFORCEMENT" graph="0"
    src="10.0.0.3" dst="30.0.5.2"/>
<OWASPprop value="MULTIPART-ATTACK" graph="0"
    src="10.0.0.3" dst="30.0.5.2"/>
<OWASPprop value="WEB-SHELLS" graph="0"
    src="10.0.0.3" dst="30.0.5.2"/>
```

Listing 5.8: OWASP aggregation second example input.

In this case an aggregation is possible, as shown in listing 5.9:

```
<owasp_rules>
    <owasp_rule>WEB-SHELLS</owasp_rule>
    <source>10.0.0.3</source>
    <destination>30.0.5.2</destination>
</owasp_rules>
<owasp_rules>
    <owasp_rule>METHOD-ENFORCEMENT</owasp_rule>
    <owasp_rule>MULTIPART-ATTACK</owasp_rule>
    <source>10.0.0.-1</source>
    <destination>30.0.5.2</destination>
</owasp_rules>
```

Listing 5.9: OWASP aggregation second example output.

This is the best result a service designer could receive, inasmuch it would be provided the best readability for this configuration, ensuring the minimum expenditure of resources with respect to the number of OWASP enforcements installed on the WAF. In order to achieve this result, a hard constraint and a soft constraint have been formulated in the MaxSMT problem.

Aggregation constraints

Given a generic set of elements $o \in O_s$, to ensure the minimal expenditure of resources, the mathematical relationship between the activated and non-activated OWASP rules, formulated as hard constraint, is the following:

$$\forall\,(O_i, a)\,, a = true \Rightarrow \bigwedge_{j \neq i} (O_j, \neg a)$$

As previously mentioned, VEREFOO includes the wildcards feature, which, in

order to optimize the readability of OWASP rules in alignment with the model proposed in this thesis, must be activated in the limited number of cases stated in this section. To achieve this goal, a soft constraint has been formulated. The constraint is of the form *Soft(f, c)*, where f is a function and c is the weight associated with it.

Let F_o be the set of flows that connect a web client or a subnet to a specific web server. For every $o \in O_s$ with $a = true$, if there is no flow $f \in F_o$ that does not contain o, then o can be aggregated in O_s and the wildcards feature exploited.

$$\forall o \in O_s : (\nexists f \in F_o : o \notin f \cap o.a = true) \Rightarrow \text{Soft}\,(aggregatable\,(o) = true, c_h)$$

With regard to the value assigned to c_h, it should coincide with the one assigned for the minimization of isolation and reachability constraints, as also in this case, the goal is of lesser importance compared to that of OWASP minimization, which will be discussed further in subsection 5.3.5.

Therefore, the usefulness of such aggregations is limited to a few specific cases, where all or most of the flows have the same OWASP rules, effectively reducing the number of *OWASP_rules* instances within the allocated WAFs. The combination of these two constraints allows us to achieve the second of our initial objectives.

Integration

The existing set of NSRs in VEREFOO, as previously mentioned, includes isolation and reachability constraints. These constraints, in the MaxSMT model, are defined as requirements r, composed of the pair (C, a), where $a \in \{Allow, Deny\}$ and $C \in \{IPsrc, IPdst, PORTsrc, PORTdst, transportProto\}$. In order to integrate web-based attacks defense functionalities via OWASP enforcements with the aforementioned constraints, the formulation of two more hard constraints was required.

For our purposes, if an isolation constraint is already in place for a given communication, with the WAF working in blocklisting mode, specifying an OWASP requirement is unnecessary. This is because packets would already be discarded at the IP level, preventing further analysis at the application level; hence, the installation of an OWASP enforcement would be redundant in such cases.

In the original model each default action for the firewalls was determined before solving the MaxSMT problem, as described in [21]. This solution would have assured the optimality as concerns the minimization of rules enforced on each firewall, inasmuch it would have limited the number of constraints generated. This

peculiarity has been exploited in our model in order to achieve our goals.

Let a_h be a generic AP in the set A_A of APs, and let us introduce the $wlst$ predicate, which is true for a_h if the WAF allocated on it will operate in allowlisting mode, false if the WAF allocated on it will operate in blocklisting mode. Let R_s and O_s be respectively the set of r and o requirements for the network; the mathematical relationship between each requirement r and each requirement o is the following:

If $wlst(a_h) = false$, then
$\forall o \in O_s : (\exists r \in R_s : O_i.IPsrc \subseteq C.IPsrc \wedge O_i.IPdst \subseteq C.IPdst) \implies o.a = false$

A slightly different observation has to be made in the case of a WAF in allowlisting mode. Installing OWASP enforcements on it would add an extra layer of defense, ensuring that even traffic from trusted sources is scrutinized for potential security threats. In fact, even if the source is trusted, it doesn't necessarily mean that its content is free of harmful exploits.

This means that in the case of a WAF in allowlisting mode, it is necessary that a reachability constraint is present for the communication on which an OWASP enforcement is desired to be installed. Without this condition, once again the OWASP enforcement would be redundant, as the packets for that communication would already be discarded at the IP layer. Therefore, the mathematical relationship between each requirement r and each requirement o become the following:

If $wlst(a_h) = true$, then
$\forall o \in O_s : (\nexists r \in R_s : O_i.IPsrc \subseteq C.IPsrc \wedge O_i.IPdst \subseteq C.IPdst) \implies o.a = false$

It is important to note that OWASP constraints are subject to a form of dependency on isolation and reachability constraints, which hold greater impact in decision-making. This dependency is essential, as isolation and reachability constraints govern packet handling at a lower level of the OSI stack, whereas OWASP constraints apply at a higher level.

