1 2  9 0 UNIVERSIDADE D COIMBRA

Practical Assignment 2

Information Tecnology Security

Alexy de Almeida	2019192123	adenis@student.dei.uc.pt
Rodrigo Ferreira	2019220060	rferreira@student.dei.uc.pt

PL3

April 2023

Contents

1 Introduction

The aim of this project is to explore the Web Security Testing Guide (WSTG) and to configure and explore the usage of ModSecurity reverse proxy as a Web Application Firewall (WAF). The project is divided in tho phases: the first phase is dedicated to exploring the JuiceShop security and the second phase is dedicated to monitor, filter and block HTTP traffic to the JuiceShop by using ModSecurity WAF. In other words, the first phase aims to explore the security of a web application and the second phase aims to address the security issues explored in the first phase.

2 Experimental scenario

For this assignment, only one virtual machine was used, called **Kali_OWASP**. In there we're be running Kali distribution, the JuiceShop and OWASP ZAP software. The reason we opted for a single VM was the inability to do the second phase of the project and to run the tests way faster, as having 2 separate VMs would make the process slower.

JuiceShop is a web application that contains several hacking challenges to be exploited as to detect underlying vulnerabilities, which can be found by using OWASP ZAP, a software that detects vulnerabilities in web applications.

2.1 Initial Configurations

```
# download OWASP Juice Shop
sudo wget https://github.com/juice-shop/juice-shop/releases/download/v14.5.1/juice-shop-14.5.1
    _node14_linux_x64.tgz
tar zxvf juice-shop-14.5.1_node14_linux_x64.tgz

# install NodeJs and NPM
sudo wget https://nodejs.org/download/release/v20.2.0/node-v20.2.0-linux-x64.tar.gz
tar zxvf node-v20.2.0-linux-x64.tar.gz

sudo cp -r node-v20.2.0-linux-x64/{bin,include,lib,share} /usr/
```

In order to verify if everything is working do the following commands:

```
node --version
npm --version

# expected output
14.5.0
6.14.5
```

3 First Phase - Web Application Security Testing

We begin by starting the 'juice-shop', which can be achieved with the following commands:

```
cd juice-shop-14.5.1_node14_linux_x64
npm install
npm start
```

After that, we run the program OWASP ZAP program and install the following add-ons:

- Active Scanner Rules, version 46.0.0
- Advanced SQLinjection
- FileUpload
- FuzzDB Files
- FuzzDB Offensive
- Sequence
- Wappalyzer

3.1 Automated scan to the website

The goal of an automated scan is to let the software automatically detect several vulnerabilities, each having a risk that goes from Informational to High.

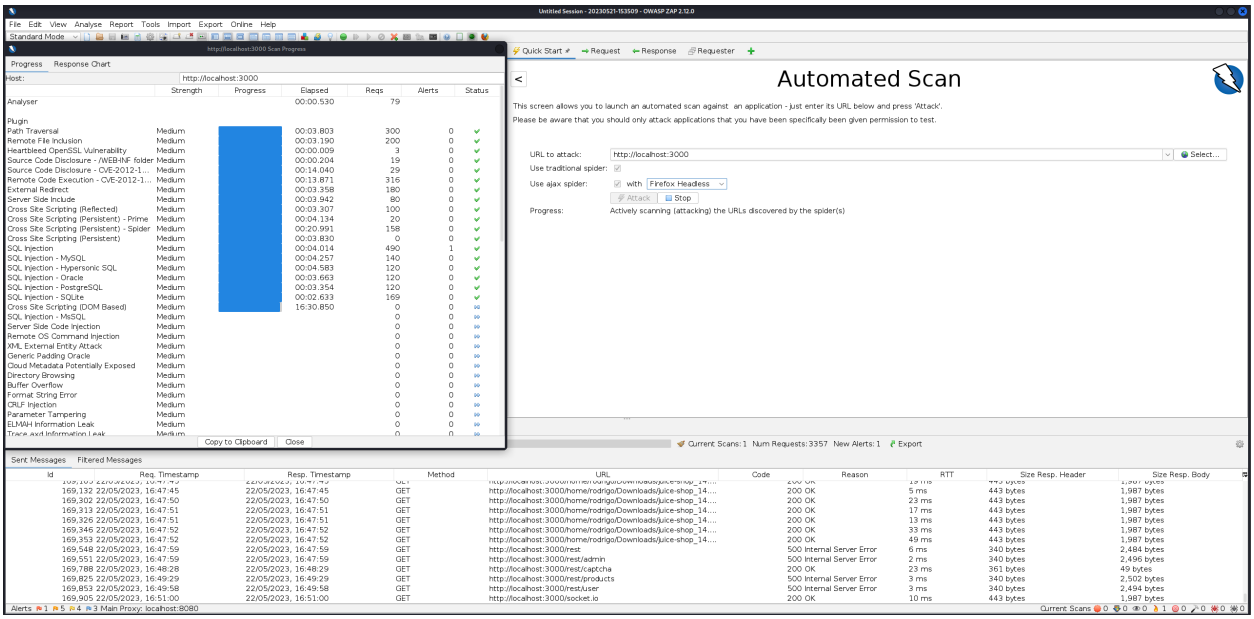

Figure 1: Automatic Scan

To run an automated scan, do the following:

- URL to attack: http://localhost:3000

☒ Use traditional spider

☒ Use Ajax spider

We obtain the following results:

Risk	Alerts
High	3
Medium	8
Low	4
Informational	5

Name	Level	Instances
Advanced SQL Injection - AND boolean-absed blind - WHERE or HAVING clause	High	1
Cloud Metadata Potentially Exposed	High	1
SQL Injection - SQLite	High	1
Backup File Disclosure	Medium	31
Bypassing 403	Medium	5
CORS Misconfiguration	Medium	153
Content Security Policy (CSP) Header Not Set	Medium	113
Cross-Domain Misconfiguration	Medium	114
Missing Anti-clickjacking Header	Medium	43
Session ID in URL Rewrite	Medium	166
Vulnerable JS Library	Medium	1
Cross-Domain JavaScript Source File Inclusion	Low	124
Private IP Disclosure	Low	1
Timestamp Disclosure - Unix	Low	5
X-Content-Type-Options Header Missing	Low	166
Cookie Slack Detector	Informational	30
Information Disclosure - Suspicious Comments	Informational	5
Modern Web Application	Informational	63
Retrieved from Cache	Informational	24
User Agent Fuzzer	Informational	145

3.2 Active scan to the website

To perform a more complete scan to the website, we can make use of the "Active Scan" feature by assigning several policies, each with their own threshold and strength. We defined the policies as follows:

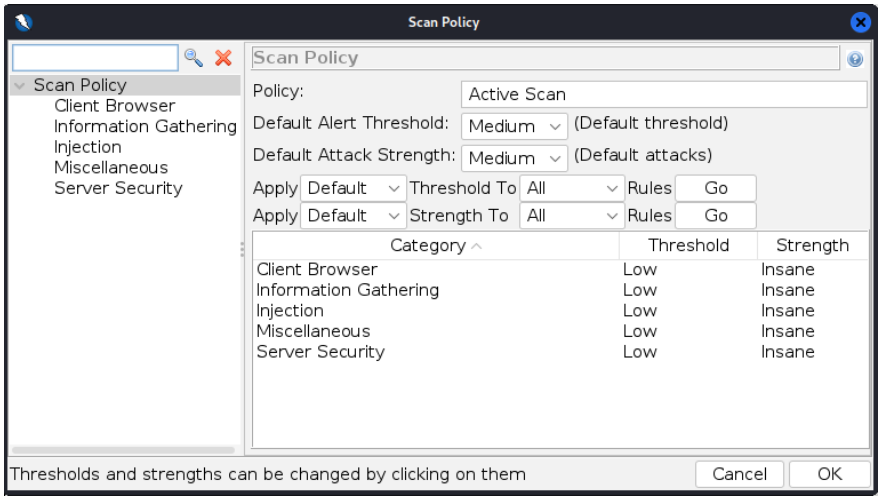

Figure 2: Active Scan - Options

After running the active scan, we obtain the following results:

Risk	Alerts
High	4
Medium	4
Low	0
Informational	5

Name	Level	Instances
Advanced SQL Injection - AND boolean-based blind - WHERE or HAVING clause	High	2
Advanced SQL Injection - AND boolean-based blind (Generic comment)	High	2
Cloud Metadata Potentially Exposed	High	1
SQL Injection - SQLite	High	2
Backup File Disclosure	Medium	233
ELMAH Information Leak	Medium	1
HTTP Only Site	Medium	1
Source Code Disclosure - SVN	Medium	59
.env Information Leak	Informational	3
.htaccess Information Leak	Informational	3
Cookie Slack Detector	Informational	27
Trace.axd Information Leak	Informational	3
User Agent Fuzzer	Informational	162

We got different results from the automated scan and detected less alerts compared to using "Default Policy", but we also found 3 high level alerts related to SQL injection.

3.3 Fuzz scan

To further detect more vulnerabilities relted to user interaction, we made use of the ZAP fuzzer functionality, which allows us to make attacks on the login, username or email, which allows us to retrieve user passwords.

