Bachelor Degree Project

Web application and WAF vulnerabilities

Author: Mirza Durakovic
Supervisor: Ola Flygt
Semester: VT 2017
Subject: Computer Science
**Acronyms**

CSRF – Cross-site request forgery  
DB - Database  
DBMS - Database Management System  
DVWA - Damn Vulnerable Web application  
IPS - Intrusion Prevention System  
MITM - Man In The Middle  
SQL – Structured Query Language  
SQLi – SQL Injection  
URL – Uniform Resource Locator  
Web app – Web application  
WAF – Web application firewall  
XSS – Cross-site scripting
Abstract
With the growth of the internet, web pages evolved from being simple static information providers to web sites and nowadays to web apps. Symbiotically, flaws in security followed this development which in its turn could cause devastating damages to many areas of our lives. This report researches vulnerabilities that exist in web apps nowadays, in addition to the lack of proper protection that WAFs should offer. Selected web applications and WAFs with default rulesets were put through the tests for SQLi and XSS attacks. The conclusion that was derived was that the WAFs in question were not secure enough and that in fact there are issues with out-of-the-box rules that come with Web Application Firewalls in general.

Keywords: web apps, web application firewall, WAF, XSS, SQLi, internet security
1 Introduction
This chapter will give short introduction of the development of the web apps, their history and evolution. Even though there was some previous research later mentioned on certain aspects of this report, the general feeling was that there is a lack of hands-on studies that show the actual behavior of WAFs when attacked. In order to gradually develop this report, background of web apps will be explained followed by exact problem formulation and research questions this report will try to answer.

1.1 Background
Looking at the history of the web from the very beginning, evolution of the content offered by web servers could be categorized into simple web pages, web sites, and lately, web applications where it all came together. In order to understand the difference, we need to shortly explain each of them.

Web pages were the first that offered web content. Knowing a little bit of HTML was sufficient to make simple, informational web pages. Although there were tools which could be used to create them, even a simple text editor was sufficient to create a web page.

Even though many mix terms web page and website and use them in the same context, according to Leon Shklar and Rich Rosen[1], they are immensely different. Authors claim that the biggest distinction between a group of web pages and a web site is that web sites are more strict with fixed related content. Also, aesthetic of the web site is maintained by using the same design and layout. Lastly, architecture is of a big concern when developing a web site since planning and predicting the future development is crucial in successful organization of the web site content.

Since the early days of web expansion, the mentioned content was mostly static. This would mean that the web sites offered static information without any interaction with the user. Slowly but surely this changed and dynamic web sites started emerging offering interactive usage and giving different results based on diverse parameters thus making them web applications.

So, a web application is software that is client-server orientated and where client is web browser based. Server side in this relationship offers all the computational operations such as transmitting, processing and storing data while client side inputs the information and gets representation of the result.

Today, it is unimaginable to think of the web or internet without web applications. Actions like searching the web, online shopping, banking, login forms etc. are all dealt with web applications so its significance is huge.

1.2 Previous research
There is a number of research studies done in the field of IT security in general. While some research web app weaknesses [2][3], others explore and analyze WAF concepts[4]. Mike Shema in his book [3] in a very structured and easy to read way explains some of the most severe attacks on web apps such as XSS which he compares to the cockroaches of the web and SQLi. He later concentrates on web sites design deficiencies and how they can create loopholes which can be exploited. The author explains also vulnerabilities that may occur solely based on the web server platform (apache, IIS) even though the same web app is installed on them. This depends on the settings server administrators need to be aware of and which are, of course, different for different web servers. At the end, M. Shema focuses on the security of web browsers,
and switches the direction of the attacks from how he says “someone targeting a web site to someone using a web site to target the browser”.

Z. Ghanbari, Y. Rahmani, H. Ghaffarian, M. Hossein Ahmadzadegan in their report [4] offer some advantage points of WAFs. They explain that even though a problem resides in a web app, it is not often possible to fix the web app because it could potentially require downtime and huge costs so WAFs are installed as a layer of protection. Authors list and analyze the characteristics of a standard WAF and make a comparison between IPS and WAF, later shortly reflected in this report also.

However, I have not managed to find a study that combines this knowledge together with practical part. It is what this report will do. It will explain from the very beginning to the end and how it looks like in real life.

1.3 Problem formulation

Parallel with the development of web applications, security issues started to emerge. Since most are custom made software and not off-the-shelf products they are more prone to coding flaws. This results in a variety of security problems and vulnerabilities. Considering the fact that most of the web applications are connected with databases, exploiting these vulnerabilities can result in disclosing sensitive information which can be, for example, personal and/or financial in nature. Attacks needed to be dealt with and solved, thus, web application firewalls (WAFs) emerged. This report aims to address and discuss web application and WAF security issues and to explore the status of some of the currently available solutions. The goal is to examine web application weaknesses but more importantly, weaknesses that reside within the very rules of the software that is meant to solve them. By doing experiments on an open source WAFs with out-of-the-box ruleset, expected results will show how the (current) selected WAF solutions can be bypassed and can be considered insecure still. Exposing the security issues of WAF, hopefully, will help these security solutions to provide stronger protection in the future.

