1 2 9 0

UNIVERSIDADE D
COIMBRA

Information Technology Security

Web Application Security

Francisco Catarino Mendes - 2019222823
Leonardo Oliveira Pereira - 2020239125
Department of Informatics Engineering
University of Coimbra

Introduction

This assignment will be focused on exploring the WSTG (Web Security Testing Guide) web security testing guidelines, configuring and exploring the usage of ModSecurity reverse proxy as a WAF (Web Application Firewall), and configuring Keycloak as an identity provider.

After the presentation of its structure, this assignment is split into three phases: the first phase is dedicated to exploring the JuiceShop's security, and the second phase aims to monitor, filter, and block HTTP traffic to the JuiceShop through the implementation of a ModSecurity WAF, with the goal of addressing the security issues identified in the first phase. The third phase involves Keycloak as an identity provider. Keycloak is going to be set up to enable SSO on an application, and then it is going to be tested and improved security wise. The scenario of the first two phases is displayed in Figure 1:

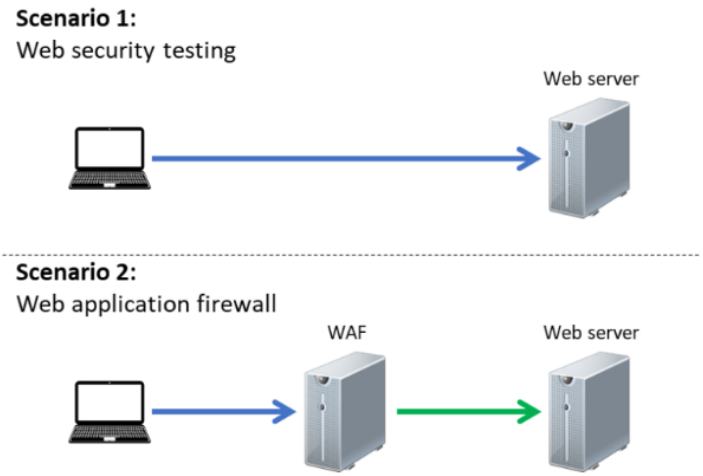

Figure 1: Security testing and WAF phases of the Assignment

The main goals set to be achieved are to explore web application security and to implement a web application firewall to successfully secure a web application against application-layer attacks, as well as to understand how to harden the security of Keycloak. The web application going to be used in this assignment is the OWASP JuiceShop.

1 - Structure of the Practical Assignment

In this section, the structure of the project is defined, going through the network, server, and services components.

1.1 - Network

All of the elements of the assignment, such as the Juice Shop web server, the ModSecurity WAF, and Keycloak are going to be used in the same Virtual Machine, therefore using the same network. The different states of the Juice Shop are going to be differentiated by the ports. The vulnerable version will be on port 3000 like the manufacturers instructions have, and the version protected with the WAF will be on apache2's port, 80.

Furthermore, for this to work the proxy modules need to be enabled and activated, like Figure 2 has, so that apache2 knows which webpage it is targeting for the use of the WAF.

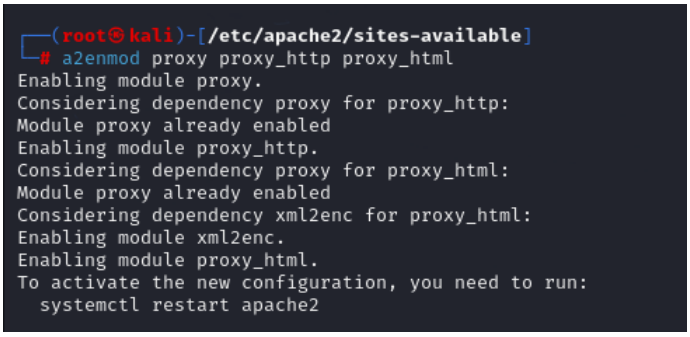

Figure 2: Enabling the proxy modules

1.2 - Server

Once again, the OWASP Juice Shop web server is divided in two servers, in two different ports, to simulate the vulnerable situation and the WAF protected situation. As for the WAF, it runs on the apache2 server.