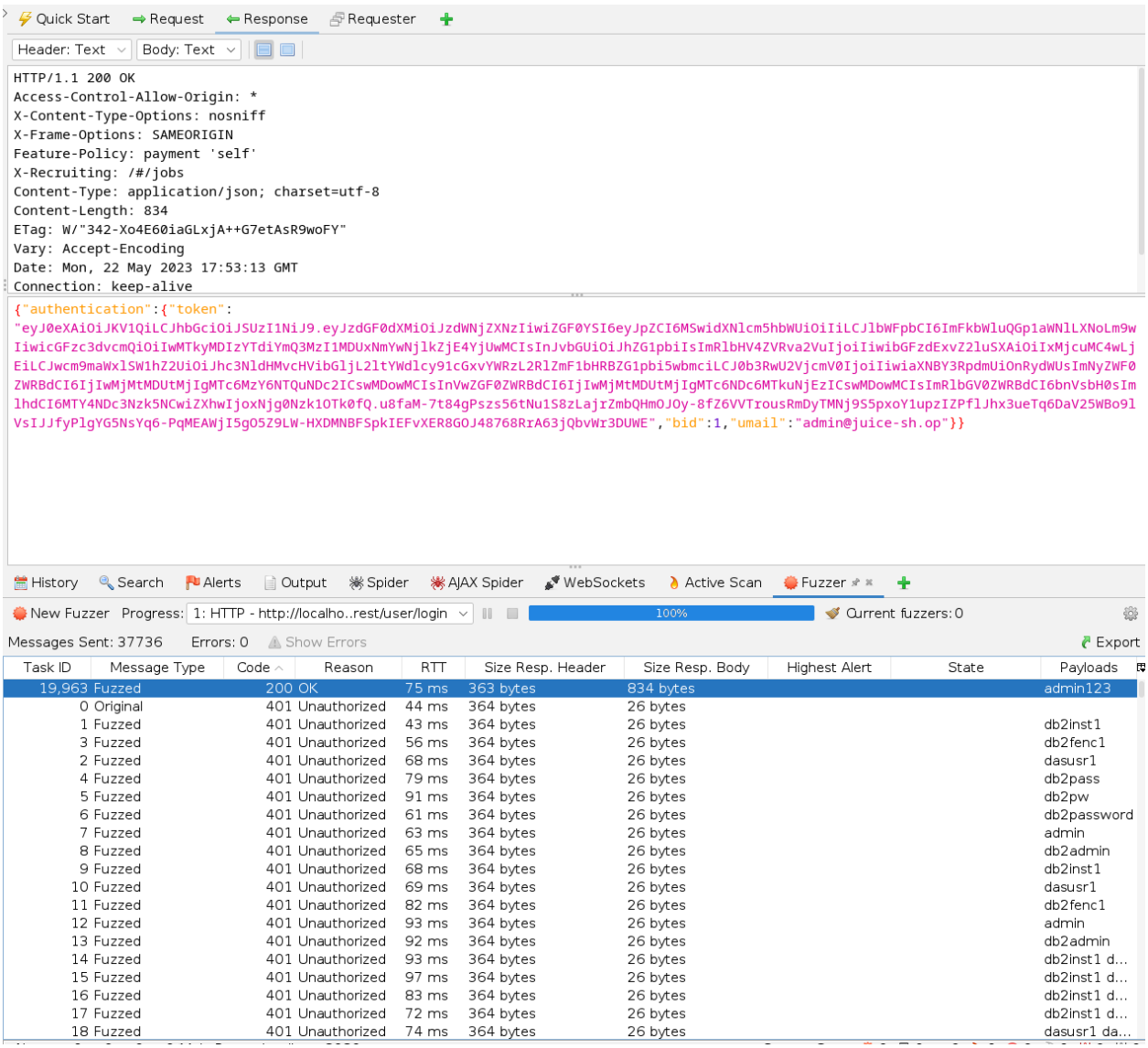

Figure 3: Admin password found using Fuzz

3.4 Manual testing

After doing different types of scans to the website, we obtained several vulnerabilities that can be explored. Every vulnerability has its own solutions and they can be found on the OWASP Web Security Testing Guide page. Below we're presenting the list of vulnerabilities that we found and we're going to explain them separately:

ID	Alert
WSTG_INFO_02	Fingerprint Web Server
WSTG_INFO_03	Review Webserver Metafiles for Information Leakage
WSTG_CONF_01	Test Network Infrastructure Configuration
WSTG_CONF_04	Review Old Backup and Unreferenced Files for Sensitive Information
WSTG_IDNT_04	Testing for Account Enumeration and Guessable User Account
WSTG_ATHN_02	Testing for Default Credentials
WSTG_ATHN_04	Testing for Bypassing Authentication Schema
WSTG_ATHZ_02	Testing for Bypassing Authorization Schema
WSTG_SESS_02	Testing for Cookies Attributes
WSTG_SESS_04	Testing for Exposed Session Variables
WSTG_CLNT_03	Testing for HTML Injection

3.4.1 WSTG_INFO_02 - Fingerprint Web Server

The goal of this test is to determine the version and type of a running web server, which can be useful when a particular framework in a specific version shows a vulnerability, so the web app is automatically vulnerable as well. For that, we made an HTTP request (***GET / HTTP/ 1.1***) using netcat.