1.4 Motivation

Since it is almost impossible to think of Internet without web application nowadays, the security of them is one of the most crucial concerns in this area. WAFs are considered crucial part of any website security planning upon which many businesses rely and depend. Therefore, it is necessary to do research on the possibility of doing certain attacks such as SQLi and XSS on web applications with WAFs and default ruleset in place and see is it still possible to succeed.

1.5 Research Question

In order to cover the topic better, the following research questions were developed:

<table>
<thead>
<tr>
<th>RQ1</th>
<th>What are the most common security issues at present time in this field?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ2</td>
<td>What are the most common successful attacks on web applications and how do they work?</td>
</tr>
<tr>
<td>RQ3</td>
<td>How is it possible to bypass WAFs?</td>
</tr>
<tr>
<td>RQ4</td>
<td>What are the difficulties in finding solutions for these vulnerabilities?</td>
</tr>
</tbody>
</table>
Answering them one by one will gradually introduce the reader to the material and at the end, give conclusion. It will be possible to learn about types of the known attacks on web applications. It will also show how these attacks work and how common are they today. Furthermore, the report will explore WAFs and show in what way (if any) is it possible to bypass this security mechanism. Answering that question can help us decide how much trust we should give to these solutions. Also, since some of these vulnerabilities are known for few years, this study will try to research if these issues can be resolved and if yes, why are they not yet.

1.6 Scope/Limitation

Since this topic is broad, different aspects can be processed in the studies dedicated to each. This study however will concentrate mainly on the vulnerabilities of web applications and WAFs. It will try to show if it is possible and in what way to take advantage of security issues present in these apps and WAFs.

Due to extensiveness, this report will not offer guidelines and solutions since this can be researched and written in a study of its own. Choice of WAFs will be limited to those which are open source since proprietary ones need licensing.

1.7 Target group

This study attempts to contribute to the IT security field by revealing vulnerabilities still present today in the topic areas. This can be useful to IT professionals engaged in security or those who want to learn what the security issues are today.
2 Method
This chapter will describe the scientific approach that will be used to answer the research problem.

2.1 Scientific Approach
To be able to examine this subject a quantitative method will be used. It will be in the form of multiple experiments that will test the security of web apps and WAFs. Furthermore, this report draws conclusions and inspiration from other works as part of a literature study. You can find more information in the following sections. Results will be shown numerically and then carefully analyzed.

2.2 Method description
The report will be written in 2 major sections. First, known vulnerabilities will be theoretically explained. During the theoretical part small examples will be included to show how these attacks work on a small scale in order to easier understand the concept. After that, WAFs will be theoretically explained with some examples of WAF rules. Then, practical part will come with the experiment which will be divided into 4 phases:

Phase 1: Setting up virtual machines.
This phase would be concentrated on setting and running three different virtual server machines. They will be run on VirtualBox virtualization software. Operating system on each of them will be some distribution of Linux since it is open source. Next, webservers will be installed and the choice will depend on the platform chosen WAFs require. So, in this case, two will have Apache since two out of three WAFs can work only on this web server and the third WAF will be installed on Nginx.

Phase 2: Testing web applications and showing how and in what way they are vulnerable.
After setting up virtual machines, next step would be setting up web applications. For these experiments DVWA, BWAPP and Mutillidae are chosen since they are one of the most popular purposely vulnerable web applications. Many different attacks can be tested on these web applications but in this case, web applications will be tested against:
- SQLi
- XSS
Since these 2 types of attacks are most common and can be very devastating, the testing will be mostly concentrated on them[5]. Considering that they are very much different in a way they work or breach security, they will be tested separately. In order to automate the process of testing, a script will be made that will automate the attacks and show the result of successful and unsuccessful attacks. The script will check for the response from the server and based on if the WAF has been activated or not, conclude if the attack was successful.

Phase 3: Testing web applications with different WAF solution.
Next step is putting different WAF solutions in front of these apps, on each virtual machine. After setting up WAFs, the script will be used again and now set up and running WAFs will be tested. Same type of attacks would be performed and quantitative results of successful and unsuccessful attacks will be shown. Behavior of the WAFs is going to be analyzed.

Phase 4: Manual tweaking of attack vectors in order to increase the chance of success and bypassing WAFs.
Since the nature of the attacks on WAFs are delicate and sensitive to minor character adjustments or encoding, automated attacks can be efficient only to a certain degree and to a certain WAFs. All WAFs do not work in exactly same way and do not filter the same so dedication to each of these WAFs and manually testing and changing attacks in a way they specifically work would be highly beneficial.

