UCLouvain

epl

École polytechnique de Louvain

Automated rules generation into Web Application Firewall using Runtime Application Self-Protection

Author: **Marco Salemi**
Supervisors: **Ramin Sadre, Axel Legay**
Readers: **Michel Dricot, Igor Zavalyshyn**
Academic year 2019-2020
Master [120] in Cybersecurity

Abstract

With the increasing number of web applications on the industrial market, many related vulnerabilities have been discovered. Since these applications are not always fixable, cybersecurity experts use different solutions to protect them without modifying them. One of the most efficient solutions is to employ a WAF and a RASP to protect them. This gives us two lines of defence to protect ourselves from vulnerabilities. Currently, these solutions do not communicate directly with each other to improve the security in depth. This report aims to show a way to make the two tools communicate, and to improve the WAF using the RASP in order to improve the security level of this solution. Actually, by creating automatic WAF rules targeting certain types of attacks, it is possible to create this kind of solution while avoiding false positives and false negatives. Therefore, this type of new solution could become the basis of a better application security.

Keywords: Application security, Web application firewall, Runtime application self-protection, Security in depth

Acknowledgements

First of all, I would like to thank the professor Ramin Sadre of UCLouvain who has greatly contributed to the quality of this document, and who has always found a way to answer my questions despite the amount of work he had on his hands.

I would also like to thank the professor Axel Legay, who is from the same university, who has contributed to the realization of this project, and who has motivated me in this task.

And lastly, I would like to thank my family and closest relatives, who have supported me throughout my studies and during the completion of my dissertation, I dedicate this work to them.

Contents

Chapter 1

Introduction

"*Intelligence is the ability to avoid doing work, yet getting the work done.*"
Linus Torvalds

1.1 Context

During the last years, a lot of web applications have been deployed with the development of the internet. Today, all kinds of critical sectors such as finance, military, medical and governmental use this kind of application. Sadly, these applications are not bullet proof and can contain vulnerabilities.

Of course, hackers always find a new kind of vulnerability to exploit in new applications. Furthermore, old vulnerable applications still exist and are still used. The problem with this kind of application is that they cannot be patched/fixed for multiple reasons. For example, this may be because the libraries used are no longer maintained, or because the service depends on a complex architecture with many dependencies which is very difficult to change. For other applications, the problem is only the time to fix/update it in a production environment, which can take weeks in some cases.

As a result, a lot of vulnerabilities are exploited each year. Experts have classified each year the most critical types of risks in those applications (OWASP top ten). [28, 27] Popular types of attacks like SQL injection [51, 29] or cross-site scripting [47, 35] are part of this ranking.

Maintaining the highest level of security is therefore one of the most important objectives of cybersecurity experts. In recent years, experts have found multiple solutions to

deal with this problem and protect web applications against these types of attacks. We will describe some of these main solutions in the following points and then introduce our new type of solution which is based on these solutions.

1.2 State of the art

In this point we will describe the different solutions available on the market to protect vulnerable web applications without modifying them. These solutions are the ones used at the time this work was written. They have advantages and disadvantages that we will describe in order to give you a precise idea of how they are used, and which components of these tools are of interest to us.

1.2.1 Web Application Firewall

A web application firewall (WAF) is a security solution used in order to protect the web applications by examining the corresponding HTTP/HTTPS requests or the traffic models. It is used as an intermediary element when accessing the web server. It can protect the server by blocking, monitoring or filtering the traffic. Unlike classic firewalls, which block or authorize specific types of traffic, these technologies are aimed to handle malicious content into classical traffic (see figure 1.1). [52, 25, 34]

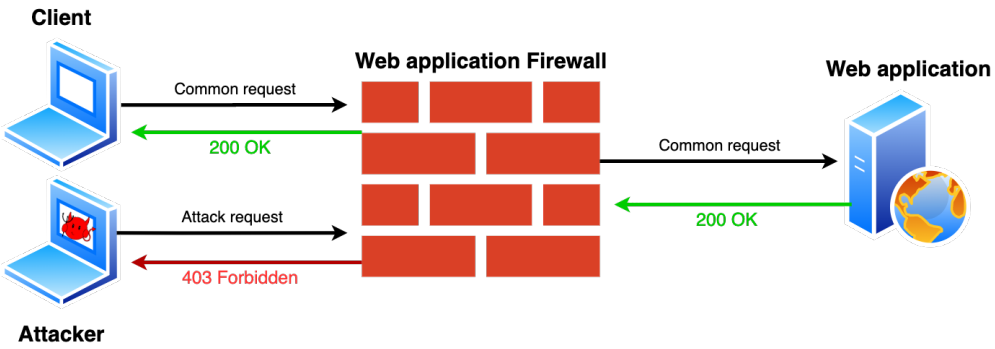


Figure 1.1: WAF example

They can be deployed in different ways such as a hardware appliance, a software plug-in for a web server, or as an add-on for existing infrastructure components, like load balancers or network firewalls. All these types of installation have different advantages/disadvantages depending on the specific case in which they must be applied

[6]. We will focus in this work on one of the recurrent cases in companies (in which they may have different web applications). This specific configuration is a WAF installed on a reverse proxy, in order to protect the web server(s) (see figure 1.2).

The first WAF was created in the early 1990s as a network firewall extension by Gene Spafford, Bill Cheswick, and Marcus Ranum. Their product marketed by the Digital Equipment Corporation (DEC) was mainly a network firewall but could handle a few applications such as FTP. Firewalls dedicated to web applications were created later, when attacks on web servers became much more noticeable. The first company to offer this solution was Perfecto Technologies with the AppShield product. [52]

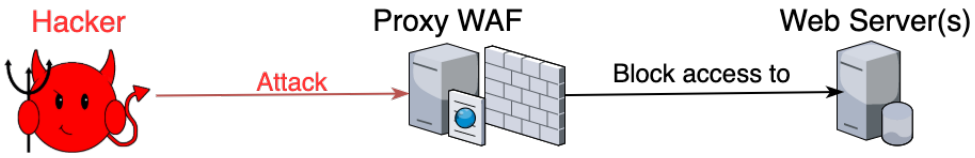


Figure 1.2: Reverse proxy WAF infrastructure

Application firewalls can use different means to fulfill their objective. The first means is pattern matching. The product will have what we call a set of rules. These rules will describe the different types of attacks that need to be filtered and will associate the appropriate response (block, monitor, let it pass, ...). The attack pattern is detected by the characters used. The use of regular expressions[1] is generally required in order to describe it.

These rules have the ability to block malicious traffic, and the ability to specify what type of response we should apply to a corresponding attack. Nevertheless, because of the way they work and their independence from the application, they suffer from a lack of precision and the necessity of maintenance by experts. [45, 38]

Indeed, a blocking rule tends to block normal traffic that should not be stopped (false positive), and to authorize malicious traffic to access the server (false negative). These are really important concepts that will be taken into account during our work. For these reasons, this type of WAF needs people to update and maintain it. In most companies, there are experts dedicated to this task, so they can use their experience to avoid false positives. In most cases, and in order to avoid false positives/negatives, the modification

[1]Regular expressions are a sequence of symbols and characters expressing a string or pattern to be searched for within a longer piece of text.

or addition of rules is done manually in a testing environment before deployment.

Afterwards, there are some more advanced WAF that use a learning mode. This is a faster method in order to minimize the number of false positives/negatives [41, 30, 45]. In learning mode, the tool learns what normal traffic looks like compared to malicious traffic and protects accordingly. Making an accurate learning is a really difficult task requiring a certain amount of data and time. Even if getting this mode right, under perfect conditions is tough, the main problem is in a production environment.

As a matter of fact, this operation still takes time. Learning to recognize normal requests requires a certain amount of traffic before you can actively block everything else. It may take a few hours to learn what safe traffic patterns are. And meanwhile, malicious requests are not actively blocked. If we consider an environment where the application code may change recurrently or where new vulnerabilities are discovered, this means that the WAF will only be in monitoring mode during this period, and thus make the web application vulnerable. In addition, the best way to improve the firewall regarding a new vulnerability is to modify it and test it in a testing environment before applying it in a production environment.

Therefore, at the moment, and in the two different WAF configuration modes, there is an ideal solution to block new attacks. This solution is based on the fact that experts will manually configure the new rules and test them before they are deployed in a testing environment. Of course, this solution takes a certain amount of time before being fully operational in production. On the other hand, they can only protect the application against attacks if they know their corresponding signature[2], which is not always the case.

The most advanced WAFs also contain other solutions to cover specific types of attacks that are not considered by normal WAFs, such as proactive bot defence, layer 7 behavioural DoS, ... [21, 4, 41, 13] But these advanced solutions are not based on the same concepts as those described above, and as these solutions only concern specific cases, they will not be taken into account in our work.

1.2.2 Runtime Application Self-Protection

For years, researchers have been looking for solutions to protect in real-time vulnerabilities in web applications that can be modified frequently. As described in the last point, the WAF cannot do this automatically and needs manual modification and time

[2] Attack signatures are rules or patterns that identify attacks or classes of attacks on a web application and its components.

to protect it. The Runtime Application Self-Protection (RASP) is a recent solution that avoids these drawbacks. Like WAF, it is a security product designed to block malicious requests. But unlike perimeter-based protections, it is based on different principles.[50, 53]

RASPs can detect or block attacks by analysing the corresponding request in the web server runtime environment. To do so, they use all the information required within the running software. This makes them more dependent on the web application and more accurate. They can stand as a last defence barrier before accessing the database, or executing a function in the application (see figure 1.3).

Runtime analysis is not a recent operation. The runtime verification terminology was formally introduced as the name of a workshop in 2001. Its objective was to address issues concerning formal verification and testing. Subsequently, different runtime verification techniques were presented under various alternative names, such as runtime monitoring, runtime checking, runtime reflection, runtime analysis, ... [19] Today, it is difficult to identify the first RASP solution created on the market because different companies offer this type of solution.

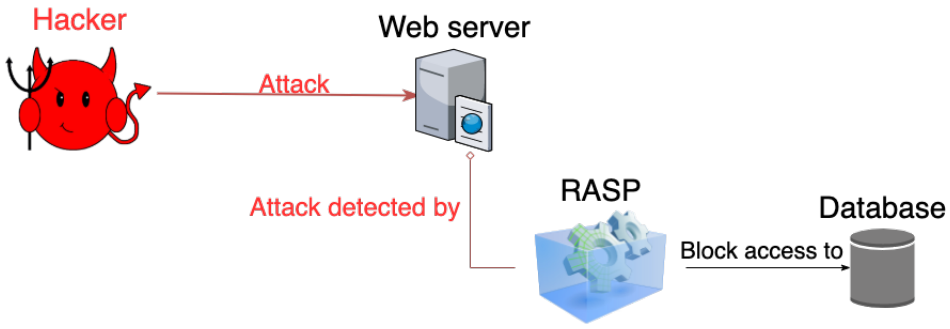


Figure 1.3: RASP example

This technology is not based on a single technique. It depends on the implementation. Each type of implementation must analyse how the data concerning a specific attack is processed in runtime (payloads, database queries, operating system commands, ...). In this way, it can detect an attack with fewer false positives and fewer false negatives.

One of the best principles on which some implementations are based is LANGSEC. [54] It is the process of formally understanding how data such as the ones described above will execute in an environment. Because it depends on building formal grammars of

languages, and because of the results given by such a technique, it should be applied whenever possible to all the different types of attacks. For example, some SQL injection could be represented as a tautology[3], in this case it must be detected as an attack or blocked. The LANGSEC concept is one of the best types of techniques, but everything from data flow analysis to heuristics can be used to implement RASP.

By their way of working, this type of tool has the best results in the application security business. The best RASPs on the market have achieved better results than the best WAFs in terms of false positives and false negatives. Some of these products tend to have no false positives at all. [10, 43, 32]

Furthermore, they can integrate a new type of attack without having to be turned off or put into monitoring mode. Unlike WAF, the ability to manage a new type of attack can be added to an existing implementation without having to test all types of attacks or traffic. In this way, the testing time before deployment is significantly reduced, while guaranteeing a low or zero false positive rate.

