**Masters in Informatics Engineering**
Information Technology Security 2022/2023

# Third Assignment

Francisco Manuel Bandeira Carreira (2019222462)
mcarreira@student.dei.uc.pt
Gonçalo Tavares Antão Folhas Ferreira (2019214765)
tferreira@student.dei.uc.pt


# 1 Contextualization

Regarding the third assignment of this subject, we are asked to explore **web application security** and to implement a **web application firewall** that is able to protect the web application against application-layer attacks. The assignment is split into **two phases**. Taking this into account we decided to split this report into two major subsections: First and Second Scenario. We will also analyse some of the tests and attacks supplied in the *WSTG* documentation. The proposed architecture for both scenarios is presented below.

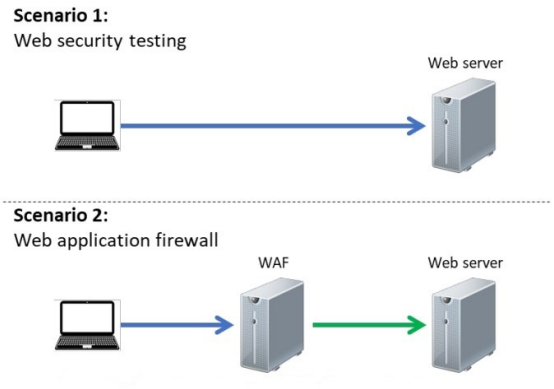



Figure 1: Proposed Architecture

# 2 First Scenario - Web Security Testing

In the first scenario we seek to analyse some security affairs regarding the **JuiceShop** website.

For the implementation of this scenario, due to some issues on the Kali Virtual Machine, we opted to instead use our own computer to run the attacks with **OWASP ZAP**. The issues faced on the virtual machine included heavy loading times and frequent crashes. The alternative we found was installing OWASP ZAP on our actual machine and executing the attacks through there. This proved efficient and significantly faster.

With this in mind, the architecture for this scenario includes 2 machines. Our *Centos 7* virtual machine that has the Juice Shop website running on *localhost:3000* and the host machine attacking the website.

With the architecture defined and OWASP ZAP up and running, we proceeded to execute the following penetration tests.

## 2.1 Automated Scan

An **Automated Scan** allows us to create a general overview on the JuiceShop website, detecting vulnerabilities and exploits. For this scan we used both the **Spider** and **AJAX Spider** (Chrome Headless) features. These crawlers identify pages on the website but according to the documentation, better results can be achieved through the combined use of both of them. The **Alerts** tab shows these problems in greater detail, showcasing the level of risk associated with each vulnerability. Some of the higher risk activities found include:

- Cloud Metadata Potentially Exposed
- SQL Injection - SQLite

- Content Security Policy (CSP) Header Not Set
- Cross-Domain Misconfiguration

After the Automated Scan was complete, we followed up with an **Active Scan** to highlight more exploits and find new alerts.

## 2.2 Active Scan - Website

This tool allows us to exploit the problems identified in the previous phase (Automated Scan). With this we can get further information about the detected alerts and where they apply. In order to see this, OWASP ZAP presents features able to auto-generate full reports containing detailed information regarding all the detected alerts.

However, upon doing research on the capabilities of the OWASP ZAP app, we found the **Add-ons tab** that showcases a market of add-ons that may come in handy in failure detection on a larger scale. Below, we can find the list of add-ons we decided to install from the **Add-On Market** in order to achieve better analysis results:

- FuzzDB Files
- FuzzDB Offensive
- Advanced SQLInjection Scanner
- Port Scanner
- Customer Payloads
- Image Location and Privacy Passive Scanner
- Token Generation and Analysis
- Access Control Testing
- Attack Surface Detector
- FileUpload
- Server-Sent Events

With this in mind, we ran the scans again with the new add-ons installed and these were the results.

**Summary of Alerts**

| Risk Level | Number of Alerts |
|---|---|
| Alto | 3 |
| Médio | 6 |
| Baixo | 5 |
| Informational | 4 |

Figure 2: Alerts Summary

| Name | Risk Level | Number of Instances |
|---|---|---|
| Advanced SQL Injection - AND boolean-based blind - WHERE or HAVING clause | Alto | 2 |
| Cloud Metadata Potentially Exposed | Alto | 2 |
| SQL Injection - SQLite | Alto | 1 |
| CSP: Wildcard Directive | Médio | 5 |
| Content Security Policy (CSP) Header Not Set | Médio | 1017 |
| Cross-Domain Misconfiguration | Médio | 932 |
| Missing Anti-clickjacking Header | Médio | 190 |
| Session ID in URL Rewrite | Médio | 546 |
| Vulnerable JS Library | Médio | 1 |
| Application Error Disclosure | Baixo | 4 |
| Cross-Domain JavaScript Source File Inclusion | Baixo | 1630 |
| Private IP Disclosure | Baixo | 1 |
| Timestamp Disclosure - Unix | Baixo | 5 |
| X-Content-Type-Options Header Missing | Baixo | 515 |
| Information Disclosure - Suspicious Comments | Informational | 5 |
| Modern Web Application | Informational | 816 |
| Retrieved from Cache | Informational | 186 |
| User Agent Fuzzer | Informational | 204 |

Figure 3: Found Alerts in Active Scan

By analysing the results, we can infer that the usage of more and better suited add-ons allowed us to identify more sources of problems, resulting in more alerts and more number of instances for the already identified alerts.