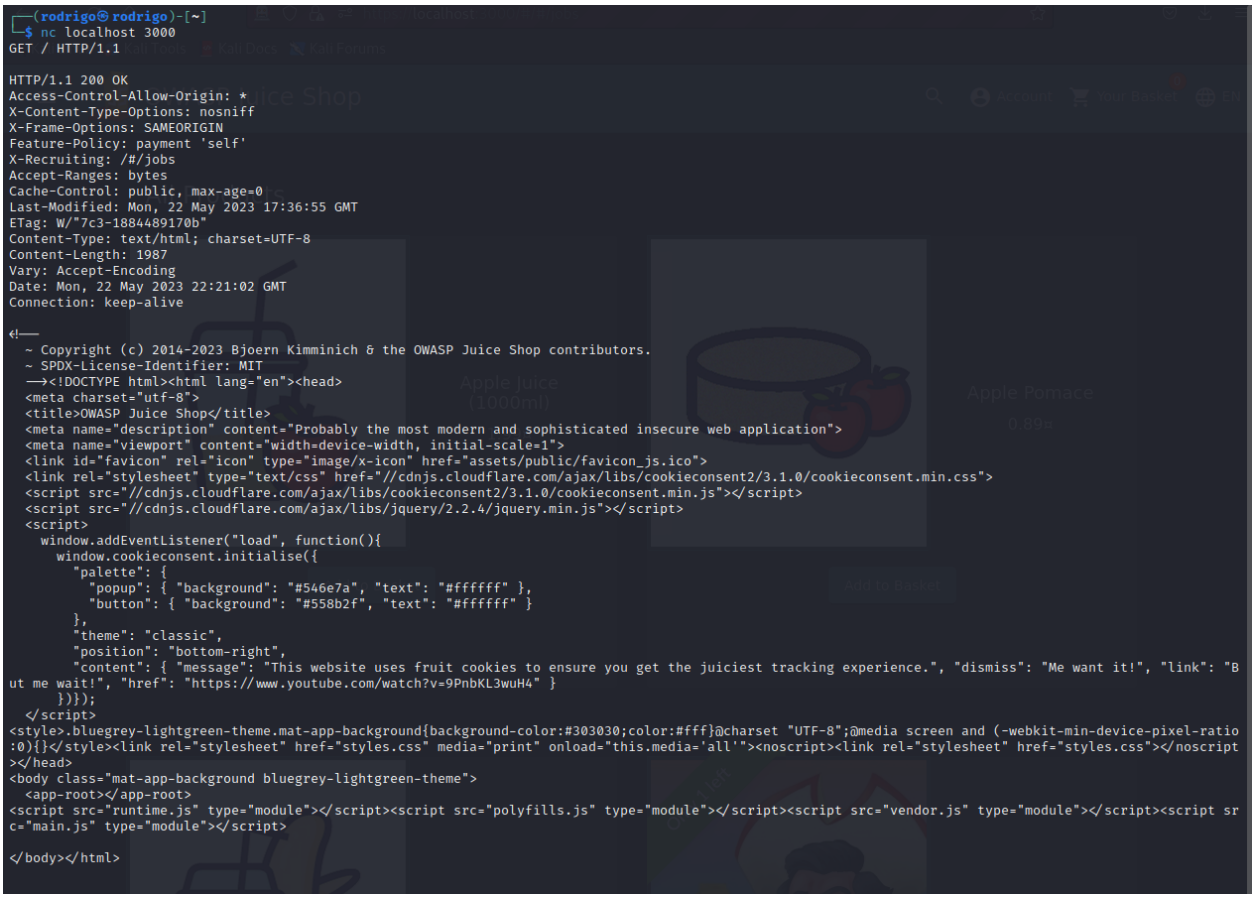

Figure 4: Fingerprinting of Juice Shop

From the output received it's not possible to extract much information about the inner works of the web server, so the fingerprinting was not very useful.

3.4.2 WSTG_INFO_03 - Review Webserver Metafiles for Information Leakage

The goal of this test is to identify hidden or obfuscated paths and functionality through the analysis of metadata files, as well as extract and map other information that could lead to better understanding of the systems at hand. To test the metadata files for information leakage of the web application we checked two things:

- Access the folder **ftp** through the URL (***localhost:3000/ftp***). The access was not blocked and allowed us to view private files and folders.

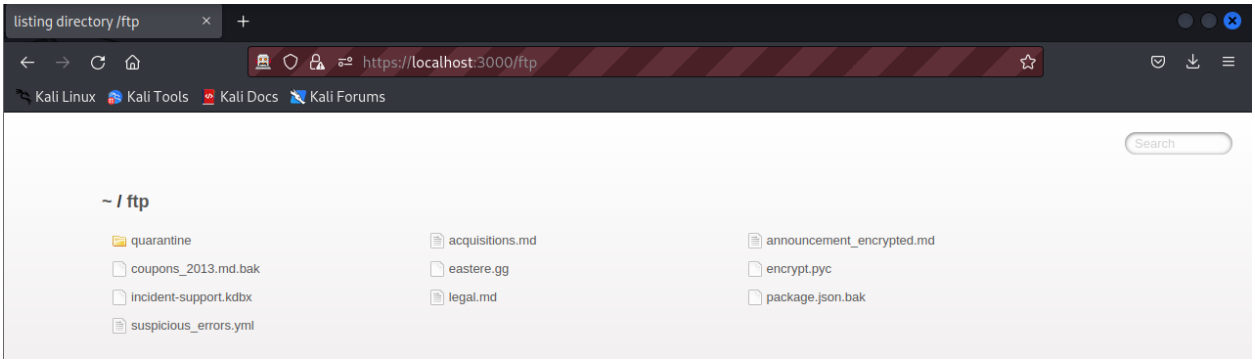

Figure 5: Access to hidden paths.

- View the file **robots.txt** through the URL (***localhost:3000/robots.txt***) or download the file in the terminal using (***wget localhost:3000/robots.txt***). Once again the access was not blocked and we get access to hidden paths and functionalities.

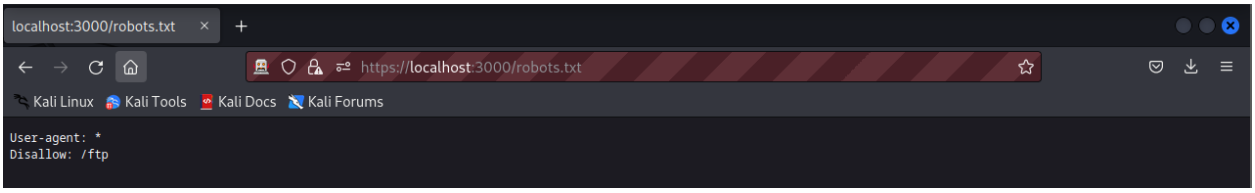

Figure 6: Contents of robots.txt file

3.4.3 WSTG_CONF_01 - Test Network Infrastructure Configuration

The goal of this test is to review the application's configurations set across the network and validate that they are not vulnerable. In addition, we also want to validate that used frameworks and systems are secure and not susceptible to known vulnerabilities due to unmaintained software or default settings and credentials.

For that we check the URL (***localhost:3000/metrics***). As it's possible to view in the figure below we have access to the configurations of the web server.