In addition, they could possibly handle a zero-day attack if it is related to a known attack. For example, considering the example given previously, a new SQL injection template which results in a tautology will be detected/blocked.

There are different installations types for this protection, there are mainly four categories. [14, 18] The first is serverlet filters, plug-ins and SDKs. It uses web server plug-ins or Java servlets to deal with HTTP Requests. Plug-ins inspect requests before they reach the application code, applying detection methods to each request. Requests that match known attack signatures are then detected or blocked.

The second is binary instrumentation. It brings monitoring and control elements into applications. When an application runs, the monitoring elements identify security events, including attacks, and the control elements log events, create alerts and block attacks. Depending on the programming language of the web server and its frameworks, these elements can be integrated into the server without the need to modify the application source code.

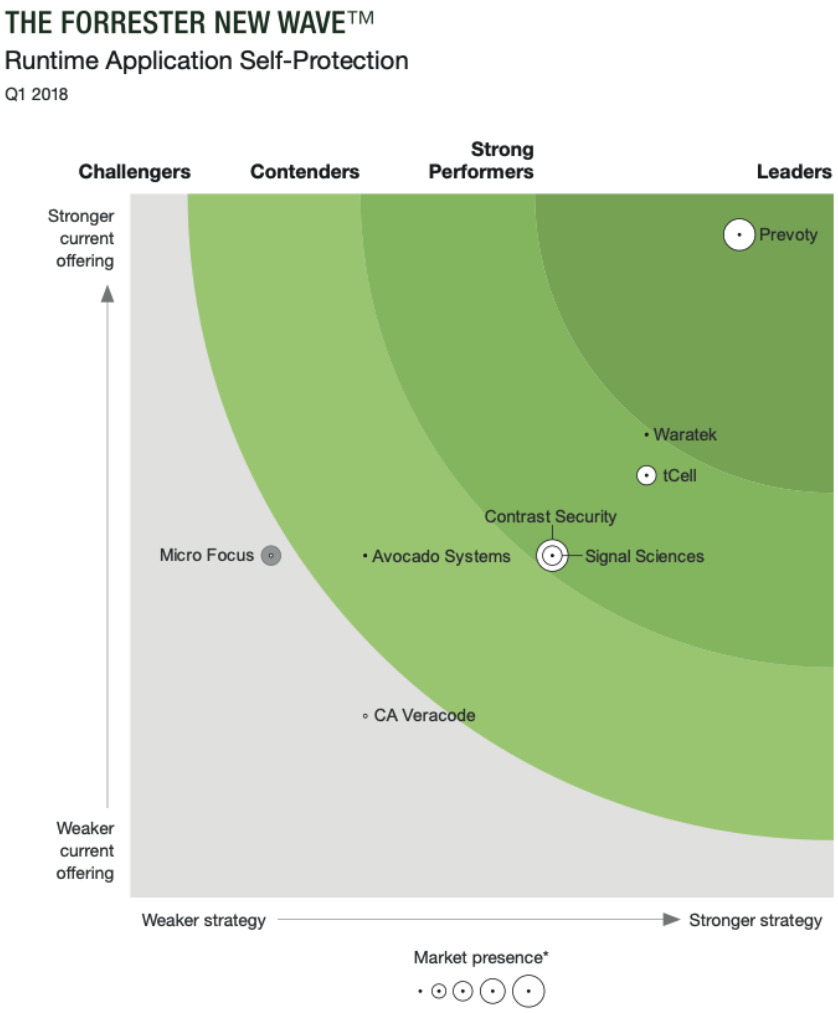
The third is JVM replacement. This is done by replacing the standard application libraries, JAR files, or JVM (Java Virtual Machine). This allows the solution to monitor application calls to the different functions, providing a comprehensive view of data

[3] A tautology in math (and logic) is a compound statement (premise and conclusion) that always produces truth. No matter what the individual parts are, the result is a true statement; a tautology is always true.

and control flows, enabling the use of detailed detection rules that can be applied as requests are intercepted.

The last one is virtualization. This type of runtime protection implements a virtual container and uses rules to govern how the application is protected. Waratek, a society selling this kind of technology, states that "its container-based solution has the advantage of allowing rule configurations to be completely separated from the application and has no impact on the application lifecycle or its normal operations."[44]

All these categories can possibly present disadvantages in terms of performance on the server depending on the implementation (CPU consumption, RAM usage, ...). Thus, it could affect the response time for the users requests on the server. A study was carried out by Forrester [9] to classify different existing solutions according to ten criteria to measure their efficiency. They found than one solution was better than the others in most of the criteria (see figure 1.4). This solution created by Prevoty is based on the first type of installation. [17]



(a) Solutions graphic

Company	Attack detection	Management UI	Attack response	Zero-day attacks	Reporting and analysis	Feedback loops	Performance load	Vision	Road map	Market approach
Prevoty	▲	▲	▲	▲	▲	▲	=	▲	=	▲
Waratek	=	▲	▲	▲	▼	▼	=	▲	=	=
tCell	=	=	=	=	▼	=	▲	=	▲	=
Contrast Security	▼	▼	=	=	=	▲	▼	▼	=	▲
Signal Sciences	=	=	=	▼	=	=	▼	=	▲	▼
Avocado Systems	▲	▼	▼	=	▼	▼	▲	=	▼	▼
Micro Focus	▼	=	▼	▼	▲	=	=	▼	▼	▼
CA Veracode	▼	▼	▼	▼	=	▼	▼	▼	▼	=

▲ Differentiated = On par ▼ Needs improvement

(b) Criteria table

Figure 1.4: RASP solutions study by Forrester

1.2.3 Best practice - Security in depth

We have seen in the previous points the advantages and principles behind RASP and WAF to protect web applications. As described in the Sans's paper about RASP [14], and most likely in various articles about application security, there is a good practice to establish regarding WAF and RASP.

In fact, considering the principle of security in depth and advantages of each solution, the application would have less probability to be exploited. Indeed, the WAF will be the first line of defence, and the RASP will be the last line of defence triggered only if the first line didn't block an attack. An interesting way to use this architecture is to use only the WAF to block the known basic attacks (like scanning, ...), which will tend to no false positive and make the RASP block the other more sophisticated attacks. This will avoid noise and bad traffic on the server, and reduce its resource consumption (CPU, memory, ...).

Furthermore, the two technologies are currently used in some companies in accordance with a SIEM tool [4]. This architecture allows a team of cybersecurity experts (Security Operations Center team or SOC team) to analyse the logs generated by the two corresponding solutions, and improve them to protect the application. For instance, they could analyse which line of code is vulnerable and develop the appropriate response on the WAF.

As a matter of fact, the combined data from both devices would provide greater context and intelligence about the attack and the perpetrator. The SOC would be more informed and could more accurately identify an attack as worthy of investigation. They would also be able to provide more specific guidance to application teams to fix an exploitable vulnerability (Contrast Security, 2018, March 20).

1.3 Contribution

Considering one of the best practices and the security in depth principle, it is understandable that further research could be conducted on the relationship between WAF and RASP. In fact, this relationship could become a major element in improving the security of the application.

[4] A security information and event management (SIEM) is a type of tool providing real-time analysis of security alerts generated by applications and network hardware. Vendors sell SIEM as software, as appliances, or as managed services; these products are also used to log security data and generate reports for compliance purposes (Wilkipedia, 2020)

The main problem with the current infrastructure is when it faces a new type of attack. This attack may be the exploitation of a newly discovered vulnerability or a new way of exploiting a known vulnerability. In both cases, WAFs have a low probability of stopping the attack. But unlike this first line of protection, RASPs have a higher probability to detect/block the attack. In this case, we can easily understand that the first line of defence has failed to protect the application and needs to be modified. Actually, this is done manually by experts in a certain period of time depending on the attack, the application and the tools used.

My contribution to this problem is to make the two technologies communicate, contrary to the actual situation where they only communicate with the SIEM. The objective is to enhance the security level of the WAF by using the alerts generated in the RASP. Moreover, this should be done in real time and without a manual interaction.

For this to be possible, we must choose which type of installation can achieve this objective and what type of improvement we want to apply. We can observe that any version of RASP could possibly communicate with a WAF using pattern matching. The communication is then translated into an automatic generation of rules in order to block attacks directly on the WAF (see figure 1.5).

This kind of generation has been done in the past. In fact, as described in the paper written by Thomas Dager from the Sans Institute [8], they used an Intrusion Detection System (IDS) to block IP addresses on a traditional firewall[5]. The IDS performs a real-time packet analysis, creating alerts for malicious behaviours. It can be used to detect a malicious IP address. Thereby, they were able to use this detection to create blocking rules inside the firewall. In this way, they created a more robust firewall to protect their network.

[5]In computing, a firewall is a network security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules [48]

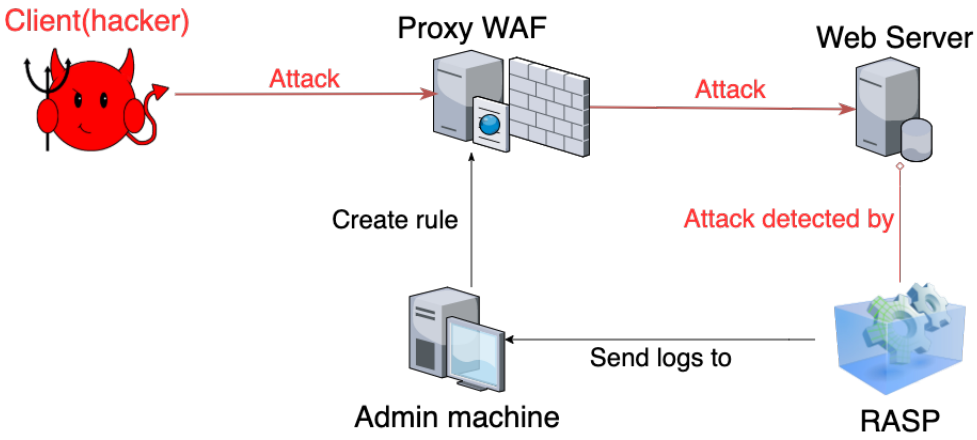


Figure 1.5: Automatic creation of rules architecture

Our objective being defined, we must also consider all the drawbacks of the WAFs in order to make an efficient creation of rules. Thus, we need to ensure that the WAF will not have a high level of false positives or false negatives. When considering a production environment, where this solution may be more useful, the priority is to focus on fewer false positives. Of course, if we have a high level of false positives, users will have difficulties to access the application, which can have really bad consequences depending on the nature of the application (for example, in the military or medical sector).

1.4 Thesis Structure

Various elements will be detailed in the next chapters of this document. The first part is intended to provide a starting point for the automatic creation of rules. To do so, we describe the different components used, how they communicate with each other, and how we have exploited them. We then describe the technical specificities of our installations, as well as the first proof that automated creation is possible.

The second part of the document aims at proving the efficiency of this kind of creation and showing its limits through a more in-depth practical proof. The different results obtained in each section, as well as the observations arising from them, will be detailed at the end of each section. A conclusion is therefore made at the end of this document in relation to these obtained results.

Part I

Basic generation of rules

Chapter 2

Architecture and technologies

2.1 Introduction

In order to prove that our concept works, we need to create the corresponding architecture. With this objective in mind, different tools were chosen and some architectural choices were made. This section will detail the different components used to create this practical part (see figure 2.1). Of course, the technologies used in this framework are based on the theoretical aspects described in the first chapter.

2.2 Architecture virtualization

With the aim of creating our specific architecture, we have used the virtualization [1]. In this way, we have created four distinct virtual machines.

The first machine contains the web server containing the vulnerable application. The second is the WAF which must stop the attacks and filter the requests. The third contains our own application to create the rules and make the link between the two other machines. And the last one represents the client who will try to access/attack the application software.