5.3.4 Maximal Flows

Let $F_o \in F$ be the set of flows that satisfy $o.O_i$. The theory of maximal flows aims to improve and optimize flow management by considering only a subset of F_o, which, although smaller, is equally representative. This subset is denoted as F_o^M and includes only those flows which are not subflows of any other flow in F_o.

This approach aggregates traffic flows that exhibit the same behavior, by following identical node sequences and experiencing the same modifications, into one

cohesive maximal flow. By adopting the maximal flows, the number of distinct instances that need to be considered is reduced, and consequently, the number of constraints input into the model is also decreased. In other words, there are fewer flows to consider.

Maximal flows offer another advantage: they are generated before the MaxSMT problem is formulated. Consequently, the variables concerning the flow model, when the MaxSMT problem is formulated, are no longer free but already set to particular values. By minimizing the number of free variables, the search space for the MaxSMT problem is reduced, leading to improved performance in its resolution.

Let's consider the algorithm proposed in [21] for the computation of maximal flows and shown in figure 5.2. In this thesis work, some small adjustments have been made to integrate the maximal flows with the OWASP rules placement.

Algorithm 1. Computation of F_r^M

Input: a requirement r, and an AG G_A, **Output:** F_r^M

1: $F_r^M = \emptyset$
2: **for each** $p = [n_0, n_1, \ldots, n_{m+1}] \in paths(r, G_A)$ **do**
3: $\quad F \leftarrow \{[n_0, t^r_{0,1}, n_1, \text{true}, n_2, \ldots, \text{true}, n_{m+1}]\}$
4: $\quad$ **for** $i = 1, 2, \ldots, m$ **do**
5: $\quad\quad F \leftarrow \{l + [b_i \wedge b'_i, n_i] + l' | l + [b_i, n_i] + l' \in F, b'_i \in \{\mathcal{I}^a_i, \mathcal{I}^d_i\}\}$
6: $\quad\quad F \leftarrow \{l + [b_i \wedge b'_i, n_i] + l' | l + [b_i, n_i] + l' \in F, b'_i \in \{\mathcal{D}_{ij}\}\}$
7: $\quad\quad F \leftarrow \{l + [b_i, n_i, b_{i+1} \wedge \mathcal{T}_i(b_i), n_{i+1}] + l' | l +$
$\quad\quad [b_i, n_i, b_{i+1}, n_{i+1}] + l' \in F\}$
8: $\quad F' \leftarrow \{l + [t^r_{m,m+1} \wedge b_{m+1}, n_{m+1}] \mid l + [b_{m+1}, n_{m+1}] \in F\}$
9: $\quad$ **for** $i = m, m-1, \ldots, 1$ **do**
10: $\quad\quad F' \leftarrow \{l + [b_i \wedge \mathcal{T}_i^{-1}(b_{i+1}), n_i, b_{i+1}] + l' | l +$
$\quad\quad [b_i, n_i, b_{i+1}] + l' \in F'\}$
11: $\quad$ **if** $F \neq F'$ **then**
13: $\quad\quad F \leftarrow F'$
13: $\quad\quad$ **goto** line 4
14: $\quad F_r^M \leftarrow F_r^M \cup F$
15: **return** F_r^M

Figure 5.2: Algorithm for computation of maximal flows

The initial set of paths consists of all possible paths that have n_0 as the starting node and n_{m+1} as the final node, where n_0 has an IP address that is one of those contained in $o.O_i.IPsrc$ and n_{m+1} has an IP address that is one of those contained in $o.O_i.IPdst$.

In the computation, the fields of interest for $t^r_{0,1}$ and $t^r_{m,m+1}$ were modified:

$$t^r_{0,1} = (\alpha(n_0) \wedge O_i.IPsrc, *, *)$$
$$t^r_{0,1} = (*, \alpha(n_0) \wedge *, O_i.IPdst, O_i.OWASPrule_i)$$

5.3.5 WAF allocation

It may be necessary to allocate a WAF in a given AP solely to set one or more OWASP rules. As concerns WAF allocation, the original model has remained untouched. It is based on the following soft constraint:

$$\forall a_h \in A_A.\text{Soft}(allocated(a_h) = false, c_h)$$

where a_h is a generic AP in the set A_A of APs.

This condition allows minimizing the number of APs allocated by the framework. In fact, since a WAF can be allocated in any AP, it is necessary to introduce a condition where it is preferable that no WAF is allocated in any AP (the optimal condition).

Chapter 6

Implementation and Validation

In this chapter, the implementation of the model theorized in chapter 5 will be presented. Examples of network topologies will be shown, where both manual and automatic configuration of OWASP rules will be performed. Finally, the performance and scalability of the model will be analyzed.