## 2.3 Fuzzing

According to the OWASP ZAP Documentation, Fuzzing is a technique based on submitting lots of invalid or unexpected data to a target. Our focus quickly became the login functionality as this would be ideal to test out some fuzzing attacks. Upon manual exploration on the Juice-Shop website, we found out that user emails are easily accessible to us by checking some product reviews. This means that we could use fuzzing to potentially discover user passwords with the help of add-ons such as *FuzzDB*. We first tried this approach with the used **jim@juice-sh.op** and this failed. However, this doesn't mean it won't work for other users. After many attempts with different users, this proved effective as the OWASP ZAP Fuzzer was able to identify a match for the user **admin@juice-sh.op** in a subset of over 37.7k attempts.

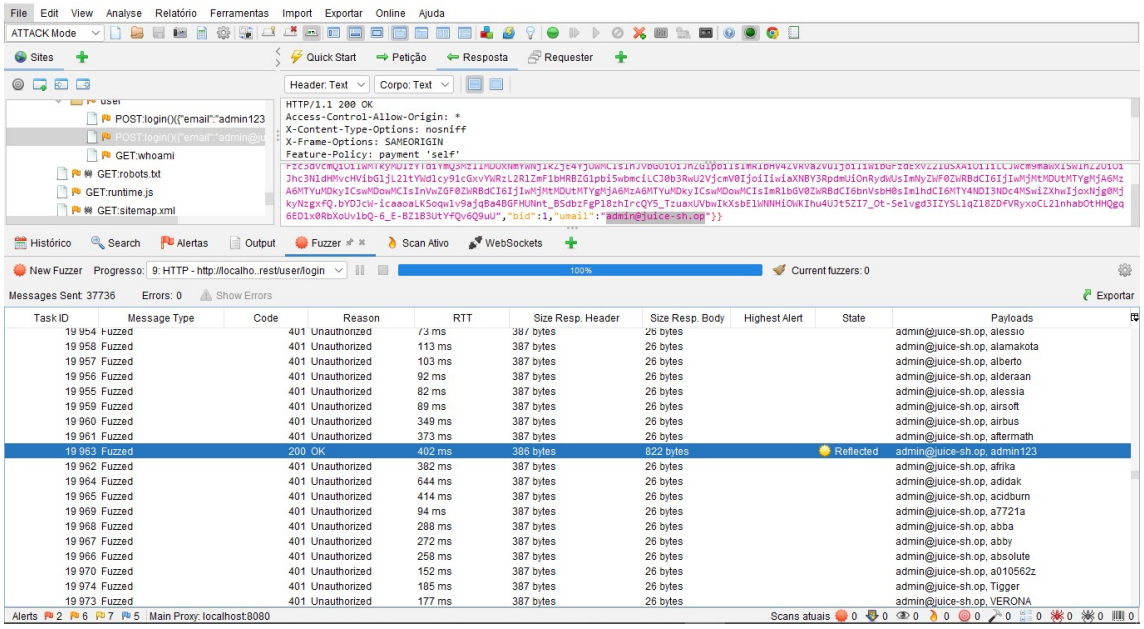



Figure 4: Password for an Admin user detected by the OWASP ZAP Fuzzer

Besides this attack, we also tried SQL Injections on the login view and some of these attempts were successful as seen below.