This architecture was chosen taking into account the various good practices in the field of cybersecurity. First of all, we consider the fact that the RASP should not be able to directly modify the WAF. It is therefore preferable to have an intermediate machine to

[1] In computing, virtualization refers to the act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, storage devices, and computer network resources. (Wikipedia, 2020)

ensure communication between the two. Indeed, this machine can filter the logs sent by the WAF. We can also use our administration machine to collect the logs from the WAF and check the proper functioning of our automated creation.

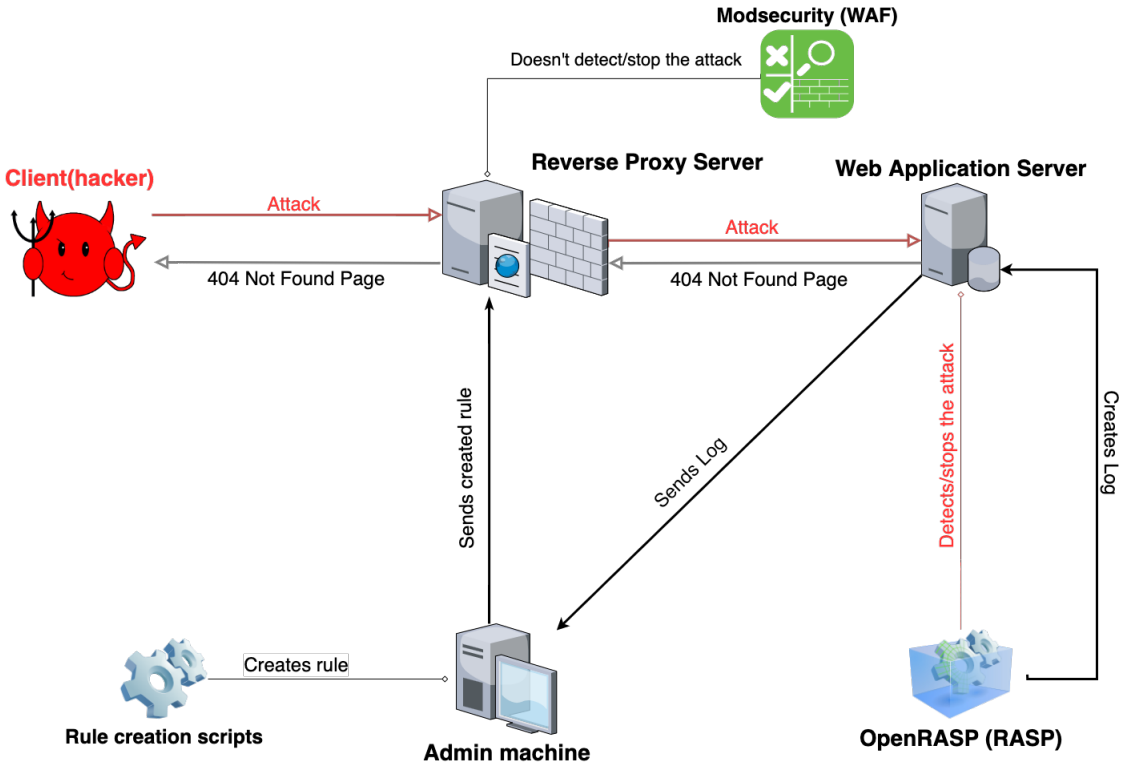


Figure 2.1: Automated creation architecture

2.3 Web Application Firewall machine

We describe in this section different components regarding the web application firewall machine namely WAF machine (software used, possible configurations, ...).

2.3.1 Reverse proxy

The web application firewall application is installed in our application on a reverse proxy server. A reverse proxy server is an intermediate server between the client and the web server(s). It permits to hide the actual machine(s) containing the application(s), it can behave as a load balancer,... [49]

It is currently a good practice to have such infrastructure in order to protect an internal network containing the different applications of a company. Furthermore, it is

interesting in our testing to use a separate machine that only contains the WAF. Like that, we can ensure that the requests are not passing the machine and don't reach the application server containing the RASP.

2.3.2 ModSecurity

Solution description

In the objective to simplify our installation and give us the power to personalize it if needed, we decided to choose an open-source and simple software. Thereby, we choose to use ModSecurity which is an Apache plug-in acting as a WAF. This solution is very popular and has been used for years. It permits us to create rules simply by modifying the dedicated configuration files. [12]

Moreover, experts have created, over years and based on their experience, a specific set of rules aimed to detect/block the generic OWASP top ten attacks (OWASP ModSecurity Core Rule Set). [40] This can be used for testing our architecture and our vulnerable application.

The ModSecurity tool can be configured in different ways. We can specify the type of information we want in the log, the maximum body size of requests, the range of possible rules ids (we have to specify an id corresponding to a rule, in order to identify the attack in the logs), which certificate we use for SSL to encrypt the communication with the client, ...

The chosen tool has two different ways to create rules. [26] We will describe in the next point the two modes to implement rules, their characteristics, and which one we have chosen in our proof of concept.

Traditional mode

The first mode is a traditional mode as described in the introduction. It is based on regular expressions that are used to detect an attack pattern.[22] This way we can specify the type of action we want to apply to the detected attack. We can choose to create a log in a specific log file (see the example in the listing 2.1), which should be used in correlation with a SIEM to create alerts for experts (monitoring). The format of the log can be configured, in our case we preferred to use the JSON format which is a popular and readable format to read the logs. We can also block the attack and describe the response we want to give to the attacker (defence). For example, we can choose to respond with a 404 page not found or a custom web page. The default behaviour, which we have kept in our current configuration, is to block the detected attack and

respond with a 403 forbidden page.

The advantages of this mode are a better performance (lower latency/resources) because the first disruptive match stops further processing, and a better understanding of how to create rules because they are fairly simple. The main disadvantage is that not all rules that could have been triggered will be recorded, only the first one. This could allow a lower severity rule to be triggered without any higher severity rule being logged.


```
{
  "transaction":{
    "time":"06/Jun/2020:07:42:28 +0200",
    "transaction_id":"XtssxMkgG4JCjtdt3-U9eAAAAAU",
    "remote_address":"192.168.1.12",
    "remote_port":53172,
    "local_address":"192.168.1.2",
    "local_port":80
  },
  "request":{
    "request_line":"GET /vulnerabilities/sqli/?id=%27+union+all+select+load_file
        %28%27%2Fetc%2Fpasswd%27%29&Submit=Submit HTTP/1.1",
    "headers":{
      "Host":"192.168.1.2",
      "User-Agent":"Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox
          /60.0",
      "Accept":"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
      "Accept-Language":"en-US,en;q=0.5",
      "Accept-Encoding":"gzip, deflate",
      "Referer":"http://192.168.1.2/vulnerabilities/sqli/",
      "Cookie":"PHPSESSID=69qjbkr2osd8n5kvgiteek6j6e; security=low",
      "Connection":"keep-alive",
      "Upgrade-Insecure-Requests":"1"
    }
  },
  "response":{
    "protocol":"HTTP/1.1",
    "status":403,
    "headers":{
      "Content-Length":"276",
      "Keep-Alive":"timeout=5, max=100",
      "Connection":"Keep-Alive",
      "Content-Type":"text/html; charset=iso-8859-1"
    },
    "body":"<!DOCTYPE HTML PUBLIC \"-//IETF//DTD HTML 2.0//EN\">\n<html><head>\n<title
        >403 Forbidden</title>\n</head><body>\n<h1>Forbidden</h1>\n<p>You don't have
        permission to access this resource.</p>\n<hr>\n<address>Apache/2.4.29 (Ubuntu)
         Server at 192.168.1.2 Port 80</address>\n</body></html>\n"
  },
  "audit_data":{
    "messages":[
      "Access denied with code 403 (phase 2). Pattern match \"vulnerabilities\\\\/sqli
          \\\\/\\\\?id\\\\=\\\\%27\\\\+union\\\\+all\\\\+select\\\\+load_file(?:.+)\"
          at REQUEST_URI. [file \"/etc/modsecurity/rasp-rules/BLOCKING_GEN_URL.conf\"]
           [line \"1\"] [id \"421\"]"
    ],
```

```
39      "error_messages":[
40        "[file \"apache2_util.c\"] [line 273] [level 3] [client 192.168.1.12] ModSecurity
              : Access denied with code 403 (phase 2). Pattern match \"vulnerabilities
              \\\\\\\\\\\\\\\\/ sqli \\\\\\\\\\\\\\\\/\\\\\\\\\\\\\\\\?id
              \\\\\\\\\\\\\\\\=\\\\\\\\\\\\\\\\%27\\\\\\\\\\\\\\\\+union\\\\\\\\\\\\\\\\+
              all\\\\\\\\\\\\\\\\+select\\\\\\\\\\\\\\\\+load_file(?:.+)\" at REQUEST_URI.
               [file \"/etc/modsecurity/rasp-rules/BLOCKING_GEN_URL.conf\"] [line \"1\"] [
              id \"421\"] [hostname \"192.168.1.2\"] [uri \"/vulnerabilities/sqli/\"] [
              unique_id \"XtssxMkgG4JCjtdt3-U9eAAAAAU\"]"
41      ],
42      "action":{
43        "intercepted":true,
44        "phase":2,
45        "message":"Pattern match \"vulnerabilities\\\\/sqli\\\\/\\\\?id\\\\=\\\\%27\\\\+
              union\\\\+all\\\\+select\\\\+load_file(?:.+)\" at REQUEST_URI."
46      },
47      "handler":"proxy-server",
48      "stopwatch":{
49        "p1":28,
50        "p2":15,
51        "p3":0,
52        "p4":0,
53        "p5":1,
54        "sr":0,
55        "sw":0,
56        "l":0,
57        "gc":0
58      },
59      "response_body_dechunked":true,
60      "producer":"ModSecurity for Apache/2.9.2 (http://www.modsecurity.org/)",
61      "server":"Apache/2.4.29 (Ubuntu)",
62      "engine_mode":"ENABLED"
63    }
64  }
```

Listing 2.1: ModSecurity log example

Anomaly scoring mode

This mode was created to overcome the drawbacks of the previous mode. As the traditional mode tends to create a certain level of false positives (as described in the introductory section), the experts found this simple solution to reduce this risk. All the features and components of the previous mode are also present here. But in this case, each rule that detects an attack will not directly perform the corresponding action.

Each triggered rule generates a defined score. When different rules are triggered, the scores generated by each rule are cumulated to obtain a final score. If the final score exceeds a certain limit defined in the configuration files, actions are performed. Experts can define the degree of criticality of a rule by generating the appropriate score for each rule.

This has several advantages. More than just having a more precise detection, this mode

allows several low severity events to trigger alerts while individual ones are suppressed. It also permits to correlate events alert and define a different threshold for different websites. On the other hand, the rules are complex to define and log monitoring scripts may need to be updated for proper analysis.

Chosen mode

After some manual testing in the two different modes, we decided that it was a better option to use the traditional mode for our operation. As a matter of fact, it is harder to test if our automated created rules detect/block an attack if we need to focus on how the scores must be generated in order to obtain a minimum value. Moreover, we want to detect how our specific rules can generate false negatives or false positives and see what are the limits of our generation. Also, we don't need to make any severity distinction between the vulnerabilities. Remarks anyway that in practice the scoring mode is more used for its better results, even the OWASP ModSecurity Core Rule Set is based on this mode.

2.4 Web Application Server machine

We describe in this section different components regarding the web application server machine namely server machine (software used, possible configurations, ...)

2.4.1 OpenRASP

The last important tool needed to finalize our architecture is a RASP. Sadly, there are not a lot of free and open-source RASP solutions available. Despite that, we have found one existing solution: OpenRASP. This solution created by Baidu is free and open-source. But this company being Chinese, its project is also documented in Chinese. I invite you to use an online translator if you want to read the specifications of the product and if you don't speak that language (the chrome navigator can do it by default) [3]. We describe in this point the different information given by the online documentation.