2.3 Reliability and Validity

Reliability and validity requirements should be satisfied in this report. The experiment will be done using a developed script and anyone using that script on WAFs in question will get the same results as in this report. Of course, provided that the same payloads are used. Finally, conclusions drawn at the end will solely be based on the results of the experiment.

2.4 Ethical Considerations

While the purpose of this report is purely educational and meant to be helpful to the community and vendors, one can discuss that it has elements that can be seen as malicious. In the same way that this report hopes to find vulnerabilities in order to make the products stronger, it can also be used for malicious actions. For example one could exploit the information and the knowledge presented in this work and use it for unethical purpose. Another aspect that needs to be addressed is a confidentiality issue since this thesis is dealing with WAFs that could be present on a production level somewhere. This is considered sensitive, especially because the WAFs will be linked to different vulnerabilities (if found). If vulnerabilities are found the issue will be handled by protecting real names of WAFs linked to that particular vulnerability.

If during this report some vulnerabilities are found then the ethical principle of beneficence needs to be addressed, since this would mean that the vulnerability could be used for material purpose, i.e. selling the information.
3  Web application vulnerabilities

Web application vulnerabilities are many. This report will first explain some of the most common ones and in what way can they be exploited. Also, many, but not all, will be applied and tested if they are working later on in the experiment phase. Due to extensiveness of each topic, all the examples are the simplest ones just so the idea behind it is understood.

3.1 SQLi

Probably the best known vulnerability, SQLi has been around for years now. Although known, it is still widely present and poses a huge threat to those who do not keep it in mind when designing and building web applications.

As mentioned before, web applications do some sort of computation, store or retrieve data etc. In these cases, it is not uncommon for them to be connected with some sort of SQL database where this data is stored. By doing some request on the web site, web application translates our demand to a SQL query language that corresponds to that SQL database and retrieves data. In this way, if certain security mechanisms are not in place, it is possible to adjust those requests and do a query on a database through the web application just as it would be normally done on a database. Based on M. Shema’s book “Hacking Web Apps”[3], SQL injection attacks can be classified into 3 types:

- Error based injection
- Union based injection
- Blind injection (boolean based and time based)

3.1.1 Error based injection

Error based injection is nothing more than injecting a certain SQL command in web application and seeing is there a response from the database. The response can be some actual result or an exception that the query cannot be completed. In both cases, it means that the injection is working and that the web application is vulnerable. For example, let us imagine that on some website with names directory we want to find all the people named “Mirza”. In search field, we will enter Mirza. Programing language that the web site is made with, for instance PHP, will parse that search query inside the quotations (‘Mirza’) to the DB and the DB will return the result of all the names. Translated to the SQL language, this query would look like this:

```
SELECT * FROM users WHERE name IS ‘Mirza’;
```

Knowing this, we could manipulate our search term, and instead of typing Mirza, we could type Mirza’

```
SELECT * FROM users WHERE name = ‘Mirza’;
```

Since this is a syntax error and the last quote is not closed, DB would throw an exception. If it is not handled properly, it would be shown and then the attacker could see that the web application is prone to this type of weakness. Furthermore, many combinations using this vulnerability are possible. Another example is if an attacker uses the following tautology:
Again, this would translate into SQL language looking as follows:

```
SELECT * FROM users WHERE name = 'Mirza' or 1=1--'
```

What the query is saying here is “give me all records from users table where the name is “Mirza” or true (1=1). Of course, since there is a condition that is always true, 1=1, an attacker would end up with all the records from that table. Double dash at the end cancels any commands after, so it ignores the quote after and an attacker does not get an exception for unclosed quotation.

Another example of seriousness of this vulnerability is a possibility of logging into a system without really providing any credentials. When the logging happens, after providing username and password, it is translated into an SQL language such as:

```
SELECT COUNT(*) FROM users WHERE username = 'mirza' AND password = 'password'
```

This checks how many users are with the provided username and password. If it exists (and only one should exist), the returned value would be 1, and the system would know that the user exits and would let him/her login. Again, applying the above explained method an attacker could put the following in a username (or password) field:

```
mirza’ or 1=1--'
```

This would translate into:

```
SELECT COUNT(*) FROM users WHERE username = 'mirza’ or 1=1--' AND password = ''
```

Now, it does not even matter if there is a provided username since there is a condition that is true, the query is being translated as “give me a count of all users where this is true”, and it will count all the users in the table. Depending on how the application is handling the login, it is possible to log in this way and the assigned username would be the first one in the DB table.

The above showed examples are just a small fraction of the malicious actions that can be performed. Basically, if a web application has this vulnerability, it is possible to, for example, add users, update passwords etc.