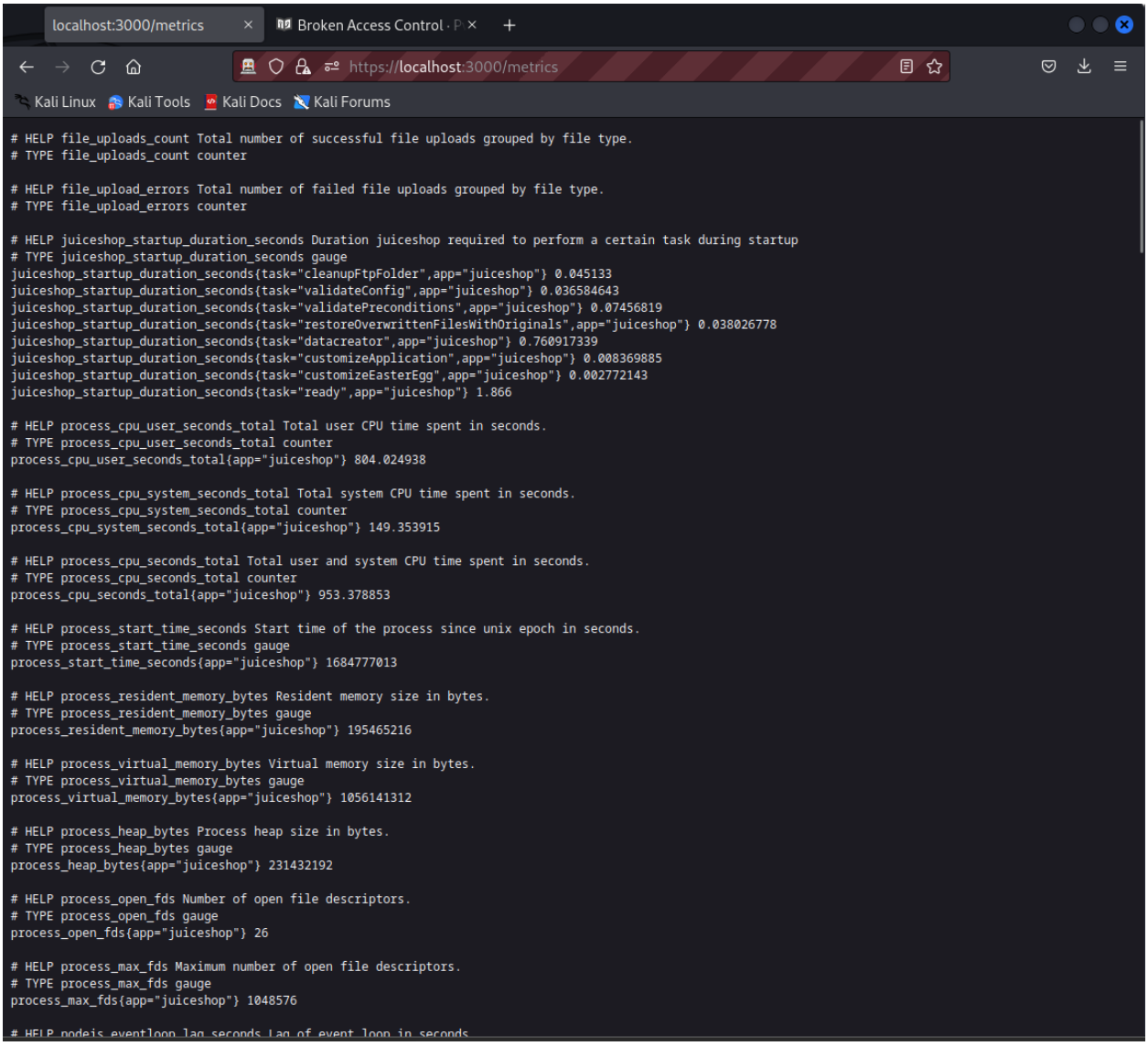

Figure 7: Metrics of Juice Shop

3.4.4 WSTG_CONF_04 - Review Old Backup and Unreferenced Files for Sensitive Information

The goal of this test is to find backup and unreferenced files, which can give information to hackers about the web app infrastructure, alterations it suffered and other sensitive information. These files are created automatically and common file types are *.old* or *.bak*, but others can be also used depending on the editor.

To do that, we have access to the ftp URL using (***localhost:3000/ftp***), inside of which there are 2 *.bak* files. Using ***wget http://localhost:3000/ftp/coupons_2013.md.bak*** we can't obtain the file, but using a Null Byte Injection it's possible do download it. This works by adding ***%25%30%30.md*** at the end of the URL, as this will be translated to ***%00*** (NULL character). If the web server does not handle the NULL character correctly, then we should gain access to the file. The figure below shows the results.

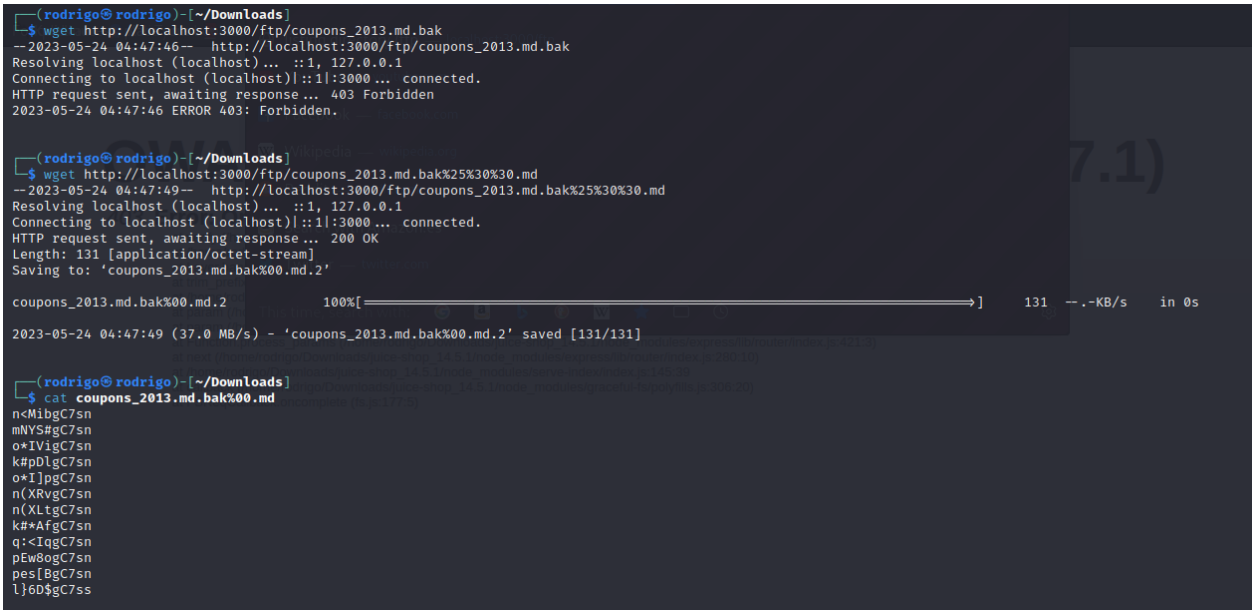

Figure 8: Access to a backup file

3.4.5 WSTG_IDNT_04 - Testing for Account Enumeration and Guessable User Account

The goal of this test is to review processes that pertain to user identification, such as the login and register page of the web server, and to test common authentication credentials, such as *admin*, *user1*, etc. Inside the web app, we have access to the reviews and the emails respective to the user that reviewed the content. With that information, we can use brute force to test different combinations of username and password, and try to guess the password associated to the user.