The solution takes place as a Javascript plug-in on the web server. It is compatible with a PHP application and easy to integrate to our actual installation. In the tests realized by Baidu and under a lot of traffic, the interface response time should only be delayed by 2~8ms which is acceptable. This represents a time loss not exceeding 5%.

OpenRASP is able to detect most of the vulnerabilities contained in the chosen web application. But some of them are only detected in the commercial version. Therefore,

we will focus in our tests on the vulnerabilities detected in the free version such as SQL injection or XSS.

The different algorithms used by OpenRasp to detect attacks are called "zero rule vulnerability detection". These are based on the LANGSEC principle as described in the introductory section. For instance, in order to detect SQL injections, the implementation is based on the number of tokens[2]. It calculates the number of tokens in the sent request without parameters, and the number of tokens in the query with parameters. If the difference exceeds two tokens, it means that the query sent is most likely an attack.

As an example, when an attacker initiates this type of request:

```
http://192.168.1.2/vulnerabilities/sqli/?id=%27+union+all+select+
load_file%28%27%2Fetc%2Fpasswd%27%29%2Cnull&Submit=Submit
```

OpenRASP will see the complete corresponding SQL query:

```
SELECT first_ name, last_ name FROM users WHERE user_ id = ”
union all select load_ file(’/etc/passwd’),null #’;
```

Like this, It will be able to tokenize the corresponding query. The resulting parsed token list is the following:

```
["SELECT", "first_ name", "last_ name", "FROM", "users",
"WHERE", "user_ id", "=", "\’ ", "\’ ", "union", "all", "select",
"load_file", "(", " \’ ", "/etc/passwd", " \’ ", ")",
",", "null"]
```

It will then remove the user’s input, and tokenize it also into the corresponding elements:

```
["SELECT", "first\_ name", "last\_ name", "FROM", "users",
"WHERE", "user\_ id", "="]
```

In this case, the solution will detect that there is a difference of more than two tokens between the queries. Therefore, this is detected as an attack.

The RASP can be applied in two modes: blocking or monitoring. In the two modes, it generates logs describing the corresponding attack (type, source, URL, corresponding detection function, ...). The logs are created in JSON format, which is a popular and readable format to read the logs. This solution can be used in correlation with SIEM tools, by pushing the logs with compatible tools such as Syslog. In our case, we do not use such tools. [42]

[2] A token is the smallest meaningful unit of information in a sequence of data for a compiler (Oxford languages, 2020)

The tool can be configured in different ways. It can specify a personalized web page or status for a blocking request (like in the WAF), the maximum number of logs per second, ... The default behaviour, which we have kept in our current configuration, is to block the detected attack and respond with a 404 page not found.

In order to create automated rules, we need information about the attacks that reach the RASP. This information is available in a corresponding JSON log file. You can see for example below all the information generated by the tool for a SQL injection (see listing 2.2). As the logs are created in a specific file, they can be collected in different ways. In our collection process, the logs are automatically transferred to the administration machine in case of modification.

```
{
    "attack_type":"sql",
    "attack_params":{
        "query":"SELECT first_name, last_name FROM users WHERE user_id = '' union all
            select load_file('/etc/passwd'),null #';",
        "server":"mysql",
        "stack":[
            "/var/www/html/vulnerabilities/sqli/source/low.php@mysqli_query:9",
            "/var/www/html/vulnerabilities/sqli/index.php@require_once:34"
        ]
    },
    "intercept_state":"block",
    "plugin_message":"SQLi - SQL query structure altered by user input, request
        parameter name: id, value: ' union all select load_file('/etc/passwd'),null
        #",
    "plugin_confidence":90,
    "plugin_algorithm":"sql_userinput",
    "plugin_name":"official",
    "event_time":"2020-03-16T12:44:20+0000",
    "source_code":[],
    "request_method":"get",
    "target":"192.168.1.2",
    "server_ip":"192.168.1.11",
    "attack_source":"192.168.1.2",
    "path":"/vulnerabilities/sqli/",
    "url":"http://192.168.1.2/vulnerabilities/sqli/?id=%27+union+all+select+load_file
        %28%27%2Fetc%2Fpasswd%27%29%2Cnull&Submit=Submit",
    "client_ip":"",
    "event_type":"attack",
    "server_hostname":"server",
    "server_type":"php",
    "server_version":"7.2.24-0ubuntu0.18.04.3",
    "request_id":"801ade0b9a4edeeb0000058c3b873380",
    "body":"",
    "rasp_id":"006149606ea2152c68a4ef03bdccb502",
    "server_nic":[
        {
            "name":"enp0s3",
            "ip":"10.0.0.111"
        },
```

```
37          {
38              "name":"enp0s8",
39              "ip":"192.168.1.11"
40          }
41      ],
42      "app_id":"",
43      "header":{
44          "host":"192.168.1.2",
45          "user-agent":"Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox
                /60.0",
46          "accept":"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
47          "accept-language":"en-US,en;q=0.5",
48          "accept-encoding":"gzip, deflate",
49          "referer":"http://192.168.1.2/vulnerabilities/sqli/",
50          "cookie":"PHPSESSID=efvnfl3npa759o01am0dorptp2; security=low",
51          "upgrade-insecure-requests":"1",
52          "x-forwarded-for":"192.168.1.12",
53          "x-forwarded-host":"192.168.1.2",
54          "x-forwarded-server":"127.0.1.1",
55          "connection":"Keep-Alive"
56      }
57  }
```

Listing 2.2: OpenRasp log example

2.5 Administration machine - Automated creation

The main purpose of this machine is to be able to use the information collected from WAF and RASP in order to create new rules and to verify that they are indeed blocking as we wish. This machine is thus in communication with the WAF and RASP.

The automated creation of the rules that must be added to the WAF is done in several steps. These operations and the steps that precede/follow this creation are described in this point and illustrated in the figure 2.2.

In accordance with my programming skills and the structure used for the logs, I decided to use the Python language. Scripts of the language are thus executed each time a log is transferred inside the machine.

Like this, a python script is executed each time an alarm is generated on the server and creates a file with the corresponding rule. With the same process, the new rule file is transferred to a dedicated folder inside the WAF machine. If the file exists and the rule doesn't exist, the rule is appended to the file and the same process take place. Otherwise, if the file exists and the rule generated is already present in the file, no modification is done and the process stops.

The addition of a rule inside ModSecurity need a reloading of the configuration by the

root user. For that reason, the new rule file is not directly transferred by the administration machine in the corresponding configuration folder of Modsecurity.

At the end of the reloading, the WAF must be able to block the new incoming attacks having the same pattern. If blocking occurs, a log is generated by the tool. As described previously, we have automated the transfer of the log files to the administration machine. This allows the developers to have all logs centralized, and thus to check if the blocking occurs.

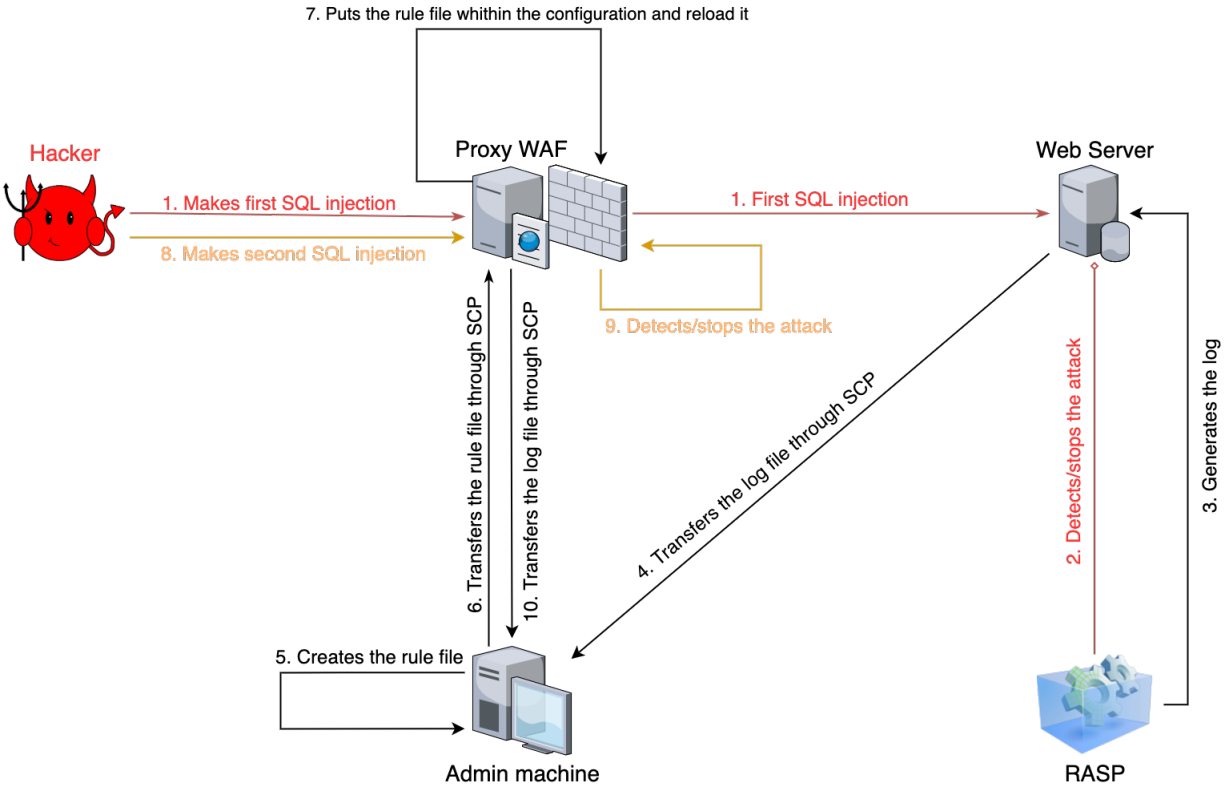


Figure 2.2: Automatic generation steps

Chapter 3

First step - Blocking IP

3.1 Introduction

Different questions can be considered regarding the automated generation of rules. Can it be applied to all types of attacks? Is it possible to make such a creation even for basic rules? Before trying a more complex and more general creation, a first simple step had to be considered in order to ensure that this type of creation is possible. This step consists in blocking IP addresses.

Actually, when a hacker attacks the server, his IP address can temporarily be considered as malicious. Like a basic firewall, the WAF is able to block a specific IP address. Note that we only block it temporarily in order to avoid blocking a legitimate client that has been tampered or spoofed without knowing it (for example if he is part of a botnet).

It can also be used to block a subnet using regular expressions. The blocking of a subnet can be applied in two cases: if this type of detection is performed in the RASP, or manually when creating rules. But we do not focus on subnet blocking in this section. First of all because OpenRASP does not analyse directly if the attacks come from the same subnet, so we don't have this distinction in the logs. And secondly because this step is just a first step to make sure we can create some basic rules. However, it might be interesting to test it in another study, to show how this type of blocking could increase the level of security and how it could be applied in practice.

3.2 Setting up

The purpose of this section is to describe the different installations/configurations made in order to test our creation.

3.2.1 Hardware specifications and Virtualization

We don't need to use powerful configurations (with multiple cores and a lot of RAM) to prove the efficiency of our tool. But as it is the case with most virtualization tools, we can modify/upgrade some components if necessary after their creation.

In this case the chosen tool for the virtualization is VirtualBox (version 6.1) which is a free and very popular solution that works on a large number of different operating systems. Our hardware and operating-system specifications for the different virtual machines are listed below. The network configurations for all the machines are a NAT access network to have access to the internet, and an internal network in order to have an isolated network for the tests.