### 3.1.2 Union based injection

In order to exploit this vulnerability, instead going in totally blind and trying all sorts of methods, it is imperative for an attacker to try to acquire the knowledge about the DB through the web application. If known, tables, schemas and similar can be of a huge help in extracting the information later.

By testing the above methods of discovering if a web app is vulnerable to SQL injection, an attacker could then proceed in trying to discover the tables. One way would be guessing what DBMS is used and then trying to access the information in that DB system table. For example, if it is Microsoft SQL Server DBMS, then sys.tables would be where the information about all the tables is stored. By doing some research, an
attacker could learn the structure of that table and get the information about the tables by doing following:

‘union select object_id, name from sys.tables--

By using UNION clause, the final result would be joined with the information gathered from the second query. All names and unique table object ID’s of each table would be outputted. From there, it is possible to get the columns of each table and then not only extract, but also insert data.

Another way would be by guessing the tables based on what does the web app do. If it is a user registry, the table name would probably be “users”, “profiles”, “nicknames” etc.

3.1.3 Blind SQL injections

Both error based injection and union based injection were mainly focused on the response an attacker gets from the system. Depending on the error or the exception an attacker could tweak his/hers query and get the required result. With blind SQL injection, there is no response from the DB or at least not shown which makes it much harder to exploit. Instead, an attacker depends on the different result output to draw conclusions. For example, let us imagine we are on a vulnerable website and the last part of the URL looks like this:

/Players/Leaderboard?orderBy=Goals

In a query language, this would look like the following:

`SELECT * FROM players ORDER BY goals`

Since the names of any tables is not known, an attacker can start with a similar approach as with union based injection, and that is, testing to see which DBMS it is and using the system table. Then, it is possible to extract the names of the table letter by letter by testing certain queries and getting them, in a sense of speaking, from the result.

Let us assume that in this scenario there are other sorting ways also, not only by goals, but also by assists, yellow cards, red cards etc:

/Players/Leaderboard?orderBy=Assists
/Players/Leaderboard?orderBy=YellowCard
/Players/Leaderboard?orderBy=RedCard

So for example, query like this could be used:

`/Players/Leaderboard?orderBy=case when(select top 1 ascii(substring(name,1,1)) from sys.tables)<=10 then assists else yellowcard end`

What this query is saying is “give me top row, from the column name of the sys.table, first letter and compare is it equal or less than m. If it is, show players by assists, if not, show players by yellow card”.

Depending on the result, some conclusions can be taken. If the result is sorted by players assist, then the first letter of the table is in a range of letter a to and including letter m. With the following queries, an attacker can cut down the search and end up
with a correct first letter, after which he/she can continue to the next one and so on. This is called boolean SQL injection.

On the other hand, time based SQL injection can be used when there is no response at all, or at least not even in a way of sorting as seen before. In that case, an attacker could figure out the database tables by purposely delaying the response from the database. For example, let us assume that we want to find out the first letter of the first table in sys.tables again.

```sql
userId=62&teamId=2&comments=best'); if (select top 1 ascii(substring(name,1,1))) from sys.tables)<=109 waitfor delay '00:00:05'--
```

If the first letter is in a range from a to m, then the response will come after 5 seconds. The rest of the procedure is similar to the boolean SQLi.

### 3.2 XSS and session hijacking

There are several types of session hijacking, such as MITM attacks where an attacker sniffs the network and obtains the session ID or using malware to obtain the cookie with the session ID. This report will concentrate on XSS.

XSS is injecting malicious scripts (in this case, JavaScript) from the client side into the web sites in order to obtain victim's cookie/session ID. This method exploits the possibility of inserting untrusted data into the web page. There are two types of XSS, reflected and persistent[6].

Reflected XSS is based on reflecting that data into the code of the web app. If an attacker enters some data, for instance, search field and that data gets repeated back by the web page, then the web page reflected the data. Inspecting the source of that web page will probably show the data is reflected in the JavaScript code (if used). This is where the problem lies. It means that an attacker can manipulate the code by his/hers wishes. In other words, it is possible to construct an URL with the legitimate web site path, but also inject into JavaScript code that logs the cookie and sends it to the other website[6]. All that is needed for this type of an attack is:

- XSS vulnerable website
- Logged user
- Unawareness and not inspecting the link before clicking

Persistent XSS is much more dangerous type of XSS since there is no need of constructing a URL and trying to get the victim to click on it as it is in reflected XSS. Instead, an attacker would save a malicious script on the web site itself and anyone visiting that web page would unknowingly pass the session ID to the attacker. For instance, posting something on a forum which is XSS vulnerable would allow the attacker to add a script after the comment and store it permanently. Since it is a script tag, it will not show in the body of the message. Anyone visiting this webpage would execute the code and by that also malicious script. This makes this kind of XSS devastating and the scope of the possible targets is much larger than in reflected[7].