6.1 Manual configuration

In the manual configuration process, the service designer is responsible for defining the security configuration of the network topology. Within this context, VEREFOO enables the verification of whether the configured topology adheres to the specified security criteria, through the insertion of NSRs. Let us consider the network topology shown in figure 6.1:

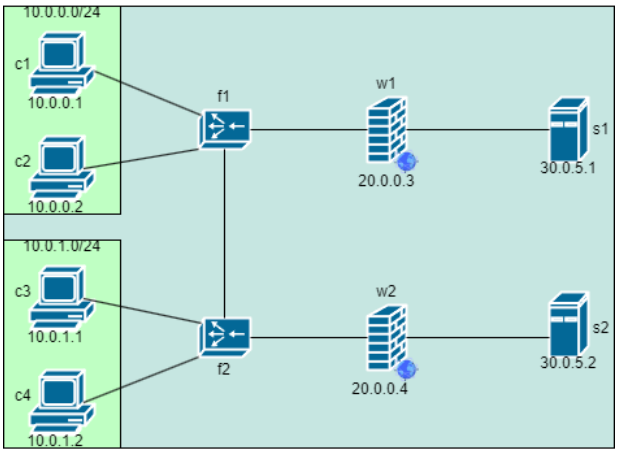

Figure 6.1: Example of network topology for manual configuration

As can be observed, the proposed topology presents four web clients (c1, c2, c3, c4) on the left, belonging to two different subnets, whose traffic is handled by two simple forwarders (f1, f2) with their respective static routes, simulating generic routers. In addition to the web clients, there are two web servers (s1, s2) on the right and two WAFs (w1, w2), already deployed by the service designer in the optimal locations. These WAFs are responsible for analyzing the packets originating from the web clients and destined to the web servers. The table 6.1 presents a definition of each node, along with its IP address as utilized within the VEREFOO environment, and its respective role within the network topology.

Name	IP Address	Role
c1	10.0.0.1	Web Client in 10.0.0.0/24
c2	10.0.0.2	Web Client in 10.0.0.0/24
c3	10.0.1.1	Web Client in 10.0.1.0/24
c4	10.0.1.2	Web Client in 10.0.1.0/24
f1	20.0.0.1	Forwarder
f2	20.0.0.2	Forwarder
w1	20.0.0.3	Web Application Firewall
w2	20.0.0.4	Web Application Firewall
s1	30.0.5.1	Web Server
s2	30.0.5.2	Web Server

Table 6.1: Newtork nodes for manual configuration example.

Now, let us assume that the service designer wants to verify that the configured WAFs comply with the NSRs of table 6.2.

Policy	IPsrc	IPdst	Psrc	Pdst	tProto
Isolation	10.0.0.1	30.0.5.2	*	*	ANY
PROTOCOL-ENFORCEMENT	10.0.0.1	30.0.5.1	//	//	//
METHOD-ENFORCEMENT	10.0.0.1	30.0.5.1	//	//	//
APPLICATION-ATTACK-JAVA	10.0.0.2	30.0.5.1	//	//	//
APPLICATION-ATTACK-XSS	10.0.1.1	30.0.5.1	//	//	//
DATA-LEAKAGES-IIS	10.0.1.1	30.0.5.1	//	//	//
SCANNER-DETECTION	10.0.1.1	30.0.5.2	//	//	//
MULTIPART-ATTACK	10.0.1.2	30.0.5.2	//	//	//
WEB-SHELLS	10.0.1.2	30.0.5.2	//	//	//
BLOCKING-EVALUATION	10.0.1.2	30.0.5.2	//	//	//

Table 6.2: NSRs for manual configuration example.

These requirements include an isolation property and various OWASP properties

between web clients and web servers. After providing the NSRs as input along with the network topology, VEREFOO will formulate a MaxSMT problem and verify that the manually allocated WAFs shown in figure 6.2 meet the security requirements. If the verification is successful, the output will appear as shown in figure 6.3, meaning that all requirements are satisfied.

```
<node name="20.0.0.3" functional_type="FIREWALL">
    <neighbour name="20.0.0.1"/>
    <neighbour name="30.0.5.1"/>
    <configuration name="conf1" description="A simple description">
        <firewall defaultAction="ALLOW">
            <owasp_rules>
                <owasp_rule>PROTOCOL-ENFORCEMENT</owasp_rule>
                <owasp_rule>METHOD-ENFORCEMENT</owasp_rule>
                <source>10.0.0.1</source>
                <destination>30.0.5.1</destination>
            </owasp_rules>
            <owasp_rules>
                <owasp_rule>APPLICATION-ATTACK-JAVA</owasp_rule>
                <source>10.0.0.2</source>
                <destination>30.0.5.1</destination>
            </owasp_rules>
            <owasp_rules>
                <owasp_rule>APPLICATION-ATTACK-XSS</owasp_rule>
                <owasp_rule>DATA-LEAKAGES-IIS</owasp_rule>
                <source>10.0.1.1</source>