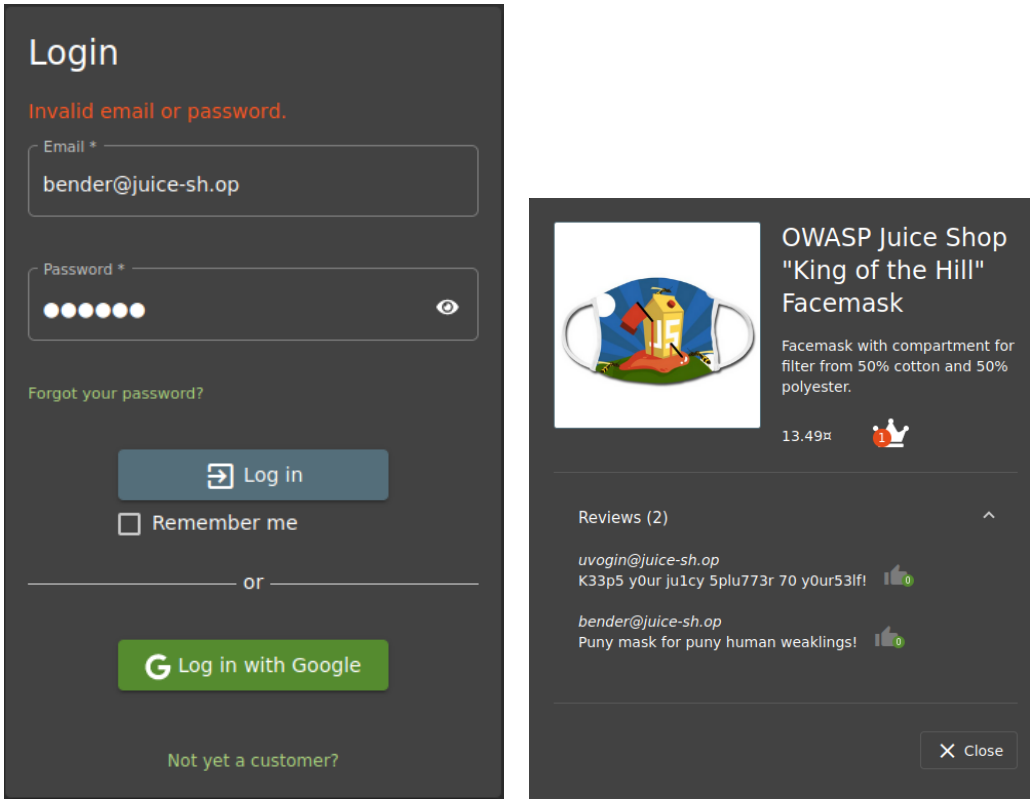

Figure 9: Attempt of obtaining the users credentials by analysing public data

3.4.6 WSTG_ATHN_02 - Testing for Default Credentials

The goal of this test is to look for default credentials and validate if they still exist. In addition, we also want to review and assess new user accounts and if they are created with any defaults or identifiable patterns. To do that, we will use a fuzz scan for common usernames with common passwords.

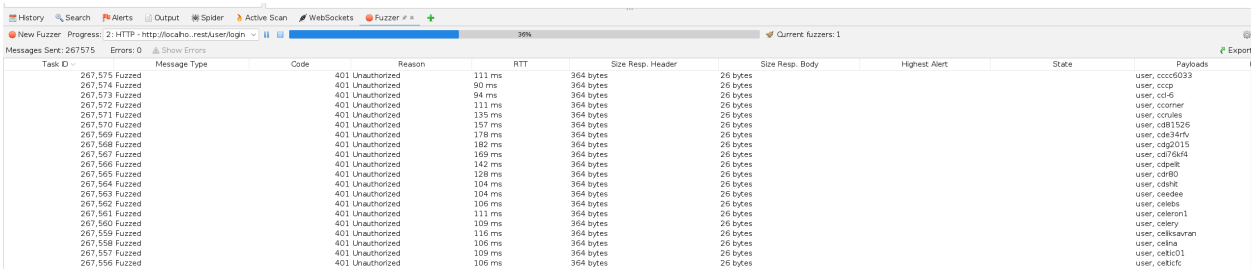

Figure 10: Fuzzer looking for default credentials

3.4.7 WSTG_ATHN_04 - Testing for Bypassing Authentication Schema

The goal of this test is to bypass the authentication security mechanisms and gain access to several pages that can only be accessed with a login. We began by using direct URLs but it wasn't a very successful method. Then, we proceeded to using SQL injection by adding ***' or 1 = 1–*** at the end of the email in the login page. The figures below demonstrate the SQL injection and the account access gained.