XSS is widespread, and very easy for an attacker to detect. By simple script injecting such as `<script>alert(1)</script>` an attacker can test for a vulnerability and plan further actions if this vulnerability is present.
3.3 Cross-site request forgery (CSRF)

While XSS is stealing an auth cookie and using it to become logged in as victim without knowing victim's credentials, CSRF works the opposite. It tries to change the password to an attackers liking without the need to possess the auth cookie. It does however exploit cookies once more.

For the sake of explanation, let us assume that a victim is registered and logged in a web page example.com. On example.com/page/ChangePassword is a form where a victim can change a password by entering new password and confirming that new password. Attacker, knowing that the victim is logged, can send a link with a new POST command. That link can be his/hers malicious website which on the first sight has nothing suspicious about it. In the background however, in the web page code, it can contain a coded link like this:

\[
\text{http://example.com/page/ChangePassword?password\_new=EvilPassword&Change}\]

Since a victim is logged in on example.com from before, a cookie is created so that when the victim visits a page again in a shorter period of time he/she does not need to enter credentials again. CSRF exploits that weakness and can change settings of the user profile without ever dealing with victim’s cookie and/or username and password.

This attack can be performed on variety of services, among others, online banking. If a website is not protected against this attack, it is even possible to transfer funds from one account to another.

3.4 Other vulnerabilities

There are many other vulnerabilities worth processing but can only be mentioned considering the scope of this report. Some of those are:

- Insecure Direct Object References
- Security Misconfiguration
- Sensitive Data exposure
- Missing function level access control etc[5].
4 WAFs

Z. Ghanbari, Y. Rahmani, H. Ghaffarian, M. Hossein Ahmazadegan explain in their work[4] how the idea of the HTTP is far different than how it’s being used nowadays. HTTP as such is a stateless protocol but stateful programs are needed to maintain client sessions. That, accompanied with different strategies, technologies and programming languages in creating these web apps is an almost guarantee that there will be security issues.

Although many times confused with Intrusion Prevention Systems (IPS), web application firewalls are a special piece of software and differ from IPS in ways more than one. Compared to IPS, WAFs are, as name implies, concentrated around securing web applications and not the whole network as is case of IPS. Although both inspect packets, IPS discards suspected packets while WAFs need to keep them for the sake of maintaining the overall context of the request and the response of the server. Since WAFs work on Layer 7 of the OSI model[9] (application layer), it needs to recognize protocol commands like GET or POST but also a number of other programming languages and mechanisms needed for the HTTP protocol to function, such as HTML, Javascript, SQL, cookies etc. IPS, working on layer 3 and 4, do not need to know about the above and therefore, provide less security. IPS as such, does not inspect packets intended as attacks on the application layer and because of that, it is nearly impossible to detect it[10][11].

![Figure 4.1.: IPS and WAF](image)

Even though there are differences between IPS and WAFs, they still can work together, as shown on Figure 4.1. While an “ordinary” firewall can repel non-HTTP, it is the WAFs duty to recognize and reject application level attacks.

WAFs work on a concept of having a set of rules that define the action done by the firewall. Many of these are proprietary and a company can pay and not worry about it anymore. Others come with a default rule set, which does indeed provide a certain level of security, but still requires certain level of tweaking to achieve desired level of security. To get an idea about how rules on WAFs look like, the following is the example of them:

```
SecRule ARGS\REQUEST_HEADERS "@rx <script>" id:101,msg: 'XSS Attack',severity:ERROR,deny,status:404
```

Based on @rx <script> part of the script, we can see that it checks script expression and if it matches it, then offers a severity level, in this case ERROR(3). Severity levels go from DEBUG(7) to EMERGENCY(0). Also, deny part terminates request and
returns 404 code to the individual who sent the request[12]. Of course, given example is from one of the WAFs and the structure and syntax of rules on different WAFs is not the same.

Although WAFs can and still are implemented locally, on a web server or as a web plugin, lately, popular solutions are based on cloud computing. Cloud based WAFs do not depend on any web server platform ran locally nor “care” about hardware configuration of the server. In this way, many web servers can be served by cloud based WAF. Also, another advantage is that WAF can “learn” from the attacks performed on one of the protected web servers and apply it as a general rule thus resulting in increased security[13].

WAF rule set can be based on so called “whitelist” and “blacklist”. If blacklist is implemented then everything is allowed except predefined rules that block certain requests and sets of characters. Most of the WAFs come with predefined blacklists that should be current with the security issues present at the time. If not, security consultant should create each of these rules and be sure all of them are covered. Since security vulnerabilities are emerging on a daily basis, it is difficult to keep track all the time and update these blacklists.

Whitelist on the other hand blocks everything and allows only certain requests or set of characters. If a security consultant knows exactly the behavior of a web application then he/she can create rules accordingly and allow only what the web app receives. This requires more knowledge of the system and web app on which WAF is running and is more difficult to set up compared to blacklist. But, overall, it does provide better security.