                <destination>30.0.5.1</destination>
            </owasp_rules>
        </firewall>
    </configuration>
</node>
<node name="20.0.0.4" functional_type="FIREWALL">
    <neighbour name="20.0.0.2"/>
    <neighbour name="30.0.5.2"/>
    <configuration name="conf1" description="A simple description">
        <firewall defaultAction="ALLOW">
            <elements>
                <action>DENY</action>
                <source>10.0.0.1</source>
                <destination>30.0.5.2</destination>
                <protocol>ANY</protocol>
                <src_port>*</src_port>
                <dst_port>*</dst_port>
            </elements>
            <owasp_rules>
                <owasp_rule>SCANNER-DETECTION</owasp_rule>
                <owasp_rule>MULTIPART-ATTACK</owasp_rule>
                <source>10.0.1.1</source>
                <destination>30.0.5.2</destination>
            </owasp_rules>
            <owasp_rules>
                <owasp_rule>MULTIPART-ATTACK</owasp_rule>
                <owasp_rule>WEB-SHELLS</owasp_rule>
                <owasp_rule>BLOCKING-EVALUATION</owasp_rule>
                <source>10.0.1.2</source>
                <destination>30.0.5.2</destination>
            </owasp_rules>
        </firewall>
    </configuration>
</node>
```

Figure 6.2: WAFs manual configuration

```
<PropertyDefinition>
    <Property name="IsolationProperty" graph="0" src="10.0.0.1" dst="30.0.5.2" lv4proto="ANY" src_port="null" dst_port="null" isSat="true"/>
    <OWASPprop value="PROTOCOL-ENFORCEMENT" graph="0" src="10.0.0.1" dst="30.0.5.1" isSat="true"/>
    <OWASPprop value="METHOD-ENFORCEMENT" graph="0" src="10.0.0.1" dst="30.0.5.1" isSat="true"/>
    <OWASPprop value="APPLICATION-ATTACK-JAVA" graph="0" src="10.0.0.2" dst="30.0.5.1" isSat="true"/>
    <OWASPprop value="APPLICATION-ATTACK-XSS" graph="0" src="10.0.1.1" dst="30.0.5.1" isSat="true"/>
    <OWASPprop value="DATA-LEAKAGES-IIS" graph="0" src="10.0.1.1" dst="30.0.5.1" isSat="true"/>
    <OWASPprop value="SCANNER-DETECTION" graph="0" src="10.0.1.1" dst="30.0.5.2" isSat="true"/>
    <OWASPprop value="MULTIPART-ATTACK" graph="0" src="10.0.1.2" dst="30.0.5.2" isSat="true"/>
    <OWASPprop value="WEB-SHELLS" graph="0" src="10.0.1.2" dst="30.0.5.2" isSat="true"/>
    <OWASPprop value="BLOCKING-EVALUATION" graph="0" src="10.0.1.2" dst="30.0.5.2" isSat="true"/>
</PropertyDefinition>
```

Figure 6.3: WAFs manual configuration output

As we can further examine from figure 6.2, WAF w2 has an additional OWASP rule for the c3/s2 communication, the MULTIPART-ATTACK enforcement. This does not preclude the solution from being satisfiable, as the key factor is that all the rules within the requirements are satisfied. Naturally, the reverse is not true; if even a single OWASP rule included in the NSRs is missing from the WAF manual configuration, the solution would become unsatisfiable.

6.2 Automatic configuration

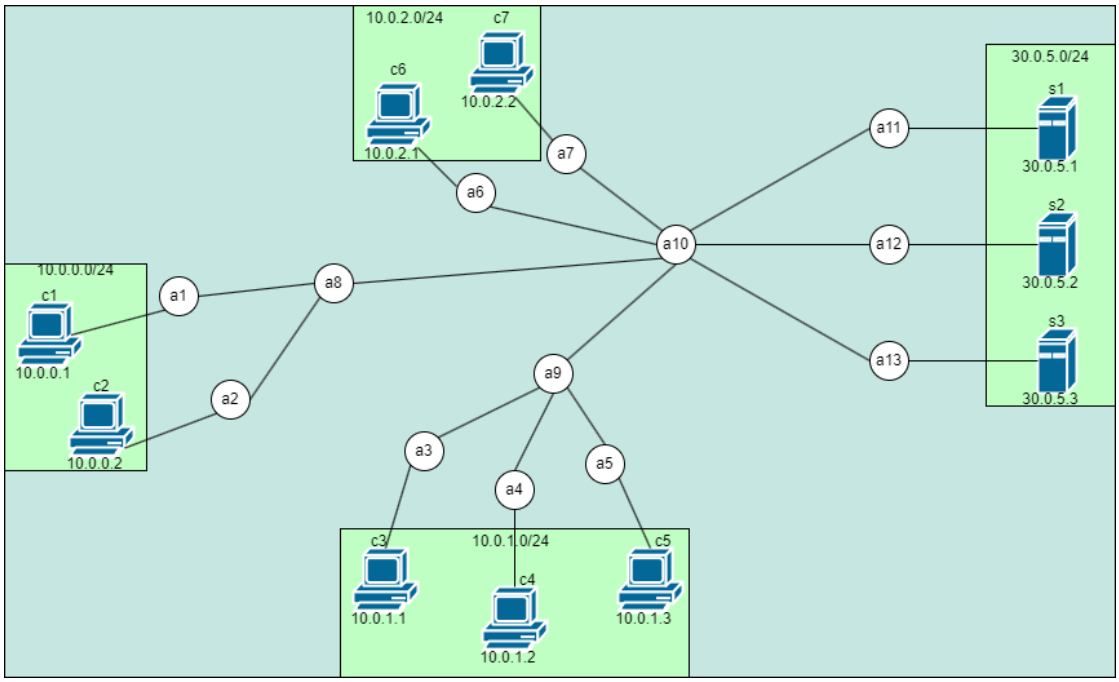

Figure 6.4: Example network topology AG for automatic configuration

The automatic configuration, in contrast to the manual one, allows the automatic allocation of WAFs within the provided network topology, configuring these

WAFs with the OWASP rules specified in the requirements, as well as the isolation and reachability constraints inherent to the firewall, which have been successfully integrated into the implementation developed in this thesis work. In accordance with the MaxSMT model formulated in chapter 5, the minimum number of WAFs will always be allocated, along with the minimum number of rules, which will be aggregated by LCP to enhance the overall readability of the configuration.