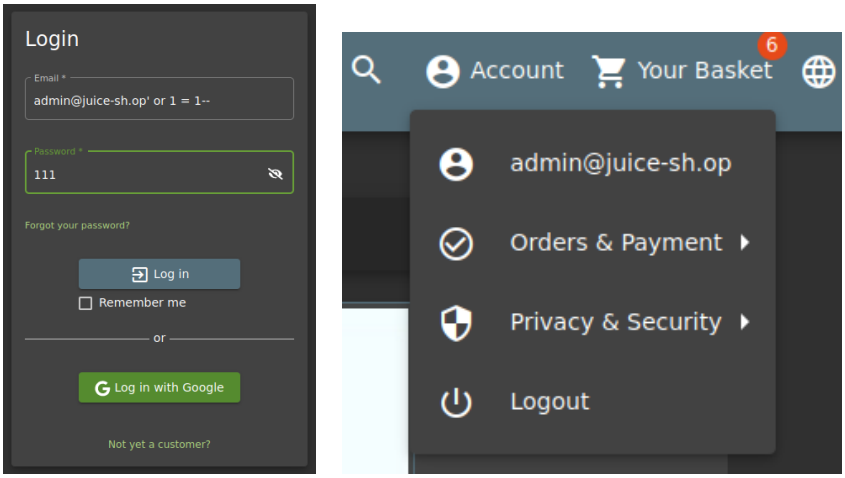

Figure 11: Access to user account using SQL injection

3.4.8 WSTG_ATHZ_02 - Testing for Bypassing Authorization Schema

The goal of this test is to bypass the authorization schema, allowing us to access private information or functionalities. We found one possible situation where it's possible to bypass the authorization schema and view other users baskets, which could be done by opening the *inspect view* on our basket and, in the *storage* tab, by finding the variable ***bid***. By changing its value and reloading the page, it's possible to access other users' baskets.

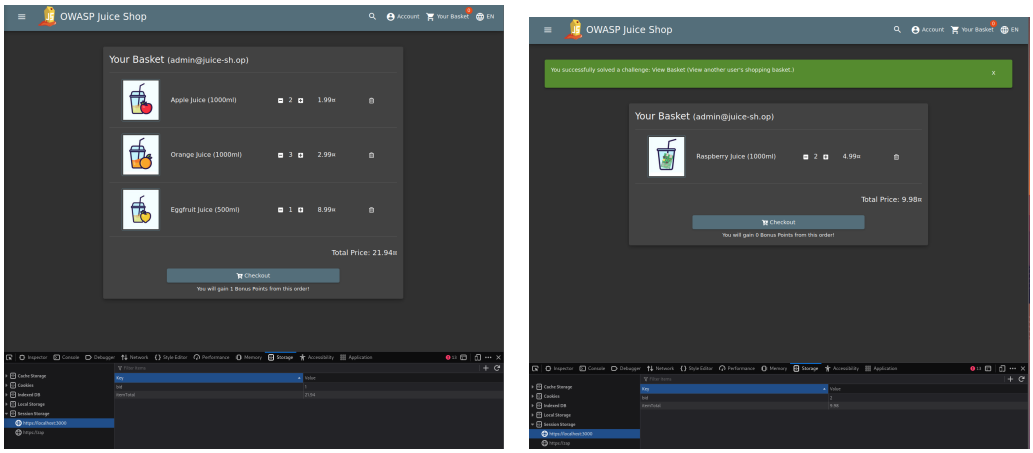

Figure 12: View of other users baskets

3.4.9 WSTG_SESS_02 - Testing for Cookies Attributes

The goal of this test is to ensure that the proper security configuration is set for cookies. Web cookies are important in websites since they use HTTP, as it's a stateless protocol. Those cookies are often targeted because they may hold important information and it's necessary to make sure they are properly configured and in this test we want to check the parameters of the cookies. The figure below shows the cookies that the **Juice Shop** uses. Some of the parameters are not properly configured, for example, ***HttpOnly*** is set to false but it should be set to true in order to prevent attacks, such as session leakage. Another one is the ***Path*** parameter, which is set to root (/) in all cookies and this may cause security vulnerabilities. Additionally, the ***Secure*** parameter is set to false because we were using **HTTPS** and it should be set to true.

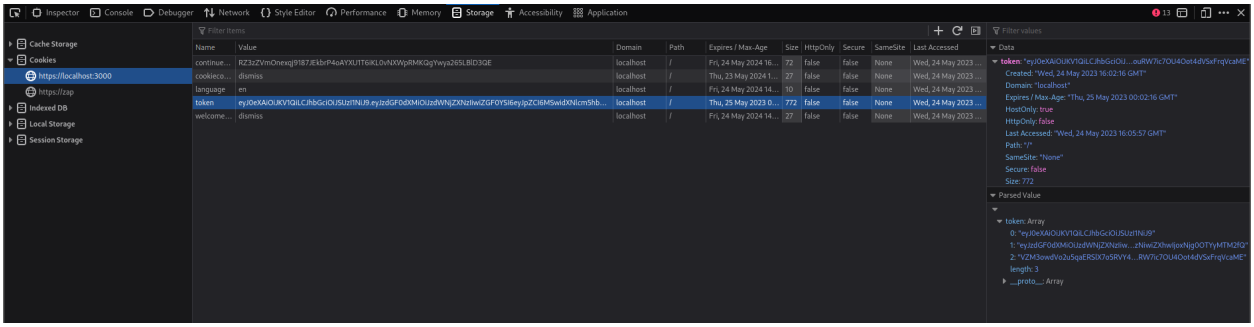

Figure 13: Fuzzer looking for default credentials

3.4.10 WSTG_SESS_04 - Testing for Exposed Session Variables

The goal of this test is to ensure that proper encryption is implemented, to review the caching configuration and to assess the channel and methods' security. We want to verify if the session tokens are visible when making requests and that can be done using ZAP. In the figure below, we can observe that the *sid* is visible in the URL.