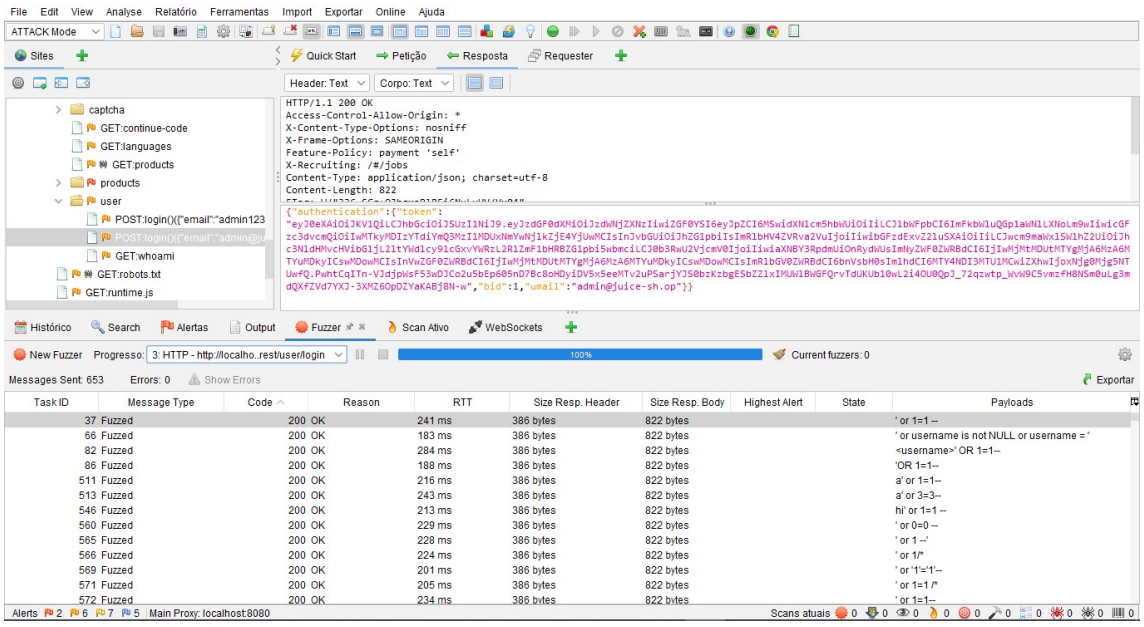



Figure 5: SQL Injections

## 2.4 Manual Penetration

As stated before, we did some manual analysis on the platform and found some interesting details. Towards the goal of analysing the authenticated area of the platform we ran a Manual Exploration where we accessed the website and navigated our way to the login view, accessing with the credentials identified in the Fuzz attacks. Once we were logged on, some product were added to the basket and purchased. Throughout this process we found a possible bug, as paying should not be possible in our conditions. The user's wallet didn't have enough money to cover for the expenses in the basket, yet this transaction was successful. With this in mind, we decided to aim our Active Scan towards the **cashout** section and check for possible attacks.