For this test, a more complex network topology was examined compared to the one used for manual configuration, which includes a higher number of web clients, APs and web servers. The aforementioned topology, represented by the allocation graph in figure 6.4, includes seven web clients (c1, c2, c3, c4, c5, c6, c7) belonging to three different subnets, thirteen APs (a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13), and three web servers (s1, s2, s3). The table 6.3 presents a definition of each node, along with its IP address as utilized within the VEREFOO environment, and its respective role within the network topology.

Name	IP Address	Role
c1	10.0.0.1	Web Client in 10.0.0.0/24
c2	10.0.0.2	Web Client in 10.0.0.0/24
c3	10.0.1.1	Web Client in 10.0.1.0/24
c4	10.0.1.2	Web Client in 10.0.1.0/24
c5	10.0.1.3	Web Client in 10.0.1.0/24
c6	10.0.2.1	Web Client in 10.0.2.0/24
c7	10.0.2.2	Web Client in 10.0.2.0/24
a1...13	20.0.0.1...13	Allocation Place
s1	30.0.5.1	Web Server in 30.0.5.0/24
s2	30.0.5.2	Web Server in 30.0.5.0/24
s3	30.0.5.3	Web Server in 30.0.5.0/24

Table 6.3: Newtork nodes for automatic configuration example.

The primary objective in this network is to protect the three web servers from cyberattacks originating from web clients belonging to different subnets. Specifically, several requirements have been defined, as shown in table 6.4, requiring reachability from whole network to server s1 and from the subnet below (c3,c4,c5) to server s3, besides isolation from c6 to s2. In addition to these, various OWASP requirements have been defined for different client-server communications.

After computing the requirements, VEREFOO will provide the Service Graph of the new network topology. As we can see in figure 6.5, a single WAF has been allocated in what was previously AP a10, along with two forwarders for packets routing; the automatically generated configuration of this WAF, is instead shown in figure 6.6.

Policy	IPsrc	IPdst	Psrc	Pdst	tProto
Isolation	10.0.2.1	30.0.5.2	*	*	ANY
Reachability	10.0.1.-1	30.0.5.3	*	*	ANY
Reachability	10.0.-1.-1	30.0.5.1	*	*	ANY
PROTOCOL-ENFORCEMENT	10.0.0.-1	30.0.5.1	//	//	//
DATA-LEAKAGES-SQL	10.0.0.-1	30.0.5.1	//	//	//
WEB-SHELLS	10.0.0.-1	30.0.5.1	//	//	//
DATA-LEAKAGES-PHP	10.0.1.1	30.0.5.2	//	//	//
DATA-LEAKAGES-PHP	10.0.1.2	30.0.5.2	//	//	//
DATA-LEAKAGES-PHP	10.0.1.3	30.0.5.2	//	//	//
BLOCKING-EVALUATION	10.0.0.2	30.0.5.3	//	//	//
PROTOCOL-ATTACK	10.0.1.2	30.0.5.3	//	//	//
MULTIPART-ATTACK	10.0.2.1	30.0.5.3	//	//	//
APPLICATION-ATTACK-LFI	10.0.2.2	30.0.5.2	//	//	//
APPLICATION-ATTACK-XSS	10.0.2.2	30.0.5.2	//	//	//
APPLICATION-ATTACK-RCE	10.0.2.2	30.0.5.2	//	//	//
APPLICATION-ATTACK-SQLI	10.0.2.2	30.0.5.2	//	//	//
METHOD-ENFORCEMENT	10.0.2.1	30.0.5.2	//	//	//
SCANNER-DETECTION	10.0.2.1	30.0.5.2	//	//	//

Table 6.4: NSRs for automatic configuration example.

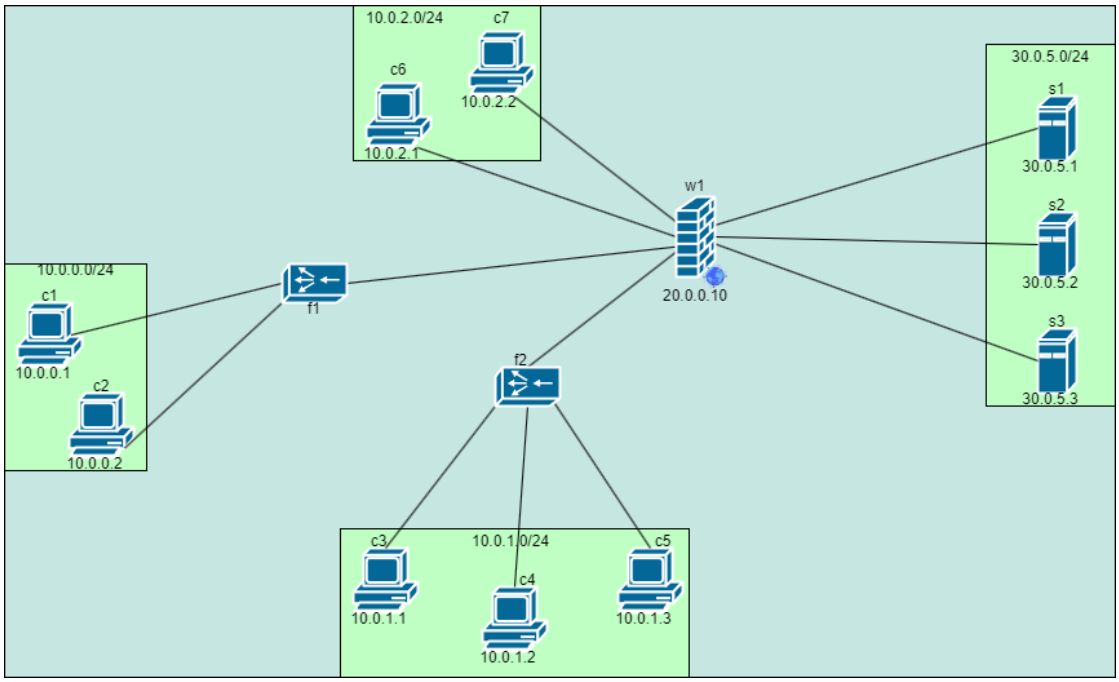

Figure 6.5: Example network topology SG for automatic configuration

```
<configuration name="AutoConf">
    <firewall defaultAction="ALLOW">
        <elements>
            <action>DENY</action>
            <source>10.0.2.1</source>
            <destination>30.0.5.2</destination>
            <protocol>ANY</protocol>
            <src_port>*</src_port>
            <dst_port>*</dst_port>
        </elements>
        <owasp_rules>
            <owasp_rule>PROTOCOL-ENFORCEMENT</owasp_rule>
            <owasp_rule>DATA-LEAKAGES-SQL</owasp_rule>