- **WAF machine:** O.S: Ubuntu server (version: 18.04.2) - CPU number : 1 - RAM: 4096 Mo - Disk storage: 10 Go
- **Server machine:** O.S: Ubuntu server (version: 18.04.2) - CPU number: 1 - RAM: 2048 Mo - Disk storage: 10 Go
- **Admin machine:** O.S: Ubuntu server (version: 18.04.2) - CPU number: 1 - RAM: 2048 Mo - Disk storage: 10 Go
- **Client machine:** O.S: Kali linux (version: 2020.1) - CPU number: 1 - RAM: 2048 Mo - Disk storage: 20 Go

3.2.2 WAF installation and configuration

The reverse proxy was created through a basic installation of an Apache server (2.4.29 version) and some modifications inside the configuration files [16]. Requests sent to the WAF are thus transferred directly to the application server.

In our tests, we don't need to modify the ModSecurity configuration files. We have just modified the default path that contains the rules, to specify our dedicated folder path containing our automated created rules. The dedicated folder does not contain any blocking rules, which allows to verify our creation with any possible vulnerabilities.

3.2.3 Web application installation and configuration

Like described in the introduction, we need a vulnerable application for our web server. We therefore chose an existing open-source application with a very explicit name: Dawn Vulnerable Web Application (DVWA). This web application has been created to allow penetration testers to increase their skills. [37]

Thereby, this application contains the most popular vulnerabilities according to the top ten OWASP described in the introduction section (see figure 3.1). Moreover, it is compatible with the installation of our web server and only requires the installation of the languages used (Apache 2 server and PHP/MySQL languages). It can be easily installed via its git repository.

The application can be configured by security levels. It permits to train experts with four different difficulties (low, medium, high and impossible). For our automated creation, we set the difficulty to low in order to have an easily vulnerable application. This application being very popular, experts from all over the world have put online solutions to exploit it. Having these descriptions that explain how to exploit the different vulnerabilities, we used for our tests some of these solutions to verify and show our results. [7]

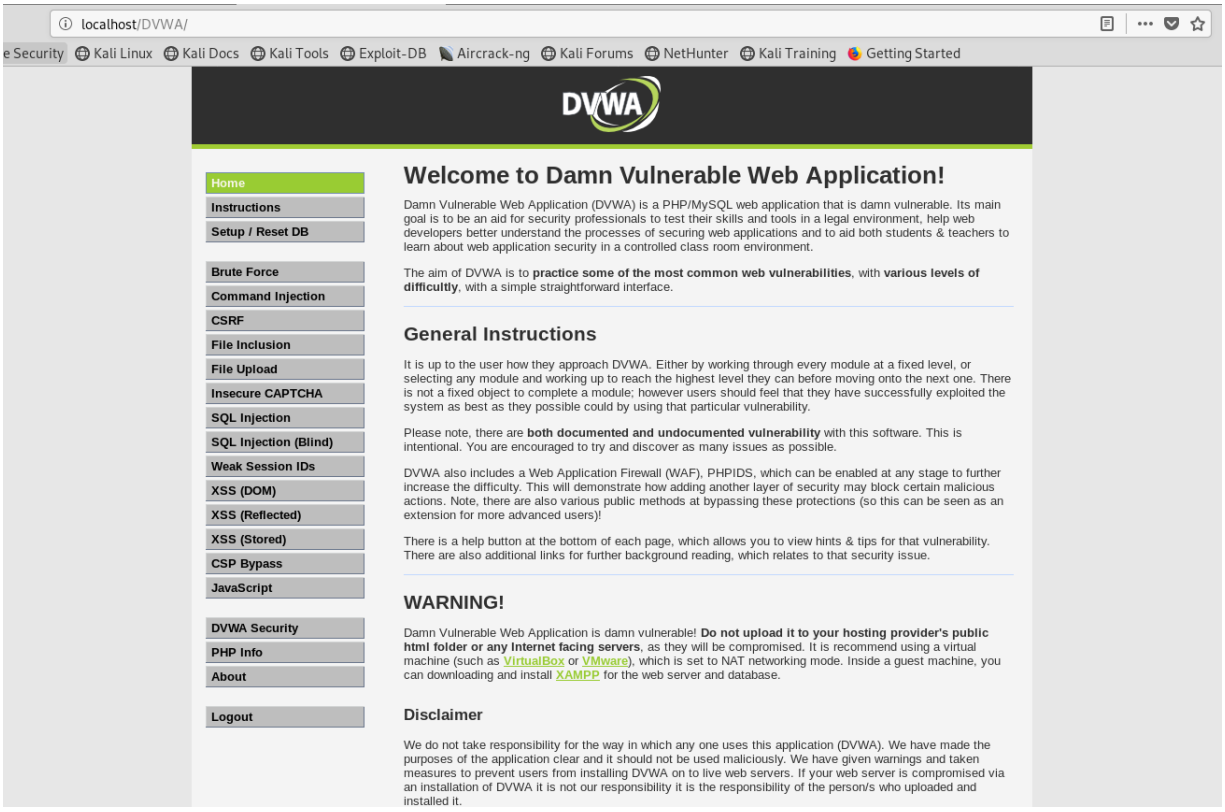


Figure 3.1: Damn Vulnerable Web Application

3.2.4 Transfer tools

To transfer the log file from the server to the administration machine, we do not use common tools such as Syslog. Since these tools need to be installed and configured, we preferred to save time by using the SCP command[1] with the Incron tool[2]. Thereby, a daemon detects each time the log file is modified and transfers it to a dedicated folder on the administration machine. The same tools and processes are used on the admin machine to send the newly created rules to the WAF, and on the WAF machine to send the log file to the administration machine.

3.2.5 Configuration reloading

The WAF server configuration is reloaded by the "Apachectl graceful" command. This command allows the server to reload its configuration without having to reboot everything. The only drawback of this command is that it can delay incoming packets by a few milliseconds, which is acceptable.

3.2.6 Penetration testing

The tests of our application are carried out by penetration tests. For this purpose, we used two machines to attack the server (the requests/attacks are sent through the WAF). The first one is the kali client. This machine is used to manually test our application and get a simple view of it in the first steps. In this way, we are able to test which specific types of patterns can exploit the vulnerabilities and to see the reaction of the different tools installed to protect the application.

The second machine used is the admin machine. This machine is very useful because it contains the logs collected as described above. This allows us to test our application in detail. But this point will be detailed in the next sections during our more in-depth tests.

3.3 Log information

The OpenRASP tool generates different information when creating a log as described in the previous chapter. Thereby, we can observe in the log a value indicating the IP of

[1]SCP (secure copy) is a command-line utility that allows you to securely copy files and directories between two locations. It relies on SSH for transferring which is encrypted and secure.[1]

[2]Incron is a daemon which monitors file system events and execute commands defined in system and user tables. [2]

the attacker.

The first reaction is to the use the **attack_source** value inside the log. But this is a mistake. Indeed, this value doesn't correspond to the client IP. It is the IP of the WAF machine acting as a reverse proxy. The correct value to use in this case is **x-forwarded-for** (inside the header field). This way we have the attacker's IP address.

3.4 Rule generated

The rule must be generated following the ModSecurity rule syntax. The syntax for permanently blocking an IP is as follows:

```
SecRule REMOTE_ADDR "@ipMatch IP_ADDR" "id:ID_NUMBER"
```

Where **IP_ADDR** corresponds to an IP address and **ID_NUMBER** corresponds to the ID number we set to identify the blocking rule.

ModSecurity using regular expressions to describe the attacks, we have to use the '\' character to make sure that the special characters are not considered in regular expressions. In this case, the ' . ' character must not be considered as 'any character possible'. Using '\.' instead will cause the WAF to consider it as a single dot.

3.5 Temporary blocking

ModSecurity syntax does not allow a temporary IP blocking. In order to permit this temporary blocking, we have to do it ourselves. By removing the rule in the configuration file and reloading the configuration, we easily understand that the blocking rule becomes temporary.

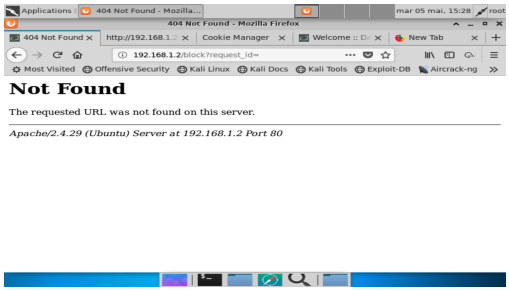
To do so, we have created a shell script that will wait a defined amount of time, and then remove the rule inside the configuration file. After that, the configuration is reloaded. This script is run each time a blocking rule is added inside the configuration. This is done in order to make disappear each IP blocking rule after the defined amount of time. This script being really basic, if the same IP attacks again during the waiting time, the rule lifetime is not extended.

3.6 Test

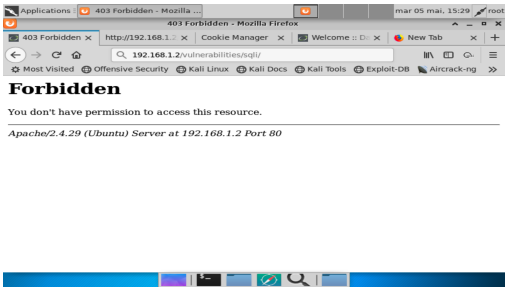
In order to test this creation, no automation of scripts was necessary. Our objective being to see if a creation is possible and if the WAF would block an IP, a manual test was enough. To achieve this goal, we used the web browser of the Kali client (Firefox) to display the page returned after two attacks.

As shown in figure 3.2, we attack and then we access the web page. The purpose of the attack is to generate the rule automatically. The response to the request is therefore a "404 page not found" as configured in the RASP. The access to the page aims at verifying that the WAF blocks the IP after the rule generation. In this case, the response of the page is a "403 Forbidden page", and we do not have access to the website for the defined period of time. The attack used for this test is an SQL injection.

(a) SQL injection



(b) Blocking page by RASP



(c) Blocking page by WAF

Figure 3.2: IP blocking testing

3.7 Conclusion

By this first approach and practical proof, we have implemented an automated creation of basic rules. Thanks to this, we can observe various elements resulting from this successful operation.

Firstly, automatic rule generation is possible when the rule needs information from the RASP logs. This is an important point for us, where we understand that we can create rules to handle certain types of attacks. The limits of this creation must be defined in our different tests.

Secondly, the creation may require some adjustments in order to be applied within the WAF tool. We understand that the creation really depends on the implementation of the tools or their functionalities. This could make the creation more complex if the tweaking is difficult to apply.

And thirdly, we observe that a general creation of rules will not be possible. Indeed, each rule is based on a certain type of syntax. These syntax aims to handle different functionalities for different types of attacks. Our goal is therefore to make an automated creation for different types of attacks, and not to create a one-size-fits-all generation of rules.

Part II

Specific attacks generation of rules

Chapter 4

URL based attacks

4.1 Introduction

After completing the first step in the previous chapter, we concluded that rule creation should be specific to certain types of attacks. By considering the types of attacks present in the top ten OWASP, we made an interesting observation: some attacks such as SQL or XSS injections are entirely applied on a URL.

Indeed, when we want to communicate with a web page, we send content through parameters specified in the URL. Thus, the content sent to exploit a vulnerability is fully integrated into the URL.

For example, if the URL is "http://192.168.1.2/sqlpage/?id=%27Salemi%27%3B%20−%20", the pattern used for attacking the server is described in the id parameter.

During this chapter, we will use this assumption in order to generate rules that aims to block URLs describing these types of attacks.

4.2 Vulnerabilities

Two main vulnerabilities have been considered to test our rule generation for URLs. We explain in this section how we can exploit these two vulnerabilities.