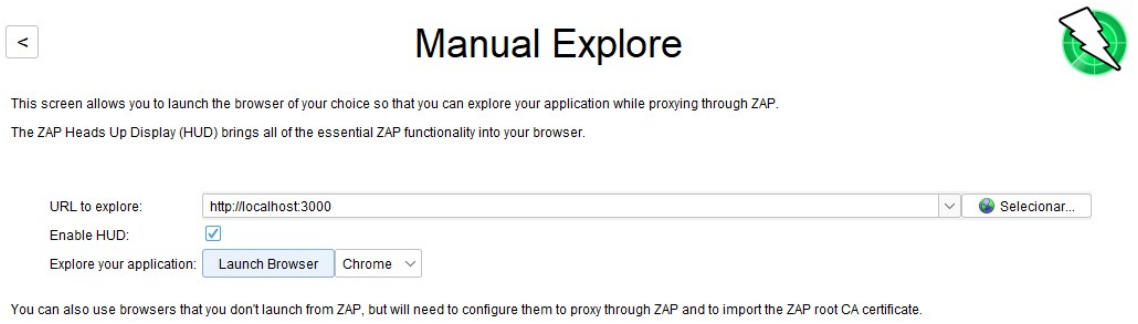



Figure 6: Manual Explore

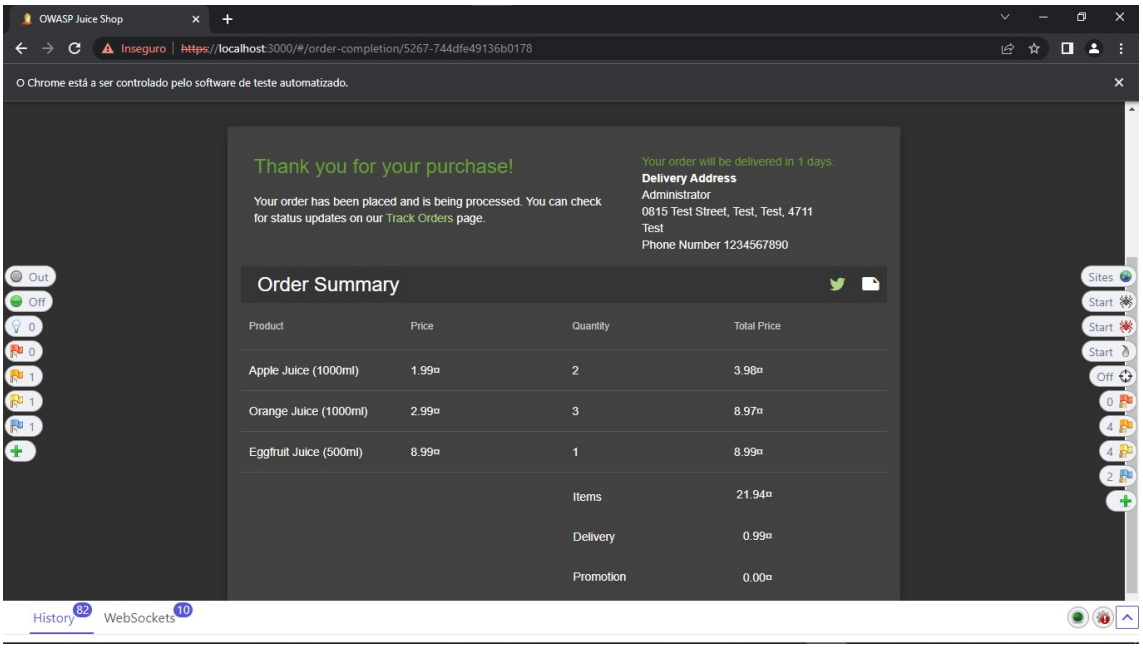



Figure 7: Successful Purchase

Another manual penetration test to explore vulnerabilities would be using SQL Injection in the login form.

### 2.5 Active Scan - Authenticated Area

By running an Active Scan through the **cashout** section, we were able to find some vulnerabilities beyond the ones detected previously, as these imply that the user is registered and logged in the platform.


Figure 8: Detected Alerts

As shown in the Figure 8, the main risks are associated with SQL Injections and Cloud Metadata being potentially exposed. The exploits are shown in greater detail on the auto-generated report attached to this in the submission file.

## 3 Web Application Security Testing Guidelines

When exploring the Juice Shop website found many attack possibilities. In this section, we present some of those findings and the conclusions achieved through their analysis, keeping in mind the WSTG guidelines and documentation.

### 3.1 Retrieving an Order

When retrieving an order request, we discovered that the **GET** request could sometimes retrieve files it was not supposed to. According to the OWASP ZAP, these files were marked as accessible or available and unauthorized access to them may lead to leaks of administrative information. The relevancy of this alert is dependant on the contents of the files. In the particular case that these files do not have any important information we can flag it as a **False Positive**.

## 3.2 Fingerprinting - Server Leaks Version Information via ”Server” HTTP Response Header Field

Fingerprinting is an information extraction technique that consists in discovering new data through internal components. Considering this we made an HTTP request (`http://localhost/HTTP`) and analysed the server response. Details such as server info, version, operating system and much more are shown. The risk associated with this attack reflects on preexisting vulnerabilities of tools that our system might be using. By releasing this information, any attacker could easily find exploits and attack our server. Figure 9 represents the details shown from the HTTP request.

```
HTTP/1.1 200 OK
Date: Sat, 20 May 2023 21:51:46 GMT
Server: Apache/2.4.6 (CentOS)
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
X-Recruiting: /#/jobs
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Sat, 20 May 2023 14:16:41 GMT
ETag: W/"7c3-18839850f75"
Content-Type: text/html; charset=UTF-8
Content-Length: 1987
Vary: Accept-Encoding
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
```

Figure 9: HTTP Request Response

## 3.3 Review Webserver Metafiles for Information Leakage

As stated in the respective WSTG documentation page, we can analyse the **robots.txt** file to unveil hidden paths which may lead to information leakage. In order to do this we first had to obtain the contents of the file itself and according to the documentation, a simple *curl* or *wget* does the job.

```
C:\Users\35192>curl -O -Ss http://localhost/robots.txt && head -n5 robots.txt
User-agent: *
Disallow: /ftp
C:\Users\35192>
```

Figure 10: Curling robots.txt file information

As shown above, the directory **ftp** is supposed to be private, which lead us into trying to reach this domain. Surprisingly, this was very easy, as simply changing the browser's path to `http://localhost/ftp` entered the private zone, without any kind of safety measures associated. This constitutes a severe flaw regarding information protection, as this is easily accessible to any user.

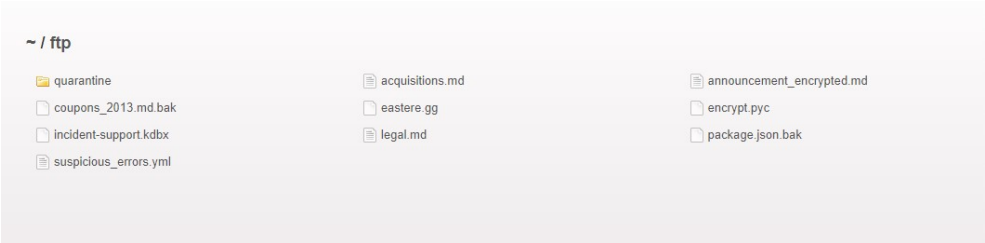



Figure 11: Private Directory

As expected, some of the files in this directory expose confidential information, making this test highly applicable for the Juice Shop server.

## 3.4 Review Webpage Comments and Metadata for Information Leakage

Another potential cause for information leaks are the comments on the source code. It's very common for programmers to include highly detailed comment sections associated to the code. These may

contain sensitive information about the system and how it works, proving to be a threat as attackers might take advantage of this. In OWASP ZAP, one of the identified alerts is connected to this problem. Although flagged as **informational**, the *Information Disclosure - Suspicious Comments* alert section shows 4 JavaScript files that may contain confidential information. Typically this kind of files may have private API keys, internal IP addresses, sensitive routes and even credentials. Assuming this, we can consider this threat applicable to the Juice Shop web server.