It is important to mention also that having WAF is a prerequisite in fulfilling and complying with security standards such as PCI DSS (Payment Card Industry Data Security Standard). That means that every website that deals with online shopping or similar needs to implement this solution in order to achieve some level of trustworthiness[4].
5 Implementation

For the purpose of this degree project, I will be writing my own tool for the Phase 2 described in 3.2 Method description. The script will be written in Python programming language. It will let a user choose which of the 3 WAFs he/she wants to test. All WAFs will have default set of rules based on blacklisting. Depending on the answer, the script will go into submenu where the user can choose what type of an attack he/she wants to conduct. The possibilities would be SQLi and XSS. More detailed explanation on the behavior of the script would be as follows:

- User is presented with choice to choose one of the 3 WAFs or exits the script (Figure 5.1).

```
Select firewall to attack
1 WAF 1
2 WAF 2
3 WAF 3
Select:
```

Figure 5.1.: WAF menu

- After choosing one of the WAFs, the script activates that specific function within the program related to the chosen WAF.
- User will have 4 choices on the following submenu, to turn off/on WAF, perform SQLi attack and perform XSS attack. The purpose of the WAF switch is to easily turn it off/on and quickly observe the behavior of the attacks on the WAF in both scenarios (Figure 5.2).

```
WAF: OFF
0 Set WAF 1
1 SQLi
2 XSS
3 Exit
Please select:
```

Figure 5.2.: Attack type menu

- After choosing one of the attacks, the method related to the chosen attack contains information about the IP address of the server and parses/visits the webpage with the malicious payload. For example http://IP_address/vulnerabilities/sqli/?id= + payload
- Payloads will be parsed from the text file saved on the user's computer with the each attack vector written in a separate row. Two text files need to exist - one for SQLi and the other for XSS attacks.
- These two text files would be used for all three WAFs. In this way, we can ensure that they are being tested properly and that the same attacks are conducted on all of them.
- The script checks the response of the website in order to decide if the attack succeeded or not. If the answer is the response WAFs give, then of course, the attack failed. Otherwise, the payload went through.
- After the attacks are finished, the result is shown with the amount of failed and successful attacks and the printout of the attacks that succeeded, if any.
6 Results

In total, 518 SQLi and 935 XSS payloads were gathered online (such as [14],[15]) and ran against web applications with firewalls turned off and on. Of course, since the web applications running on 3 servers were deliberately vulnerable, it is safe to say that 100% of the attacks went through and succeeded. Next step, as per method description, was to turn on the firewalls and run the experiment again. The results are shown in Table 6.1.

<table>
<thead>
<tr>
<th>WAF</th>
<th>SQLi Total</th>
<th>SQLi Successful</th>
<th>XSS Total</th>
<th>XSS Successful</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAF 1</td>
<td>518</td>
<td>48</td>
<td>935</td>
<td>0</td>
</tr>
<tr>
<td>WAF 2</td>
<td>518</td>
<td>15</td>
<td>935</td>
<td>0</td>
</tr>
<tr>
<td>WAF 3</td>
<td>518</td>
<td>21</td>
<td>935</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1.: Attacks summary

As seen from Table 6.1, 9.26% of SQLi attacks WAF 1 did not stop, 2.3% for WAF 2 and 4.05% for WAF 3 respectively. In case of XSS, 100% of attacks were stopped on all WAFs.

Since the success rate on XSS attacks was 0%, manual adjusting of payloads and devotion to each WAF was required in order to still try to bypass these WAFs. This approach was successful and I was able to bypass WAF 1 and WAF 3 with the following payload:

```
```

Even though there was an amount of SQLi payloads that these WAFs did not stop, it is still not concluded that these payloads are dangerous or exploitable. Each of the successful SQLi payloads still needs to be manually checked on all 3 web servers to confirm that they are indeed a security issue. Also, even though many types of payloads were applied, some are permutations of each other and it does not mean that different types of payloads succeeded. All of this will be discussed in the analysis chapter. The script and the successful payloads can be found in A Appendix 1.
7    Analysis

As mentioned in the previous chapter, manual checking of the successful payloads was needed. It is important to say that, in general, the amount of successful payloads is not important at all. It is the severity that matters even if only one attack was successful. That being said, we can now concentrate on each WAF and analyze the results separately. However, we will try to do a cross section of the successful SQLi payloads and draw conclusion if there are some that can be applied on all 3 WAFs.

7.1    WAF 1

Even though the script had the highest amount of successful attacks on this WAF, upon closer inspection, two can be picked out as severe.