Firstly, we have exploited the SQL injection. When the server does not filter the content given by the clients, this kind of vulnerability can be expected. The attack is based on a bad treatment of this content which allows to inject certain types of queries on the SQL database. Like this, hackers can execute requests on the SQL database to obtain

or modify credentials, private data, ... In our case, and as described in the example in section 2.4.1, there is an SQL injection pattern detected by RASP that allows to attack the web application. Thus, the vulnerability can be exploited by the following content:

```
1 '␣union␣all␣select␣load_file('/etc/passwd'),null␣#';
```

By using this pattern in the application, we obtain all the information located in the "/etc/passwd" file which is the default file containing the users informations in Linux machines. Indeed, the server will execute the following request on the database:

```
1 SELECT first_name, last_name FROM users WHERE user_id = ''
    union all select load_file('/etc/passwd'),null #';
```

We can observe that the first character of the attack aims to close the content of the id parameter, and the union keyword aims to execute the following command. Thereby, we are able to execute any command on the SQL database and get all the data we want from it.

The second vulnerability exploited is cross-site scripting (XSS). Like SQL injection, this type of vulnerability is predictable when the server does not filter the content provided by the clients. In fact, it is exploited by the hackers who inject code inside the web page.

The attack is limited by the languages used by the server and the type of XSS. Some are non-persistent (reflected) which means they are executed on the client browser and therefore not saved on the server. Others are persistent (stored) because they are stored on the server. The second type of vulnerability is more devastating because it can affect all the users. The type of attack used in our tests is from the first type of XSS. We can thus exploit the vulnerability with the following content:

```
1 <script>alert("hello␣marco")<\/script>
```

By doing so, we use a simple HTML/JavaScript script to show a personalized message. In this case, the exploit doesn't do anything really malicious, but we can use this method to do just about anything the language allows, such as displaying cookies on the home page. In some cases, hackers can exploit this vulnerability to steal customer cookies from another web page.

4.3 Log information

The OpenRASP tool generates different information when creating a log as in the 2.4.1 section. Thereby, we can observe in the log the **url** value indicating the encoded URL used for the attack.

This value can be used in correlation with the **target** value in order to remove the beginning of the URL (for example: https://192.168.1.2). In this way, we can obtain the part containing the vulnerable page and the content used to attack the server.

4.4 Rule generated

Given the part of the URL we can get by following the information in the log, we are able to create two types of rules: a specific URL-based rule and a general URL-based rule. In this point we will describe both types and their usage.

4.4.1 Specific URL

This first URL-based rule creation is quite simple. Considering the fact that we have the exact content used to attack the server, we use that exact content. This allows the WAF to detect/stop the exact pattern previously used to attack the server. Like this, we are sure that the attack will be stopped if repeated.

ModSecurity using regular expressions to describe the attacks, we have to use the '\' character to make sure that the special characters are not considered in regular expressions. For instance, the ' . ' character must not be considered as 'any character possible'. Using '\.' instead will cause the WAF to consider it as a single dot.

After this operation, we can create a rule following the ModSecurity rule syntax. The syntax for blocking an URL is as follows:

```
SecRule REQUEST_URI "SPECIFIC_URL" "deny,id:ID_NUMBER"
```

Where **SPECIFIC_URL** corresponds to an encoded specific URL and **ID_NUMBER** corresponds to the ID number we define in order to identify the blocking rule.

4.4.2 General URL

After the automated creation following a specific URL, we noticed that this type of creation has a main disadvantage. In fact, a hacker can exploit the same vulnerability with the same pattern but not the same tokens. For instance, he could generate a different attack by simply adding a random word inside the attack. In this way, the specific URL rule will not stop the new coming attacks, and a new rule will be generated for each attack.

Considering this case, we have created another type of rule: the general URL rule. To do so, we transform the URL content given by the logs into a more general pattern using regular expressions. This type of rule is directly inspired by the rules created by the WAF experts.

Thus, we decided to take into account reserved words in languages in order to create rules. In fact, by replacing each word of the content that is not a reserved word with a general regular expression, we have a more general rule that will match the different attacks. In our implementation, we have replaced the non-reserved words by the expression "(?:.+)" which corresponds in the ModSecurity syntax to a regular expression describing one or more characters.

To achieve this goal, we need to make a different modification per attack. In our creation, we did this operation for the SQL injection and the XSS. For each replacement, we collected the list of reserved words (SQL for SQL injection and HTML/Javascript/PHP for XSS) that must be replaced. [31, 46, 24]

In a similar way as for the previous rule, we must avoid some elements to be considered as special characters in regular expressions. Like this, the special characters present in all URL such as '%' or '+' must be replaced by '\%' or '\+'. This new created rule is following the same syntax as the precedent rule. Below you will find an example of a general rule generated from an attack similar to the one described in the 2.4.1 section example.

```
SecRule REQUEST_URI "vulnerabilities/sqli/
\?id=%27\+union\+select(?:.+)from(?:.+)" "deny,id:573"
```

4.5 False positives testing

In both types of rules, different automated tests must be implemented in order to verify that the WAF doesn't generate false positives (FP) and false negatives (FN). Thus, we

must ensure that a rule will be generated to block the same attack and that the normal traffic is not blocked. To do this, we have created one first type of test following these steps:

1. Make a first attack
2. Make a normal access
3. Wait 1 second to make sure the blocking occurs
4. Verify if the RASP blocked the attack in the corresponding logs (and not the WAF)
5. Wait 3 seconds to make sure the rule is created on WAF
6. Make a second attack (with the same pattern as the first attack)
7. Make a normal access (with the same pattern as before)
8. Wait 1 second to make sure the blocking occurs
9. Verify if the WAF blocked the attack in the corresponding logs (and so not the RASP)
10. Verify if the normal traffic is not blocked in the corresponding logs

The first step is preceded by the login to the application (required to have access to the vulnerable page), and the generated cookie is used in the different steps. These steps are repeated a defined number of times (100) to have a good picture of how the creation is going. We execute these steps 100 times to test an SQL injection vulnerability, and then the same number of times to test an XSS vulnerability. We exploit them using the same kind of pattern described in the previous creation. We also add a random ID/word inside the attack to have a different attack each time the steps are repeated. The number of waiting seconds was chosen based on our observations during the creation of this test. Cleaning scripts must be executed manually after the tests in order to guarantee the validity of the tests. This cleanup erases the different logs and rules created on the three machines (WAF,RASP and admin).

After creating this first type of intuitive test, we observed that some elements could be modified. For this reason, we have created a second version of tests taking into account the following modifications.

Firstly, we don't need to do two verification in the logs. The WAF being without any blocking rules at the beginning (we clean it after the tests) and the attacks being all different, we are sure that the firewall won't block the first attacks. Thus, we don't need to test if the first blocking is done by the RASP. In case we have a successful verification

after the second attack, we are sure that the RASP blocked it the first time and that a correct rule has been created on the WAF.

Secondly, we find that the tests are done with too much waiting. We would like to optimize the time to do the tests. We would also like to identify how long it takes to create a rule, and how the different tools are reacting when we make a lot of requests in a short period of time. Thereby, we have modified the steps for testing the creation. In fact, we don't need to do the verification directly after the attack because the log file contains all the different blocked attacks. And since the attacks are all different, we can verify that the blocking occurs after several minutes or hours (depending on how many tests we want to perform).

Thirdly, we observe that we send normal traffic twice (after the first attack and after the second). Actually, the first access is not required since we don't modify the RASP, and we know from testing it manually that no FN is created on this tool.

- **First step (for 100 different attacks):**
    1. Make the first attacks
    2. Wait the defined time (to check if the rule is created on WAF)
    3. Make all the second attacks
    4. Make all the normal accesses
- **Second step (for the same attacks patterns):**
    1. Verify if the WAF blocked all the attacks in the corresponding logs (FN verification)
    2. Verify if the normal traffic was not blocked in the corresponding logs (FP verification)

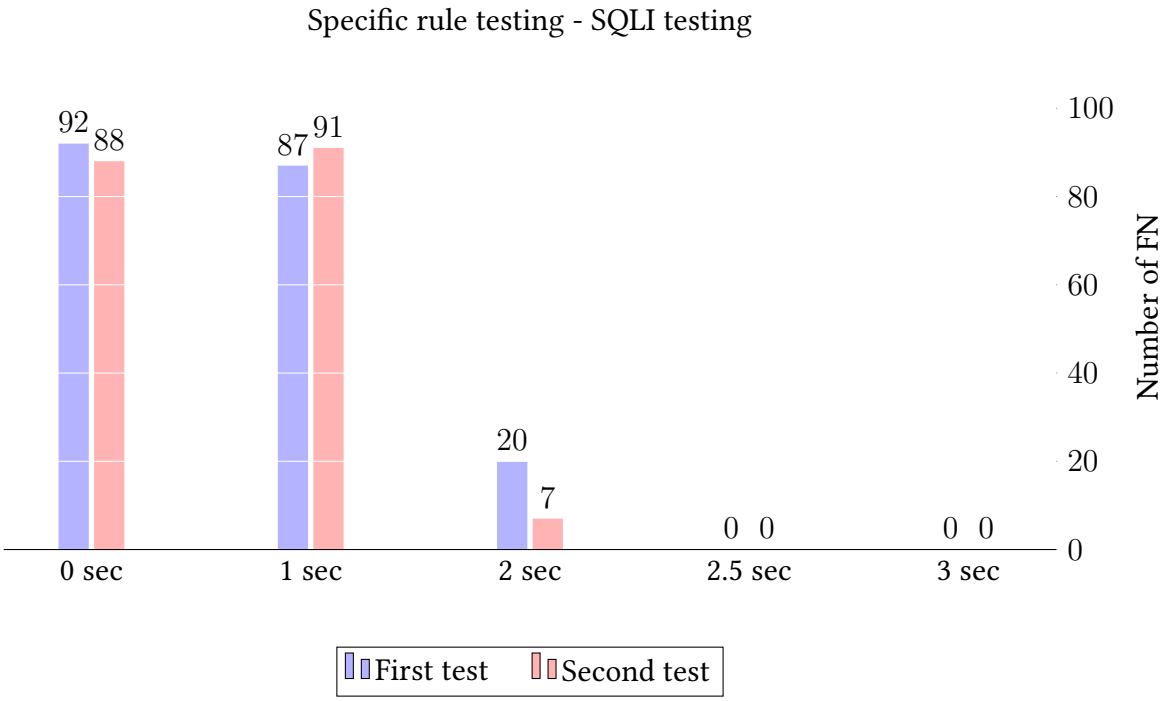
The time parameter can be configured in order to define how much time we wait before the second attacks. In this way, we can test different values and see how long our creation process takes.

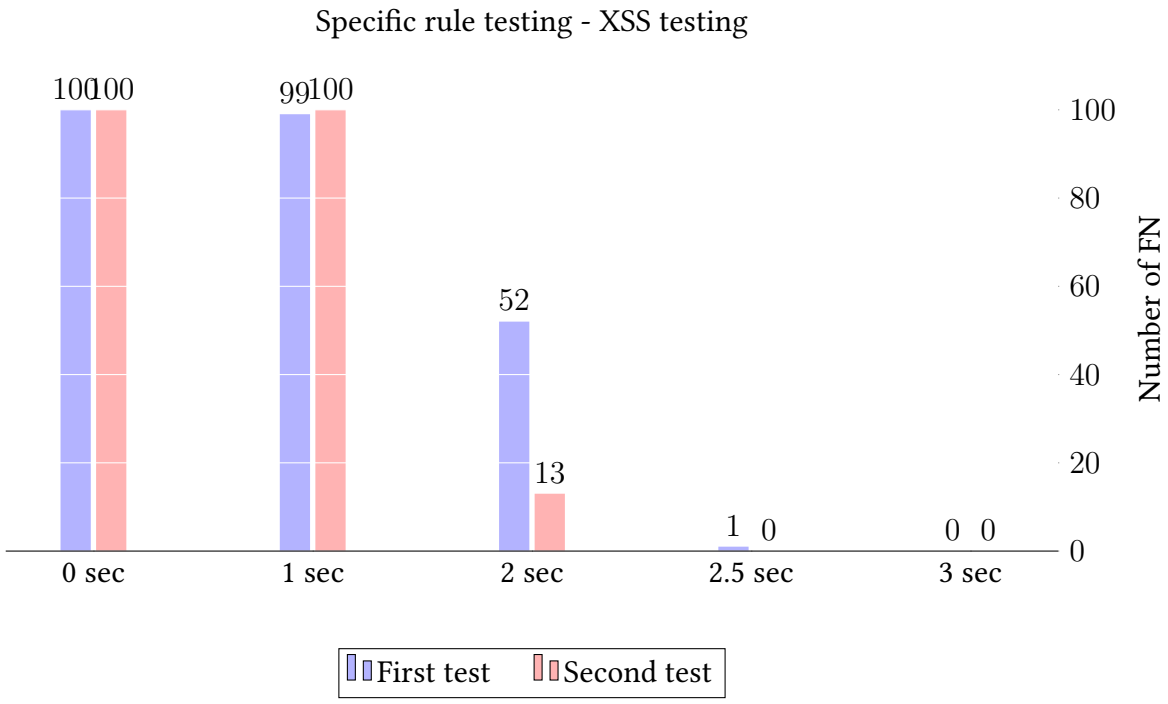
We did two tests (optimized version) in relation to the two types of rules we created for URL blocking. The results of these tests are presented in the next points.