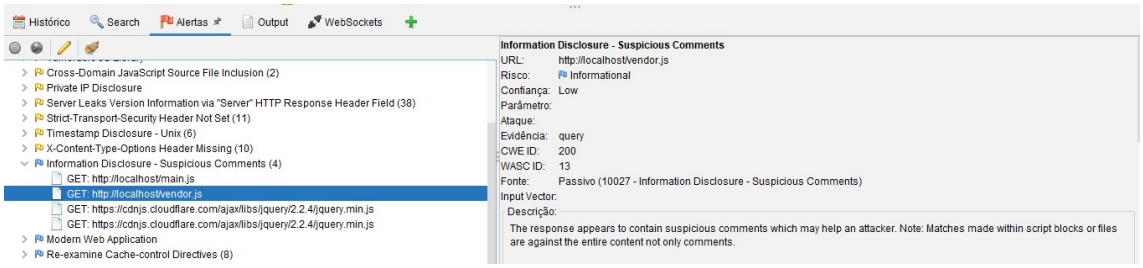



Figure 12: Suspicious Comments

## 3.5 Application Entry Points

The application's entry points are a vulnerable spot for attacks as the success of these might lead to accesses to private areas on the system. According to the documentation, looking through GET and POST requests can help us identify some of these risks and sure enough we managed to find a **Session ID** (found in the HTTP request header) associated with 2 POST requests and a GET request.


Figure 13: Session ID Leak in HTTP Request Header

This also popped up in the alerts tab as a Medium Risk, High Confidence alert.

## 3.6 HTTP Methods

According to the WSTG documentation, we can discover the allowed methods associated with the Juice Shop. We did this by running the command `curl -v -X OPTIONS http://localhost` and this was the result.

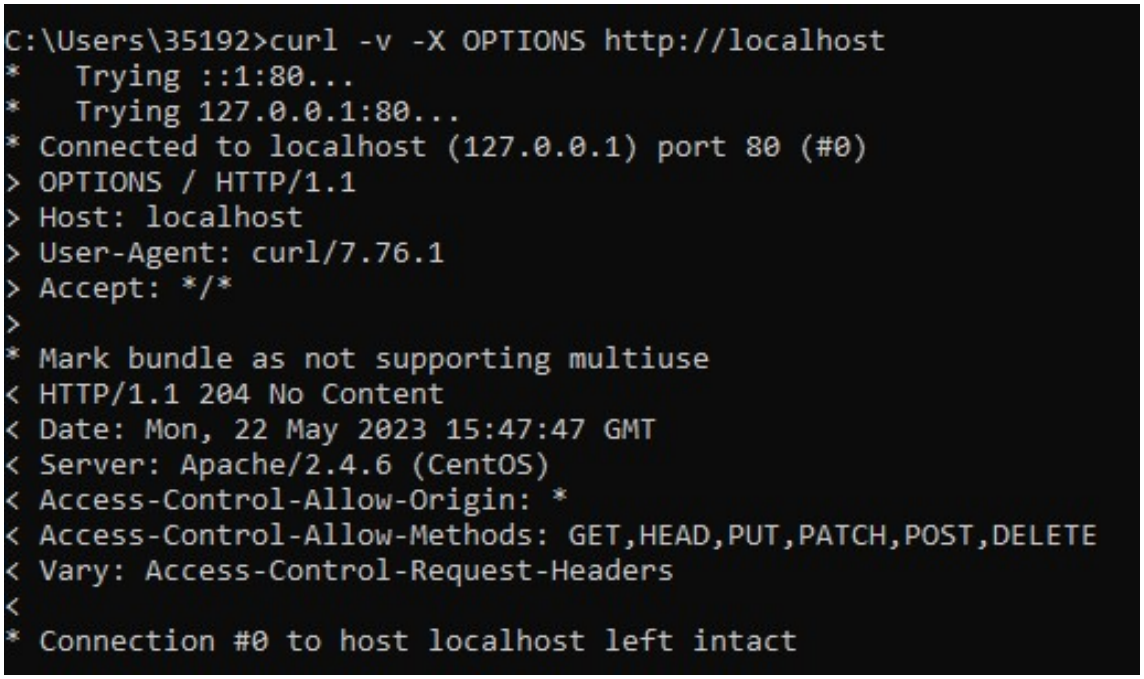



Figure 14: Allowed Methods

As shown, not every method is allowed, since CONNECT, PATH and OPTIONS are not present in that list. Besides this, we have to keep in mind the real risks of Access Control Bypassing (making requests with different methods may allow this is specific conditions) and HTTP Method Overriding (this can be achieved by using a custom header in the requests).