            <owasp_rule>WEB-SHELLS</owasp_rule>
            <source>10.0.0.-1</source>
            <destination>30.0.5.1</destination>
        </owasp_rules>
        <owasp_rules>
            <owasp_rule>APPLICATION-ATTACK-LFI</owasp_rule>
            <owasp_rule>APPLICATION-ATTACK-RCE</owasp_rule>
            <owasp_rule>APPLICATION-ATTACK-SQLI</owasp_rule>
            <owasp_rule>APPLICATION-ATTACK-XSS</owasp_rule>
            <source>10.0.2.2</source>
            <destination>30.0.5.2</destination>
        </owasp_rules>
        <owasp_rules>
            <owasp_rule>DATA-LEAKAGES-PHP</owasp_rule>
            <source>10.0.1.-1</source>
            <destination>30.0.5.2</destination>
        </owasp_rules>
        <owasp_rules>
            <owasp_rule>PROTOCOL-ATTACK</owasp_rule>
            <source>10.0.1.2</source>
            <destination>30.0.5.3</destination>
        </owasp_rules>
        <owasp_rules>
            <owasp_rule>MULTIPART-ATTACK</owasp_rule>
            <source>10.0.2.1</source>
            <destination>30.0.5.3</destination>
        </owasp_rules>
        <owasp_rules>
            <owasp_rule>BLOCKING-EVALUATION</owasp_rule>
            <source>10.0.0.2</source>
            <destination>30.0.5.3</destination>
        </owasp_rules>
    </firewall>
</configuration>
```

Figure 6.6: Example of WAF automatic configuration output

As can be seen from figure 6.6, the WAF is set on blocklisting mode, that is because there are more reachability than isolation constraints. Had the requirements included only OWASP constraints, the default action would have been set to allow by default. All the requirements are satisfied as concerns both reachability/isolation constraints and OWASP enforcements; furthermore, the best readability is ensured, as all subnets have been aggregated where needed.

As we could expect, METHOD-ENFORCEMENT and SCANNER-DETECTION rules are missing from the configuration, as there is a deny action from c6 to s2 which would make the rules redundant.

6.3 Performance and scalability

This section presents the results of the performance and scalability evaluations conducted on the developed feature of the framework. The aim is to demonstrate the objectives that have been met and to identify areas for improvement to address any existing limitations in future iterations.

All the tests have been carried out on a machine equipped with an Intel Core i7 at 1.80-1.99 GHz and 8 GB of RAM.

Three types of tests have been performed:

- Fixed number of NSRs with increasing number of APs;
- Fixed number of APs with increasing number of NSRs;
- Increasing number of APs and NSRs.

In the first case, shown in figure 6.7, the evolution of computational time is analyzed in relation to the increasing number of APs, while maintaining a fixed number of 10 NSRs. As can be observed, the time increases linearly with the number of APs, reaching approximately 6.5 seconds in the worst case.

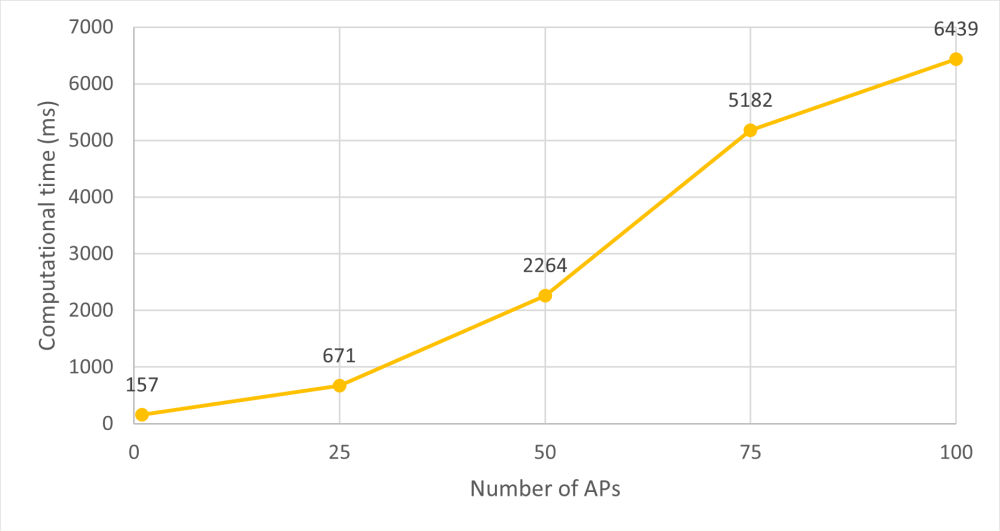

Figure 6.7: Results of scalability tests for APs

In the second case, shown in figure 6.8, the evolution of computational time

is analyzed in relation to the increasing number of NSRs, while maintaining a fixed number of 10 APs. As can be observed, the maximum computational time reached is approximately half second. This result highlights the high scalability of the OWASP requirements, as could be inferred from the lightweight and optimized MaxSMT model.