4.5.1 Specific URL

The hundred attacks were executed and tested several times with different values for the time variable. You can see the different results given for a specific URL in this table and these bar charts:

Test number	**Time variable** (in seconds)	**SQLI FP**	**SQLI FN**	**XSS FP**	**XSS FN**
1	0	0	92	0	100
2	0	0	88	0	100
3	1	0	87	0	99
4	1	0	91	0	100
5	2	0	20	0	52
6	2	0	7	0	13
7	2.5	0	0	0	1
8	2.5	0	0	0	0
9	3	0	0	0	0
10	3	0	0	0	0





We can make different observations regarding these tests. The first and more important observation concerns the number of false positives (FP). In fact, we have never had a single FP which is understandable given the specific rule we have created. Thus, we can apply this type of creation in an environment without being afraid to block normal traffic.

After that, we can see that the number of FN is decreasing with the rising time. This is due to the time it takes to create the rule. In fact, a FN occurs when the second attack is not blocked by the WAF. We thus observed that the rules were effectively created inside the WAF, but after the second attack. In this way, we can observe the number of FN to have an idea of the probabilities of stopping a second attack after the defined seconds. We can thus assume that we will always have a blocking rule 3 seconds after an attack, that we will almost always have a blocking rule after 2.5 seconds, and that we will almost never have a blocking rule in less than 2 seconds.

Finally, we can observe another fact with the FN values. We see that we can expect some variations with the same waiting time, these variations can be explained by the different steps in our automated creation process. In fact, we can identify three main elements in our implemented creation that could influence these values by slowing down the process: the consecutive scripts execution to create the rules on the admin

machine, the consecutive SCP command to transfer the logs and the rules on the different machines, and the number of successive "Apachectl graceful" commands to reload the configuration of the WAF machine.

After having created logs for each step in our process and analysed the exact time (with a millisecond precision) to perform these operations, we discovered which component can slow down during the process. This component is the execution of the different scripts. Indeed, by running a large number of scripts (and therefore a large number of new processes on the machine) to create such rules, the Linux scheduler allows some processes to run faster and others to wait longer. In our tests, it happens because we make a lot of attacks in a short period of time.

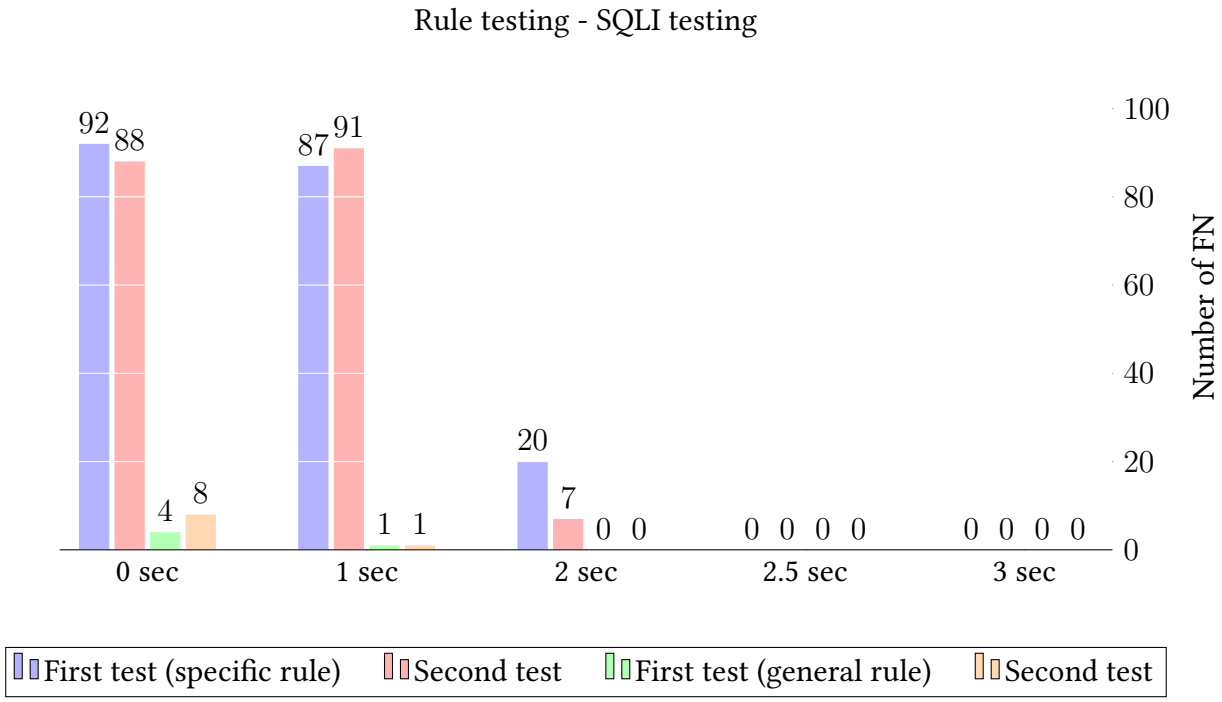
A hacker could use this slowdown to perform a denial of service [1] on the admin machine. Indeed, if this one is overwhelmed by the creation of many rules, the period of time needed to block all the attacks could be increased to its paroxysm. This would then allow the hacker to attack the RASP in the meantime with another type of pattern. This type of case can be totally prevented if the WAF has an anti DOS functionality. In fact, to succeed with this kind of behavior on the admin machine, the attacker must perform many attacks in a very short time, which is usually detected by this kind of tool. Another possible solution would be to optimize the way of creating rules. This could be done either by a low-level language or by a dedicated machine with higher performance than the machine used for the tests.

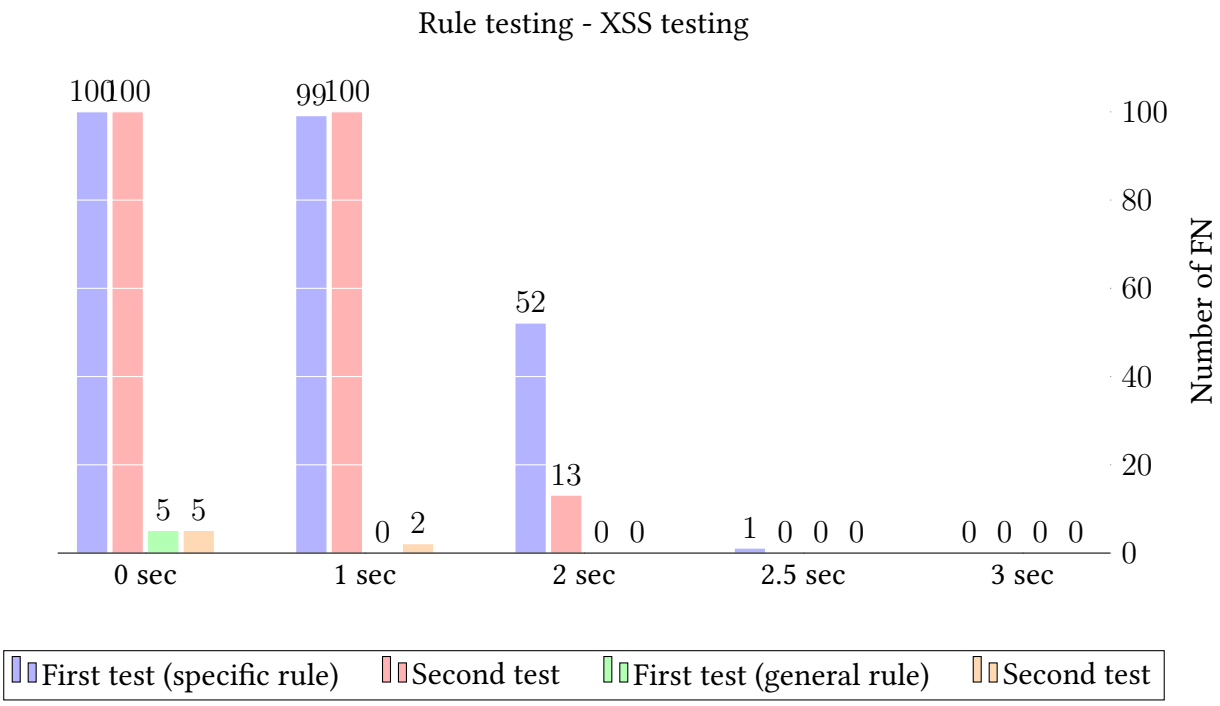
4.5.2 General URL

As for the specific rule, hundred attacks were executed and tested several times with different values for the time variable. You can see the different results given for a general URL in this table and these bar charts:

[1] A denial-of-service (DoS) attack is a type of cyber attack in which a malicious actor aims to render a computer or other device unavailable to its intended users by interrupting the device's normal functioning. DoS attacks typically function by overwhelming or flooding a targeted machine with requests until normal traffic is unable to be processed, resulting in denial-of-service to addition users. A DoS attack is characterized by using a single computer to launch the attack. (CloudFare, 2020)

Test number	**Time variable** (in seconds)	**SQLI FP**	**SQLI FN**	**XSS FP**	**XSS FN**
1	0	0	4	0	5
2	0	0	8	0	5
3	1	0	1	0	0
4	1	0	1	0	2
5	2	0	0	0	0
6	2	0	0	0	0
7	2.5	0	0	0	0
8	2.5	0	0	0	0
9	3	0	0	0	0
10	3	0	0	0	0





As for the previous rule tests, we notice that we have no false positives. This is a good result in our case, but we cannot guarantee that this will always be the case. Indeed, since this creation is based on regular expressions, we may face the same disadvantage that WAFs have shown in practice in companies. For example, some unexpected characters or patterns could cause our creation of regular expressions to block normal traffic because the replacement does not occur as expected. In fact, it is impossible to prove that a tool is totally unable to create an FP, the results of the tests are in most of the cases practical evidence based on the experience of the tool.

Then, we can look at the FN values and compare them to the previous rule. We observe a major difference from the precedent rule. In fact, that is the main advantage of the general rules. When the first rule is generated, all the following attacks are blocked, and no other rule is required to block the same attack pattern. In this case, the few number of FN that is created results from the time needed to generate the rule.

As before, you can see these values decreasing with the increasing time. In this case, we cannot consider the values as probabilities since the first rule will block all the attacks. But we notice that they can give us an idea of the time needed to create this kind of rule for each type of attack. As a matter of fact, when the time is set to one second, one or two FNs occur. This means that the creation of rules must always take more than

one second and sometimes more than two seconds.

Moreover, we can also notice that when the time is set to two seconds, we have no FN. But we cannot use this information to describe the maximum time needed to create the rule because we don't have the time needed to run the testing scripts in this tabular. In practice, by analysing the logs created on the machine at each creation, we have seen that the creation time never exceeds four seconds. But this is the case in our tests, when only one rule is created. If a lot of different patterns are used and several rules have to be created in a short time, the execution scripts should slow down as described with the previous rule.