## 3.7 File Permission

Regarding this topic we can clearly say the Juice Shop website is vulnerable. As stated in a previous section, the endpoint `localhost/ftp` shows privileged information that should not be accessible to every user and even without being logged in. Because of this, we consider this important in the project context and identify it as a potential threat.

## 3.8 Subdomain Takeover

This attack ”allows an adversary to claim and take control of the victim’s subdomain” as stated in the documentation. However, due to our project working locally, we have no need for DNS, and so this threat is not applicable.

## 3.9 Cloud Storage

According to the documentation, cloud storage services provide a better way to store and access objects in the storage service. Despite this, just like in the section above, this threat also becomes irrelevant in the context of our project, as we are not using Cloud Storage.

## 3.10 User Registration Process

Regarding the user registration process, it is fundamental that the website requires enough details. Poor data collection may result in a more lackluster registration process that leads to a less secure registry. From the user’s point of view, the registration process should be simple and should not take a lot of time. This brings complications as fast registrations typically ask for fewer details, therefore

leading to an easier to break in registry. In our specific case, Juice Shop does have an extra parameter, this being a security question. However, nowadays this is far from enough to consider the process as safe. Besides this, there is also a concern regarding the enforcement of password criteria, as at the moment of testing this, the only requirement necessary is for the password to be 5+ characters long. Some advices are shown but not enforced. This idea will be further explored in another section.

## 3.11 Account Provisioning Process

The provisioning of an account presents many opportunities for attackers to create a valid profile without proper identification. In order to test this, we tried to create a profile without proper id, meaning, a fake email domain, for example, `example123@testing.com`. We soon realized that Juice Shop doesn't prevent the creation of users with fake email domains, as well as the creation of an account with a fake email. To prevent this, Juice Shop could provide a verification mechanism in order to activate the new account. After creating it, a user receives an email confirmation and once the user activates the account, it is finally fully created.

Since an account can be created with a fake email, we consider this test applicable in the context of the project, as it leads to vulnerabilities in the accesses to the system.

## 3.12 Account Enumeration and Guessable User Account

This strategy relies on verifying possible credentials combinations in order to access a user's account. This can be done with brute force. The problem associated with this is mainly due to the server's response, as trying to find out an email can lead to 2 different outcomes. Either the server accepts the email as valid option, meaning there is an account with that email associated, or the server returns invalid email, meaning the account does not exist. This can then be used to discover the password with a brute force approach.

## 3.13 Weak or Unenforced Username Policy

Much like the previous section, this test focuses on the login form, more specifically, testing how easy it would be to guess a username by using personal info, such as combinations of names and dates, etc. In our particular case, the login is made with emails and not usernames, meaning this might not be applicable. However, the same principle can be applied by maintaining fixed domains (ex. @gmail.com) and only changing the prefix.

## 3.14 Default Credentials

This point was explained in the Fuzz Attack section, where we discovered a user's credentials with a static email and a subset of over 37 000 default passwords, meaning it is a valid threat in the Juice Shop context.

## 3.15 Weak Lock Out Mechanism

A lock out mechanism prevents attackers from trying incessantly to access an account. This usually happens after a certain ammount of times and is sometimes accompanied by **captcha** mechanisms to verify the user is not a robot.

As previously seen, this is not the case in the Juice Shop application. Not only the captcha system but even the lock out system are not implemented in the web server. And so, in the context of our application, we consider this threat applicable and very serious.

## 3.16 Weak Passwords

When creating a new user, we realized the password requirements are very lax. Although there is a tab specifically designed to show advice when creating your password, these should be enforced in order to guarantee a reliable and safe password. Not enforcing these ”advices” means the existence of weak passwords is very likely to occur and puts at stake the unique access policy for an account as it can be easily broken into.


Figure 15: Juice Shop Weak Password Advices

## 3.17 Weak Security Question Answer

As discussed on previous topics, security questions provide an extra security layer when creating an account and as long as the questions are not very simple and basic, since this would make them vulnerable to all sort of attacks.

In the Juice Shop application there is a security question. However, the presented options are all standard making this ”extra layer of security” not that effective.

## 3.18 Privilege Escalation

This point refers to the ability of a user to access higher degrees of information than it should. This was already proven to be possible in the Juice Shop application, as shown in the Fuzzing section when we were able to access an administrator’s credentials.

## 3.19 Exposed Session Variables

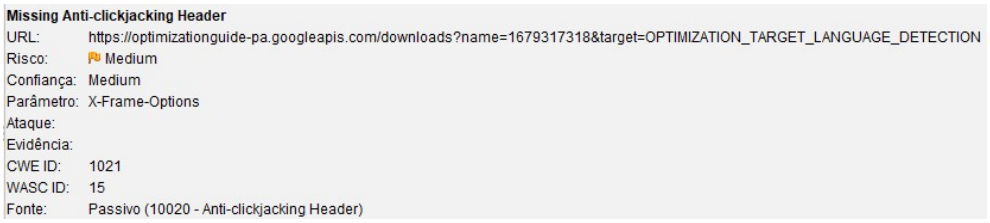
In our application, as shown before, it is possible to identify variables such as session identifiers. This is a severe vulnerability as any attacker can use this id to mimic and act as the original user without trace.