In fact, the OWASP rules are aggregated based on source-destination pairs. This means that if there are ten OWASP requirements that require protection from 10.0.0.1 to 30.0.0.1, for example, only a single packet will be generated, which wiil flow across all the possible paths. Once the satisfiability of that packet is verified, all ten enforcements will be applied to it.

In this specific test, efforts were made to achieve the greatest possible uniformity in the diversification of source-destination pairs and OWASP rules alone, in order to accurately simulate the needs of a real service designer.

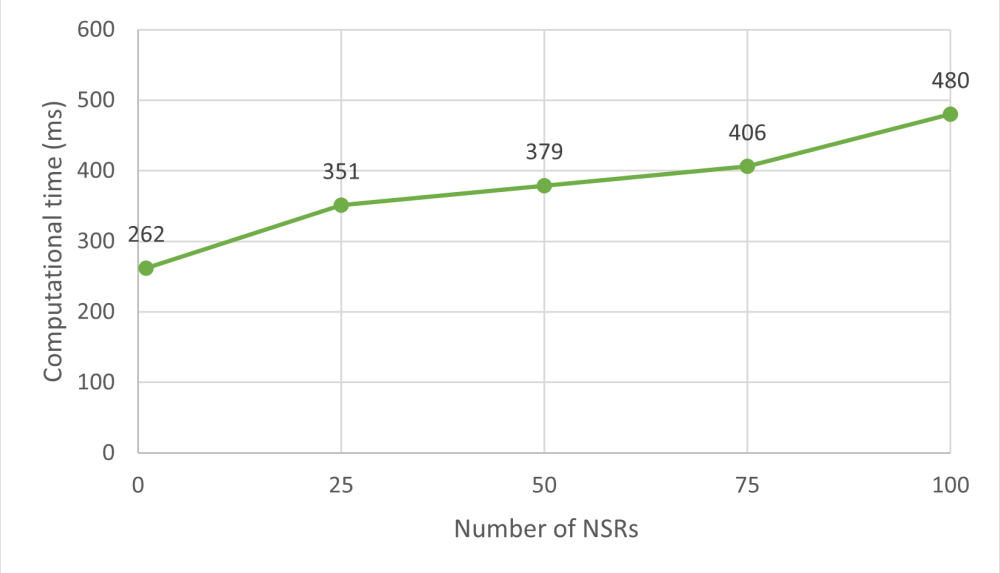

Figure 6.8: Results of scalability tests for NSRs

The primary impact on computational time is therefore due to the complexity of the network topology for this feature. To confirm this, a third and final test was conducted, shown in figure 6.9, where both the number of APs and NSRs increase proportionally.

The results of the third test, in fact, do not differ significantly from those of the first test, with only about a one-second difference in the worst-case scenario, i.e., 100 APs and 100 NSRs. Ultimately, the OWASP enforcements feature demonstrates high scalability and optimal performance.

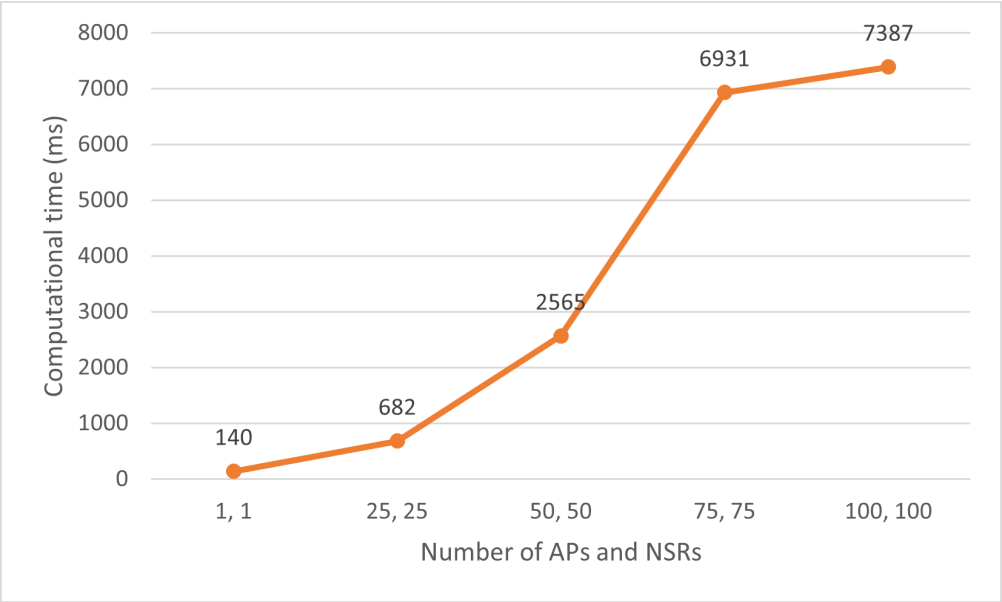

Figure 6.9: Results of scalability tests for APs and NSRs proportionally increased

Chapter 7

Conclusions

During the course of this thesis, enhancements to the ADP module of VEREFOO were developed and designed to extend the capabilities of the framework. This work was carried out with the future goal of utilizing the developed framework in real-world contexts where automation of security for web-based attacks is required.