Simple payload such as ‘or 1=1-- is blocked by WAF since it’s rules do not allow ‘ sign but interestingly enough payload ‘1 returns syntax error. This syntax error is the response of the DB, and if we keep in mind Table 4.1 where we see that the path is user/attacker -> WAF -> web app/db then we can conclude that the WAF was indeed bypassed. It did not react to our payload and we got a response from the DB. These responses are always sensitive in nature because they contain the information about the exception that occurred or something else that the DB is telling. From there, an attacker can modify the payload and extract sensitive information. Furthermore, even if the payload bypasses WAF, like it did in our case, it is WAFs job to stop the DB response which this WAF also failed to do.

Another payload which WAF did not recognize was 111' or _binary'1. Even though this WAF did block single quote sign, it seems that it did not block specifically this expression. First, as mentioned before, it did allow ‘ sign only if there is a number before. So by using a number before single quote sign, it was actually possible to use ‘ and thus close SQL expression and insert always true condition (_binary'1). Because of that, by executing this attack vector, web app/DB returned all the users including first and last names that exist in the DB. This is, most certainly, not acceptable. Simply put, WAF rules were missing a rule for forbidding combination of numbers with a single quote sign. On top of that, even though the WAF rules had a second security check of “or true” expression which was forbidden, it did not check for _binary’1 which also represents true condition.

7.2    WAF 2

When it comes to WAF 2, we have the same results but with different payloads. While ‘1 did not work, payload 1’1 did. The same DB syntax error was returned and thus, WAF 2 bypassed.

On the other hand, even though 111' or _binary'1 payload did not succeed on WAF 2 like it did on WAF 1, payload 1' or true# did. Again, list of all the users and all info was returned.

7.3    WAF 3

WAF 3 has shown itself to be the most secure when it comes to SQLi. It blocked all payloads previously successful with WAF 1 and 2. Further inspection of the attacks that managed to go through this WAF marked in the results section showed that none of these were malicious in nature and did not do anything severe. Also, no manual modification of these payloads helped to bypass this WAFs SQLi filter. While trying to find a way around, it has been noticed that, compared to WAF 1 and 2, this WAF 3 was blocking ‘ character no matter where or in what way in was written. Furthermore, it was
blocking characters like “::&... etc. This shows that even though results showed that 21 payload did go through, none of them was dangerous. Still, running the script, getting these results and analyzing them can help us in further research since we got results such as %55nion %53eLEct or unio%6e %73elect as successful. These payloads did bypass WAF 3 and even though they did not do any harm, it shows that we managed to write an SQL expression that was not allowed by this WAF. These two payloads represent UNION SELECT with some letters switched with URL encoding. So while UNION SELECT was blocked, %55nion %53eLEct or unio%6e %73elect were not. We can see from the results that expressions like above went through even though the original idea was to block them. Characters like @ % # or expressions like 1 or 1=1 are indeed allowed. That is why 21 successful attacks could help in deriving a dangerous attack. Unfortunately, I did not manage to manipulate payloads in such way to extract some data or response from DB, but theoretically, it is possible.

Comparing the results from all 3 WAFs it can be also concluded that there is no single exact payload that gives the same results on at least 2 WAFs. As shown, there are different payloads that manage to bypass WAF 1 and 3 and exploit in a same way, but it cannot be said that there is a “swiss knife” that can be applied on all WAFs. Concerning SQLi at least. It proves that the rules are made in-house.

However, manual XSS showed interesting results. This test was done with an example of a PHP webapp that gets an image from variable called “name” and it had the following coded:

`'img src="/" . $_GET["name"] --'`;

Somewhere else in the php code, there was a variable name that was pointing to a photo on the server. Visiting a page and seeing this photo gives this URL:

`http://IP_address/page.php?name=photo.jpg`

This is where these 3 WAFs came to a test. Since the web app was getting a photo in this way, it is possible to inject a different photo in the name variable such as:


Here, even though we are visiting the first web site, we would see an image from another web site. Testing this on all WAFs showed in fact that, while WAF 2 was blocking this behavior, WAF 1 and WAF 3 were vulnerable.

By exploiting this vulnerability, it was possible to steal a session cookie by creating a link such as:


With a bit of JavaScript, it was possible to inject a sequence of commands that send a cookie to an attacker's server. This payload was injecting an image in genuine server as shown before, but it was injecting with a nonexistent image on the other (attackers server) together with the value of victim's session cookie. Since there is no such image on attackers server, web server's access log will show information about the tried load of the image plus cookie value. Instead of having a really long link, malicious cookie_stealer.js script was created on the malicious server with the following content:
In short, by clicking on this kind of a link given to the victim by the attacker (Figure 7.1, Step 1.), a victim will go to a genuine web server (Figure 7.1., Step 2.) and open the content. At the time, since in the very same URL link there was a script injected, while visiting the genuine server script will try to load a photo that supposedly exits on the attacker’s web server together with cookie embedded (Figure 7.1., Step 3.). Since the photo is nonexistent on the attacker’s web server, it will report the request of the photo as an error in the attacker’s web server’s access log. That is not an issue considering that the script was made in this way – attach the stolen session cookie to the photo request that does not exist on the server. Since it does not exist, it will leave footprint in the logs and thus, attacker will get the session cookie (Figure 7.2).