4.6 Consumption testing

Another final test was carried out on the different machines. This test aims to analyse the CPU and memory consumption on the different machines during a lot of rules creation. In order to show a good visualization of these values, we have used the htop command [33] which allows to have a good view on the CPU/memory usage, the execution of the different processes on the machine, and the corresponding time they spent in user and system mode. We have thus observed the consumption before the test executions and the consumption during the test execution. We have used for that the tests described in the previous section for the generation of specific rules with a time value of zero seconds.

The results showed that the CPU passed from nearly 0 % to 100 % on the administration machine, from nearly 0 % to maximum 6 % on the web application server, and from nearly 0 % to 70 % on the firewall machine. The RAM consumption is not much increasing on all the machines. You therefore notice that the admin machine is consuming a lot of CPU for the creation, that the WAF is triggered in order to block the attacks, and that the RASP does not consume much more (as described in the OpenRasp documentation and in the first step section). This test also confirms the observations made in the previous section about the generation of specific rules for URL blocking.

4.7 Conclusion

In this chapter, we have created two different types of rules to block attacks using the content sent inside the URLs. We have therefore created a specific blocking based on the exact content sent, and a more general blocking using regular expressions.

Both types of rules have shown pros and cons, but mostly demonstrate that they have

created no false positives (FP) and no false negatives (FN) during our tests. Like this, we can assume that this type of blocking would give good results if it were applied in companies. We can thus consider different application domains.

The first is the testing environment. In this context, the creation of rules could be used as a way to create rules inside the WAF (already containing rules or not). By attacking the application in different ways, we can test that the general/specific blocking does not generate FN or FP and that the WAF is more robust.

The second application concerns the production environment. In this situation, we want to be sure that not FP are created in priority. Given our results, we can assume that it could be used in this case for both types of URL blocking. But since the general blocking is based on regular expressions, we cannot guarantee that no false positives could be created. For this reason, the creation of rules should be tested in a monitoring mode before applying it in a blocking mode. This type of mode could be created by simply modifying the created rules so that they simply log the attacks on the WAF but not block them (e.g. changing "deny,log" into "pass,log" in the ModSecurity rules). During this time, the RASP must protect the application, which should be short regarding our results.

Chapter 5

Conclusions

We have demonstrated in our project different successful automated rule creation in WAF using RASP. Due to the number of false negatives and the absence of false positives, we can claim that this type of solution is really interesting to increase the security in depth.

Firstly, we have shown that a temporary blocking of IP addresses can be done without difficulty. This could also be used to mitigate the attacks and reduce the malicious traffic reaching the given architecture.

Secondly, we have shown that we can block several types of attacks using the URL used by hackers to attack the application sever. In this way, we have created two types of blocking. One being really precise and blocking the exact content used to attack. And the other one being more general, using regular expressions to create a general pattern based on the reserved words of the languages in order to block the attacks.

As a matter of fact, this new kind of architecture could become the basis for a better application security. Being not much costly to maintain, it could be used in small companies to ensure a certain level of security when they don't have an internal team of experts to update the rules. It could also be used in critical sectors to mitigate/stop the zero-day attacks following the same principles as known attacks, and thus automatically update the WAF rules. This creation being fast and automated, we can obtain a more robust WAF without even knowing the new discovered pattern to attack the application.

Of course, various improvements,optimizations or modifications could be made regarding our creation. For example, we can look for different and better ways to create a

general rule, or integrate other different types of attacks. It would also be interesting in the future to test this kind of creation using the tools that give the best results on the market.

The actual solution created to automate rule creation depends on the tools chosen to implement a proof of concept. But recent WAF and RASP are almost all creating logs for a SIEM. Thus, we can assume than minor modifications to the actual work must be required to be applied to other tools. Therefore, this contribution to the web application security could be used as a basis to enhance the actual solutions.

Bibliography

[1] ArchLinux. How to use scp command to securely transfer files. https://linuxize.com/post/how-to-use-scp-command-to-securely-transfer-files/, May 2020.

[2] ArchLinux. Incron. https://wiki.archlinux.org/index.php/Incron, May 2020.

[3] Baidu. Openrasp. https://rasp.baidu.com/, 2018.

[4] Kelly Brazil. What's so next-gen about the next-gen waf? https://securityboulevard.com/2019/02/whats-so-next-gen-about-the-next-gen-waf/, February 2019. VP of Sales Engineering.

[5] Alejandro Perez-Villegas Carmen Torrano-Gimenez and Gonzalo Alvarez. An anomaly-based web application firewall. https://pdfs.semanticscholar.org/74da/9669a4f80272a9c3573958713c4570abcea0.pdf, 2009.

[6] Cloudflare. What is a waf? - web application firewall explained. https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf/, 2020.

[7] BERTRAND Cédric. Dvwa. https://lepouvoirclapratique.com/blog/wp-content/uploads/2018/01/UE_Cybersecurite-TP_DVWA_v0.6.pdf, October 2017.

[8] Thomas Dager. Using open source to create a cohesive firewall/ids system. https://www.sans.org/reading-room/whitepapers/firewalls/paper/792, March 2020.

[9] Amy DeMartine. The forrester new wave: Runtime application self-protection. https://advance.biz-tech-insights.com/whitepaper/The-Forrester-New-Wave.pdf, March 2018.

[10] Micro Focus. Security fortify application defender. https://www.microfocus.com/media/data-sheet/security_fortify_application_defender_ds.pdf, October 2019.

[11] Christian Folini. Including owasp modsecurity core rule set. https://www.netnea.com/cms/apache-tutorial-7_including-modsecurity-core-rules/, November 2019.

[12] Christian Folini and Ivan Ristić. *ModSecurity handbook*. Feisty Duck, 2018.

[13] Fortinet. Fortiweb: Web application firewall and api protection. https://www.fortinet.com/products/web-application-firewall/fortiweb#services, 2020.

[14] Alexander Fry. Runtime application self-protection (rasp), investigation of the effectiveness of a rasp solution in protecting known vulnerable target applications. https://www.sans.org/reading-room/whitepapers/application/runtime-application-self-protection-rasp-investigation-effectiveness-rasp-solution-protecting-vulnerable-target-applications-38950, April 2019.

[15] Vivek Gite. Linux incrond inotify monitor directories for changes and take action. https://www.cyberciti.biz/faq/linux-inotify-examples-to-replicate-directories/, December 2018.

[16] Steven Gordon. Building an internal network in virtualbox. https://sandilands.info/sgordon/building-internal-network-virtualbox, August 2018.

[17] Imperva. Runtime application self-protection (rasp). https://www.imperva.com/products/runtime-application-self-protection-rasp/, June 2020.

[18] Adrian Lane. Understanding and selecting runtime application self- protection. https://securosis.com/assets/library/attachments/Understanding_RASP_Immunio_V2.pdf, August 2016.

[19] Adrian Lane. Runtime verification. https://en.wikipedia.org/wiki/Runtime_verification, April 2020.

[20] Linuxize. How to use scp command to securely transfer files. https://linuxize.com/post/how-to-use-scp-command-to-securely-transfer-files/, May 2020.

[21] Brian A. Mchenry. What makes a waf advanced? https://www.f5.com/company/blog/what-makes-a-waf-advanced, April 2018.

[22] ModSecurity. Regular expressions. http://index-of.co.uk/Programming/Regular%20Expressions%20(Appendix%20B%20from%20ModSecurity%202.5).pdf, June 2020.

[23] Asaad Moosa. Artificial neural network based web application firewall for sql injection. https://publications.waset.org/1001/pdf, 2010. World Academy of Science, Engineering and Technology International Journal of Computer and Information Engineering Vol:4, No:4.

[24] Mozilla. Html elements reference. https://developer.mozilla.org/en-US/docs/Web/HTML/Element, June 2020.

[25] OWASP. Best practices: Use of web application firewalls. https://owasp.org/www-pdf-archive/Best_Practices_WAF_v105.en.pdf, March 2008.

[26] OWASP. Anomaly scoring mode - owasp crs documentation. https://www.modsecurity.org/CRS/Documentation/anomaly.html, 2015.

[27] OWASP. Owasp api security project. https://owasp.org/www-project-api-security/, 2019.

[28] OWASP. Owasp top ten. https://owasp.org/www-project-top-ten/, 2020.

[29] OWASP. Sql injection. https://owasp.org/www-community/attacks/SQL_Injection, June 2020.

[30] Dariusz PałkaMarek and ZacharaMarek Zachara. Learning web application firewall - benefits and caveats. https://www.researchgate.net/publication/226351120_Learning_Web_Application_Firewall_-_Benefits_and_Caveats, August 2011.

[31] PHP. List of keywords. https://www.php.net/manual/fr/reserved.keywords.php, June 2020.

[32] Prevoty. A guide to runtime application self-protection (rasp). https://s3-us-west-2.amazonaws.com/prevotyexternaldocs/2018/180131-Guide-to-RASP-External.pdf, 2017.

[33] Ubuntu Manpage Repository. htop [wiki ubuntu-fr] - doc ubuntu. http://manpages.ubuntu.com/manpages/bionic/man1/htop.1.html, March 2016.

[34] Chad Russell. *Web Application Firewalls*. O'Reilly Media Inc., 2018.

[35] Kirsten S. Cross site scripting (xss). https://owasp.org/www-community/attacks/xss/, June 2020.

[36] Amal Saha and Sugata Sanyal. Application layer intrusion detection with combination of explicit-rule-based and machine learning algorithms and deployment in cyber-defence program! https://arxiv.org/pdf/1411.3089.pdf, November 2014.

[37] Dewhurst Security. Damn vulnerable web application (dvwa). http://www.dvwa.co.uk, May 2020.

[38] Digital Security. Waf through the eyes of hackers. https://habr.com/en/company/dsec/blog/454592/, June 2019.

[39] SpiderLabs. Modsecurity reference manual (v2.x). https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual-%28v2.x%29#Variables, January 2020.

[40] SpiderLabs. Owasp modsecurity core rule set (crs) project (official repository). https://github.com/SpiderLabs/owasp-modsecurity-crs, May 2020.

[41] Sucuri. Website hack protection. https://sucuri.net/website-hack-protection/, 2020.

[42] Syslog-ng. Syslog-ng open source edition - technical documentation. https://www.syslog-ng.com, June 2020.

[43] Waratek. Waf to runtime protection. https://www.waratek.com/waf-to-runtime-protection/, September 2018.

[44] Waratek. Runtime security, real-time protection. https://www.waratek.com, June 2020.

[45] James Wickett. It's time to break up with your waf. https://devops.com/time-break-waf/, October 2017.

[46] Wikibooks. Php programming/reserved words. https://en.wikibooks.org/wiki/PHP_Programming/Reserved_words, April 2020.

[47] Wilkipedia. Cross-site scripting. https://en.wikipedia.org/wiki/Cross-site_scripting, June 2020.

[48] Wilkipedia. Firewall (computing). https://en.wikipedia.org/wiki/Firewall_(computing), June 2020.

[49] Wilkipedia. Reverse proxy. https://en.wikipedia.org/wiki/Reverse_proxy, January 2020.

[50] Wilkipedia. Runtime application self-protection. https://en.wikipedia.org/wiki/Runtime_application_self-protection, April 2020.

[51] Wilkipedia. Sql injection. https://en.wikipedia.org/wiki/SQL_injection, June 2020.

[52] Wilkipedia. Web application firewall. https://en.wikipedia.org/wiki/Web_application_firewall, April 2020,.

[53] Jeff Williams. Introduction to rasp. https://dzone.com/refcardz/introduction-to-rasp?chapter=1, January 2019.

[54] Kacy Zurkus. Is language-theoretic security the answer to internet insecurity? https://www.csoonline.com/article/3033917/is-language-theoretic-security-the-answer-to-internet-insecurity.html, February 2016.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**