Initially, a comprehensive study was conducted on the functioning of Web Application Firewalls, with a particular focus on ModSecurity, an open-source Web Application Firewall with significant capabilities. Based on the rule model employed by ModSecurity to ensure security against web-based attacks in compliance with the OWASP Top Ten, a MaxSMT problem was formulated to facilitate the integration of VEREFOO with the Core Rule Set.

Subsequently, the model was successfully implemented for both manual and automatic configurations. In addition, the implementation was integrated with the existing firewall module, allowing both implementations to coexist, thanks to new dependency constraints between them which were formulated.

After the aforementioned tasks were completed, a series of performance and scalability tests were conducted to understand the differences in computation time across various operational and network conditions. The results indicate that the computation time for the implemented feature has proven to be encouraging and consistent with the proposed model, confirming the optimality of the solution.

In the future, to enhance the work presented in this thesis, the implementation of the Web Application Firewall should be completed, specifically concerning the filtering of URLs and domains. Additionally, the framework could be further enriched with new virtual network functions.

Bibliography

[1] Medium. *WAF: Web Application Firewalls — How do they even work?* URL: https://medium.com/codex/waf-web-application-firewalls-3373d520385f (visited on 08/16/2024).

[2] Paloaltonetworks. *What Is a WAF? | Web Application Firewall Explained.* URL: https://www.paloaltonetworks.com/cyberpedia/what-is-a-web-application-firewall (visited on 08/16/2024).

[3] Modshieldsb. *Demystifying WAF Rules: Signature-based vs. Anomaly Detection.* URL: https://www.modshieldsb.com/demystifying-waf-rules-signature-based-vs-anomaly-detection/ (visited on 08/16/2024).

[4] Hackerone. *Web Application Firewall: 3 Types of WAF and Key Capabilities.* URL: https://www.hackerone.com/knowledge-center/web-application-firewall (visited on 08/16/2024).

[5] OWASP Foundation. *OWASP CRS Project.* URL: https://coreruleset.org/ (visited on 08/16/2024).

[6] OWASP Foundation. *Anomaly Scoring.* URL: https://coreruleset.org/docs/concepts/anomaly_scoring/ (visited on 08/16/2024).

[7] OWASP Foundation. *Paranoia Levels.* URL: https://coreruleset.org/docs/concepts/paranoia_levels/ (visited on 08/16/2024).

[8] SpiderLabs. *ModSecurity Reference Manual.* URL: https://github.com/owasp-modsecurity/ModSecurity/wiki/Reference-Manual-(v3.x) (visited on 08/21/2024).

[9] OWASP Foundation. *Installing CRS.* URL: https://coreruleset.org/docs/deployment/install/ (visited on 08/21/2024).

[10] OWASP Foundation. *Making Rules.* URL: https://coreruleset.org/docs/rules/creating/ (visited on 08/30/2024).

[11] OWASP Foundation. *Rule IDs.* URL: https://coreruleset.org/docs/rules/ruleid/ (visited on 09/01/2024).

[12] Wenfeng Xia et al. "A Survey on Software-Defined Networking". In: *IEEE Communications Surveys & Tutorials* 17.1 (2015), pp. 27–51. DOI: 10.1109/COMST.2014.2330903.

[13] Diego Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. DOI: 10.1109/JPROC.2014.2371999.

[14] Yong Li and Min Chen. "Software-Defined Network Function Virtualization: A Survey". In: *IEEE Access* 3 (2015), pp. 2542–2553. DOI: 10.1109/ACCESS.2015.2499271.

[15] Daniele Bringhenti et al. "Towards a fully automated and optimized network security functions orchestration". In: *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. 2019, pp. 1–7. DOI: 10.1109/CCCS.2019.8888130.

[16] Wikipedia. *Boolean satisfiability problem*. URL: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem (visited on 09/05/2024).

[17] Wikipedia. *Satisfiability modulo theories*. URL: https://en.wikipedia.org/wiki/Satisfiability_modulo_theories (visited on 09/05/2024).

[18] Wikipedia. *Maximum satisfiability problem*. URL: https://en.wikipedia.org/wiki/Maximum_satisfiability_problem (visited on 09/05/2024).

[19] Daniele Bringhenti et al. "Automated optimal firewall orchestration and configuration in virtualized networks". In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 2020, pp. 1–7. DOI: 10.1109/NOMS47738.2020.9110402.

[20] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[21] Daniele Bringhenti et al. "Automated Firewall Configuration in Virtual Networks". In: *IEEE Transactions on Dependable and Secure Computing* 20.2 (2023), pp. 1559–1576. DOI: 10.1109/TDSC.2022.3160293.