As a small digression, I would like to add that “security=low” part of the Figure 7.2. is related to the web app and not WAF. As already explained, goal was to have unsecure web apps which are protected by WAF.

XSS part of the experiment showed that 2 out of 3 WAFs did not stop this kind of an attack and it can be said that this was the most severe attack of them all. Stealing a session cookie for an unprotected web site can be equaled to knowing a password, changing it later, or if not, doing a variety of malicious actions logged in as a victim.
8 Discussion

All of these WAFs came with predefined out-of-the-box set of rules. These rules are almost exclusively very strict. Nonetheless, it was possible to bypass WAFs to a certain degree. Although they were strict, as such, usefulness of these rules can be discussed.

In case of SQLi for WAF 3, it can be noticed that it blocks even quote sign alone. Bracket signs such as > or < are also not allowed on all of the WAFs. Question arouses: are rules like this plausible in everyday life and on a web application production level? In my opinion, no. Of course the end user will want to use single quote to search for a set of strings together or maybe actually look for a > sign somewhere on the website. Because of that, many administrators will want to adjust these rules.

We can look at the situation from two sides. First is leaving WAFs with default rules which results in strict rules for certain behavior but still vulnerable for other as shown in the results and the analysis sections. The other is to adjust the rules to specific needs which again leaves space for mistakes unknowingly left in the process. So it can be even said that it is a paradox - part of the problem in finding solutions to WAF vulnerabilities is actually trying to find a solution. Another very important aspect to look at is coding practices. Developing web application without a constant thought on security is like making a car without regard to human life. It is simply not acceptable. WAF can never know how the web application is programmed or where is it possible to inject malicious code in order to block it. Badly coded web application can nullify the effect of WAF. At the end, one can always turn to cloud based WAFs and since they “learn” from the attacks on certain web apps they can apply updated protection to all web apps under its wing. Of course, these solutions cost money.

In general, on a topic of the local WAFs tested in this report, it came as a surprise how relatively easy it was to bypass them. Of course, if one knows where to look for. Since they were bypassed, due to ethical reasons, their names had to be hidden and substituted with generic WAF 1/2/3.
9 Conclusion

This report explored web application vulnerabilities and tested the security of web application firewalls. The results were as general as they could be considering different WAFs. Same payloads were applied on all of them and objective results derived. Since it is in the nature of the firewalls to modify them and adjust to a certain needs, the results presented in this report showed the behavior of the firewalls with out-of-the box rules and, in my opinion, that shows the condition of the WAFs. Another alternative could maybe be testing security of already set up WAFs of some enterprises or similar. Problems with this approach are that it would be extremely difficult to find someone who would allow for these tests to be executed. Besides that, it is hard to imagine that there would be at least two different firewalls with exactly the same rules in place so that the reliability of tests could be satisfied.

Even though firewalls were different, logic behind the attacks and the same principle can be applied on all WAFs in general. Vulnerabilities remain and as this report showed, it is hard to see how they can be solved.

It can be concluded that there will always be difficulties in finding solutions for these vulnerabilities as long as there is a direct human factor involved. Humans make mistakes and in this case, it is really hard to think that errors will never be made during the programming or setting up firewall rules. Irresponsibility or “leave it, it is working now” mentality is still much present. As we are in the constant development and contributing to the evolution of the computer science as we speak, it remains to be seen where it leads us.

Looking back to the research questions, it can be said that they were answered. RQ1 and RQ2 were answered in theoretical part of this report, namely Chapter 3 and 4. Answer on RQ3 or how it is possible to bypass WAFs is shown in the results section together with analysis. Finally, the problems in finding solutions for these security issues (RQ4) are explained in discussion section.

9.1 Future work

As a future work continuation on exploring of the vulnerabilities would be interesting. It would be intriguing to “dig deeper” and see how many more malicious actions could be done, their scope and severity. At the same time, based on the findings of these actions research on the protection and how to improve WAFs would also be possible and beneficial. Another research that would be interesting is concerning defense mechanisms of cloud based WAFs and their learning capability.
References

A Appendix 1
Since the source code of the script is 353 lines long and payloads even more than that, the following can be found on my GitHub repository.

Script:
https://github.com/babylonQ/sqli-xss-tester/blob/master/sqli-xss-tester.py
SQLi payloads:
https://github.com/babylonQ/sqli-xss-tester/blob/master/sqlipayloads
XSS payloads:
https://github.com/babylonQ/sqli-xss-tester/blob/master/xsspayloads
Results:
https://github.com/babylonQ/sqli-xss-tester/blob/master/attacksfinished.txt