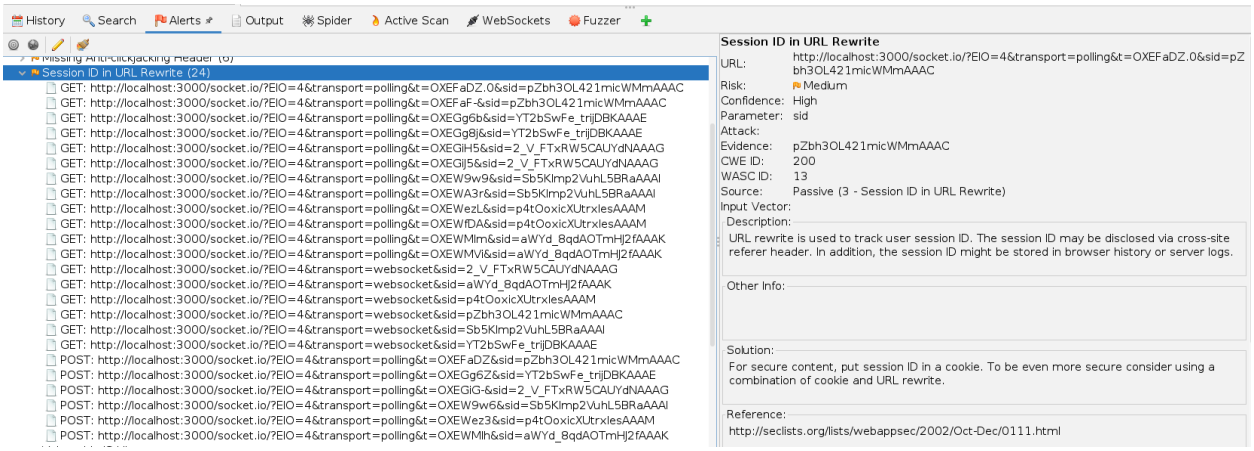

Figure 14: Session token exposed in URL

3.4.11 WSTG_CLNT_03 - Testing for HTML Injection

The goal of this test is to identify possible HTML injection points on the **Juice Shop**. One of the points we discovered where it's possible to execute HTML code is in the search functionality, as demonstrated below. The code used was:

```
<h1>HTML Injection</h1>
<img src="https://www.thedesignwork.com/wp-content/uploads/2011/10/Random-Pictures-of-Conceptual-and-
    Creative-Ideas-02.jpg" alt="Conceptual and Creative Ideas">
<br>
<a href="https://www.google.com">Visit Google</a>
```

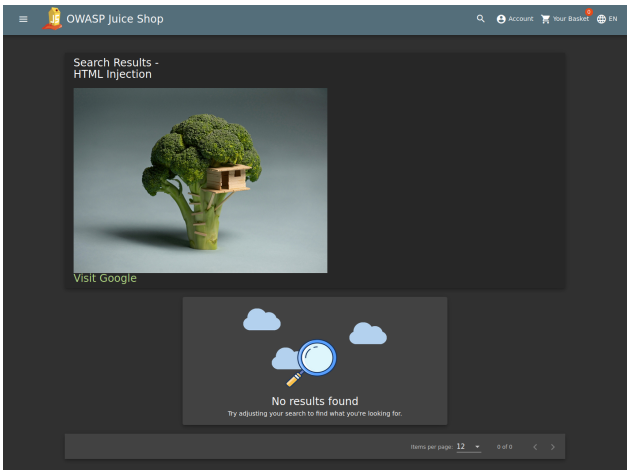

Figure 15: Image and link added using HTML injection

4 Second Phase - Web Application Security Firewall

To run WAF on the same VM we're running the web application, we can run the following command:

```
docker run -d -p 3001:80 -e PARANOIA=4 -e BACKEND=http://machineip:3000/ owasp/modsecurity-crs:apache
```

In this case, "machineip" would be the NAT IP of the virtual machine (192.168.153.140).

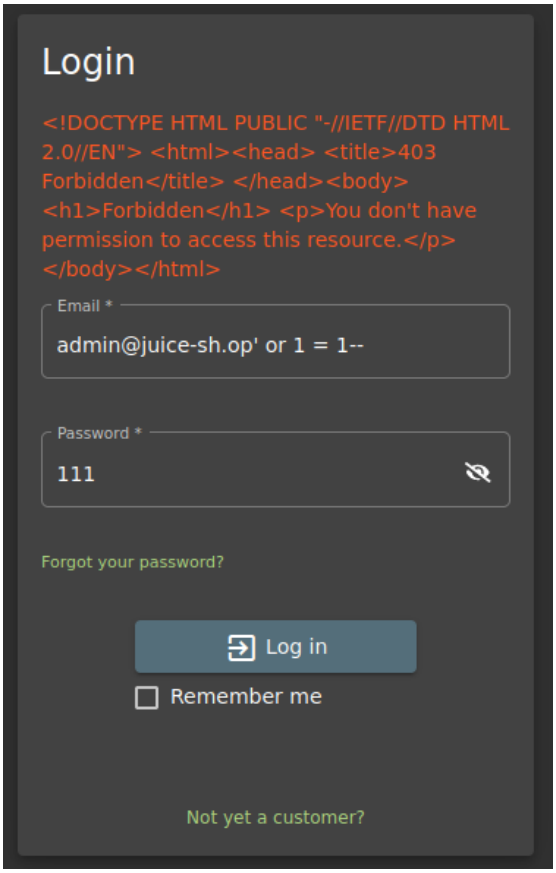

Figure 16: SQL injection failed after starting WAF

5 Conclusion

For this project, there were several goals that were not accomplished, the most notable one being the absence of the second phase, which corresponds to use the firewall to monitor, filter and block HTTP traffic to the web application. Besides that, we just tested some of the subsections of the guidelines, even though most of them could not be applied to this project in concrete.

One thing that we accomplished was using the OWASP ZAP software to detect for vulnerabilities inside a web application and how to associate the vulnerabilities found to the WSTG guidelines, even though some of them aren't as clear as we hoped them to be. One of the main reasons for the few testing is the lack of the guidelines clarity, the huge variety of tests to be executed and the understanding of the project itself. We can confidently say that our priority in future work would be to make further testing and to create the web application firewall.