## 3.20 Logout Functionality

Regarding the logout functionality, we can check that it works as expected, clearing session variables from the platform, leaving no accessible information for attackers.

## 3.21 Clickjacking - Missing Anti-clickjacking Header

We also detected that the server does not have any anti-clickjacking protection. Also known as UI Redressing, clickjacking is a mean to trick user into clicking a button when they are actually clicking another completely unrelated one. This can be used to steal personal information, user credentials and even check other data such as purchase history and much more.



**Missing Anti-clickjacking Header**

| | |
|---|---|
| URL: | https://optimizationguide-pa.googleapis.com/downloads?name=1679317318&target=OPTIMIZATION_TARGET_LANGUAGE_DETECTION |
| Risco: | Medium |
| Confiança: | Medium |
| Parâmetro: | X-Frame-Options |
| Ataque: | |
| Evidência: | |
| CWE ID: | 1021 |
| WASC ID: | 15 |
| Fonte: | Passivo (10020 - Anti-clickjacking Header) |

Figure 16: Missing Anti-Clickjacking Header

# 4 Second Scenario - Web Application Firewall

Regarding the second scenario, we are asked to implement a **Web Application Firewall**, commonly known as WAF, in our architecture. In order for this to work, we installed **Mod Security** in our Centos 7 virtual machine and then relied on a reverse proxy approach. In order to achieve this we created a **juiceshop.conf** file that holds the configuration for the proxy as shown below.

```
<VirtualHost *:80>
        ProxyPass / http://localhost:3000/
        ProxyPassReverse / http://localhost:3000/
        SecRuleEngine ON
</VirtualHost>
```

Figure 17: Proxy Configuration File

We also commented the whole **welcome.conf** file as this conflicts with ModSecurity and prevents it from blocking requests. With this setup, we can now access the Juice Shop platform with and without proxy following the paths `http://localhost` and `http://localhost:3000` respectively. The port specification in the path implies we are not using the proxy approach, meaning that ModSecurity will not be detecting and blocking the requests. On the other hand, no port specification implies the use of *PORT 80* that is setup in a way that makes ModSecurity active and blocks incoming requests.

By running an Active Scan on the WAF scenario we were able to conclude its effectiveness, as although there were still alerts, the number and risk of these is severely lowered when compared to the first scenario.

The figures below illustrates this conclusion.

**Summary of Alerts**

| Risk Level | Number of Alerts |
|---|---|
| Alto | 0 |
| Médio | 4 |
| Baixo | 6 |
| Informational | 5 |

Figure 18: Alerts Summary

| Name | Risk Level | Number of Instances |
|---|---|---|
| Content Security Policy (CSP) Header Not Set | Médio | 14 |
| Cross-Domain Misconfiguration | Médio | 17 |
| Session ID in URL Rewrite | Médio | 16 |
| Vulnerable JS Library | Médio | 1 |
| Cross-Domain JavaScript Source File Inclusion | Baixo | 2 |
| Private IP Disclosure | Baixo | 1 |
| Server Leaks Version Information via "Server" HTTP Response Header Field | Baixo | 36 |
| Strict-Transport-Security Header Not Set | Baixo | 3 |
| Timestamp Disclosure - Unix | Baixo | 5 |
| X-Content-Type-Options Header Missing | Baixo | 9 |
| Information Disclosure - Suspicious Comments | Informational | 4 |
| Modern Web Application | Informational | 1 |
| Re-examine Cache-control Directives | Informational | 3 |
| Retrieved from Cache | Informational | 3 |
| User Agent Fuzzer | Informational | 96 |

Figure 19: Found Alerts in Active Scan

To further acknowledge the effectiveness of the Mod Security System, we also made a manual analysis on the **error logs** with the intent of exploring and validating the stoppage of attacks.

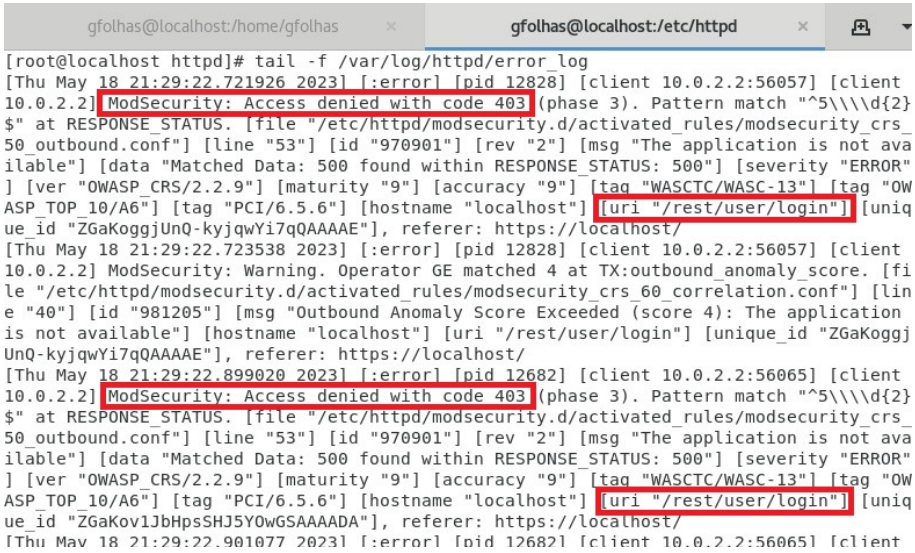
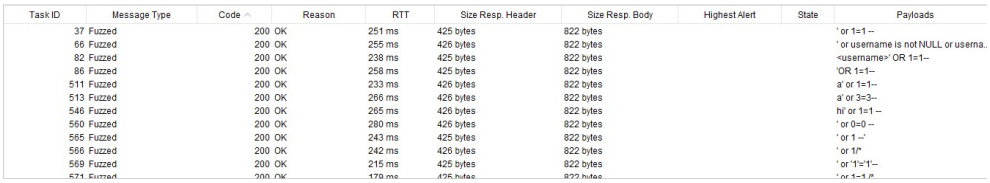



Figure 20: Error Logs

Figure 20 illustrates an attack on the login form, showing 2 attacks that resolutely came out as *Code 403, Access Denied*. This is just an example but several logs were checked in the making of this analysis and the conclusion was that Mod Security is able to improve the security of our web server, detecting and blocking attacks that without it would most likely pass with *Code 200, Accepted*, exploiting the server's resources and functionalities.

We also noticed that SQL Injections in Fuzz attacks are still a threat even with the Firewall active.



| Task ID | Message Type | Code | Reason | RTT | Size Resp. Header | Size Resp. Body | Highest Alert | State | Payloads |
|---|---|---|---|---|---|---|---|---|---|
| 37 | Fuzzed | 200 | OK | 251 ms | 425 bytes | 822 bytes | | | ' or 1=1 -- |
| 66 | Fuzzed | 200 | OK | 255 ms | 426 bytes | 822 bytes | | | ' or username is not NULL or userna.. |
| 82 | Fuzzed | 200 | OK | 238 ms | 425 bytes | 822 bytes | | | <username>' OR 1=1-- |
| 86 | Fuzzed | 200 | OK | 258 ms | 425 bytes | 822 bytes | | | 'OR 1=1-- |
| 511 | Fuzzed | 200 | OK | 233 ms | 426 bytes | 822 bytes | | | a' or 1=1-- |
| 513 | Fuzzed | 200 | OK | 266 ms | 426 bytes | 822 bytes | | | a' or 3=3-- |
| 546 | Fuzzed | 200 | OK | 265 ms | 426 bytes | 822 bytes | | | hi' or 1=1 -- |
| 560 | Fuzzed | 200 | OK | 280 ms | 426 bytes | 822 bytes | | | ' or 0=0 -- |
| 565 | Fuzzed | 200 | OK | 243 ms | 425 bytes | 822 bytes | | | ' or 1 --' |
| 566 | Fuzzed | 200 | OK | 242 ms | 426 bytes | 822 bytes | | | ' or 1/* |
| 569 | Fuzzed | 200 | OK | 215 ms | 425 bytes | 822 bytes | | | ' or '1'='1'-- |
| 571 | Fuzzed | 200 | OK | 179 ms | 425 bytes | 822 bytes | | | ' or 1=1/* |

Figure 21: Fuzz Attack (SQL Injections)

With this in mind we can conclude that even though the Firewall makes an extreme difference in detecting and blocking incoming threats, it is still penetrable and the website is still very much vulnerable to a certain group of threats.

# 5 Final Considerations

We believe the main goals proposed in this project were achieved successfully. As shown in the previous sections, we were able to detect vulnerabilities in the Juice Shop website, take them into consideration for further studies and eventually make use of a WAF to minimze the ammount of attacks that were successfully attempted. As discussed before, this means that some attacks do still proceed without blocking, however the degree of risk associated to these attacks is lower than the ones detected without firewall. The tests were thoroughly executed and analysed and we believe the report contains all the relevant information regarding the detected vulnerabilities of the platform, as well as the intention of the attacks performed and the effectiveness of the Web Application Firewall.

# References


[1] "OWASP ZAP Documentation". [Online]. Disponível em: https://https://www.zaproxy.org/

[2] "WSTG Guidelines". [Online]. Disponível em: https://owasp.org/www-project-web-security-testing-guide/

[3] "WSTG Security Testing". [Online]. Disponível em: https://github.com/OWASP/wstg/tree/master/document/4-Web_Application_Security_Testing

[4] "OWASP ZAP Core Project". [Online]. Disponível em: https://github.com/zaproxy/zaproxy