Figure 3 shows the configurations of the file that apache2 uses to coordinate itself, with the proxies redirecting the server into the Juice Shop IP address. Also, the SecRuleEngine variable has to be added and set to on so that ModSecurity works:

Figure 3: 000-default.conf configurations

Lastly, ModSecurity's rules have to be enabled in the apache2 configuration file named **security2.conf**. This procedure is illustrated in Figure 4:

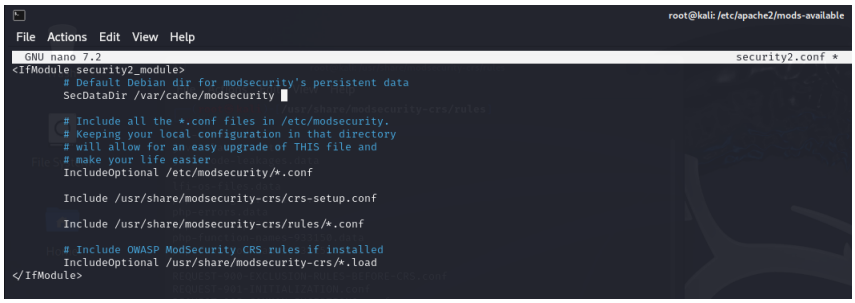

Figure 4: Enabling the core rule set of ModSecurity in apache2

1.3 - Services

Here, the details of how ModSecurity, the OWASP Juice Shop, and Keycloak were configured are provided. Starting with the OWASP Juice Shop, version 16.0.0 was downloaded and set up via Docker, according to the instructions present in the developers' GitHub, as exhibited in Figure 5:

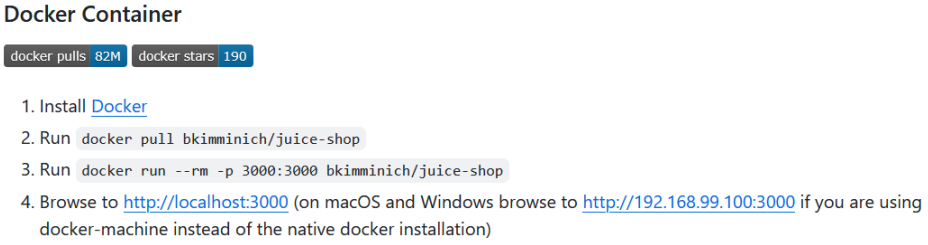

Figure 5: Docker installation of OWASP's Juice Shop

Next, regarding ModSecurity, first off, the value for SecRuleEngine has to be changed from DetectionOnly to On, inside the **modsecurity.conf** file, like Figure 6 has:

Figure 6: Changing the value of SecRuleEngine to On

After that, the OWASP ModSecurity Core Rule Set (CRS) was downloaded. It is a set of generic attack detection rules for use with ModSecurity or compatible web application firewalls. The CRS aims to protect web applications from a wide range of attacks, including the OWASP Top Ten, with a minimum of false alerts. The CRS provides protection against many common attack categories, including SQL Injection and cross-site scripting. Figure 7 shows how the current rule set that comes prepackaged with ModSecurity was deleted, and the OWASP CRS was downloaded:

Figure 7: Deleting the prepackaged ruleset and downloading the OWASp CRS

As for the configuration of the OWASP CRS, it is part of the process shown in Figure 4, where ModSecurity's rules are enabled in apache2, Those rules already correspond to the OWASP CRS. To confirm that everything was configured successfully (concerning ModSecurity and apache2), a simple local file inclusion attack was run as a test. targeting the Juice Shop. Figure 8 displays the results, which validate that ModSecurity is working as intended:

Figure 8: Simple attack test to validate configurations

2 - Web Application Security Testing

Here lies the first proper phase of the assignment. The vulnerable version of the web application will be subject to a variety of tests, mainly from OWASP ZAP and the WSTG project.

2.1 - OWASP ZAP penetration tests

First off, a series of tests with the use of the OWASP ZAP tool will be performed on the web application, to collect as much information about vulnerabilities as possible, as well as to try and exploit some of those vulnerabilities.

2.1.1 - Automated Scan

Performing an automated scan is the simplest thing one can do with OWASP ZAP. It only requires the URL to attack and gives the option of using the traditional spider, the various forms of the AJAX spider, or both together. Then, the scan is performed. In this scenario, both of the spiders were used, and the results are in Figure 9:

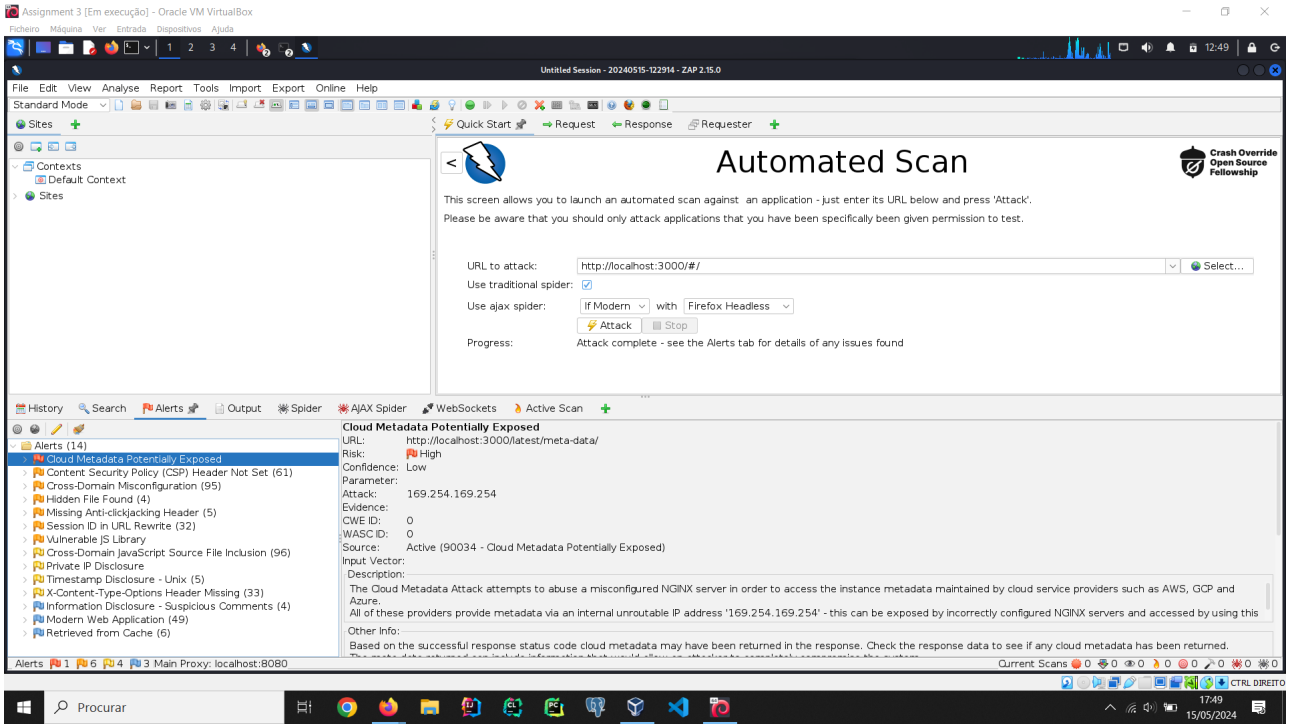

Figure 9: Automated Scan to localhost:3000

14 alerts ended up appearing, with only one of those being a high-risk alert, corresponding to cloud metadata potentially exposed.

2.1.2 - Active Scan exploring the most effective policies

The goal in this section is to explore some of the policies to determine if they are relevant when searching for more vulnerabilities. Two policies are going to be tested, one with **High** attack strength and **Default** threshold, to just up the strength of attacks a little bit, and one with **Insane** attack strength and **Low** threshold. This mode is supposedly the most powerful, as Insane is the highest attack mode and by applying a low threshold it means the tool will not filter as many alerts as normal. Figures 10 and 11 show the configuration of these policies:

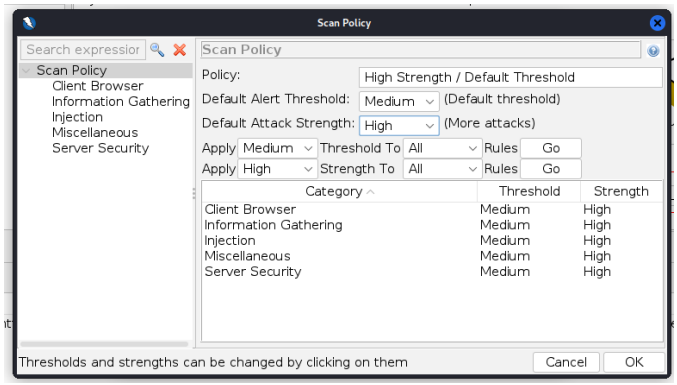

Figure 10: High Strength / Default Threshold policy

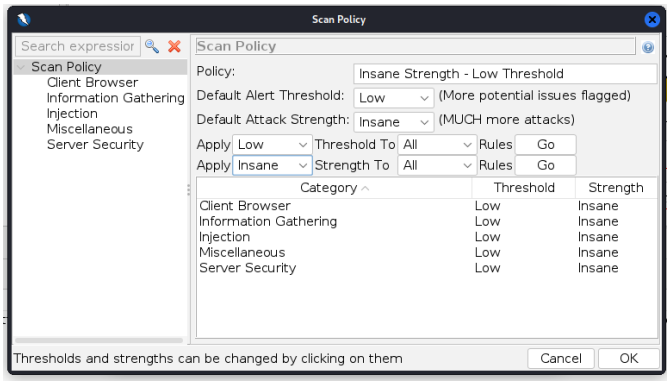

Figure 11: Insane Strength / Low Threshold policy

As for the active scans, the traditional spider was run first, and then the scans were processed. Figure 12 shows the results of the High Strength / Default Threshold policy, while Figure 13 shows the results of the Insane Strength / Low Threshold policy:

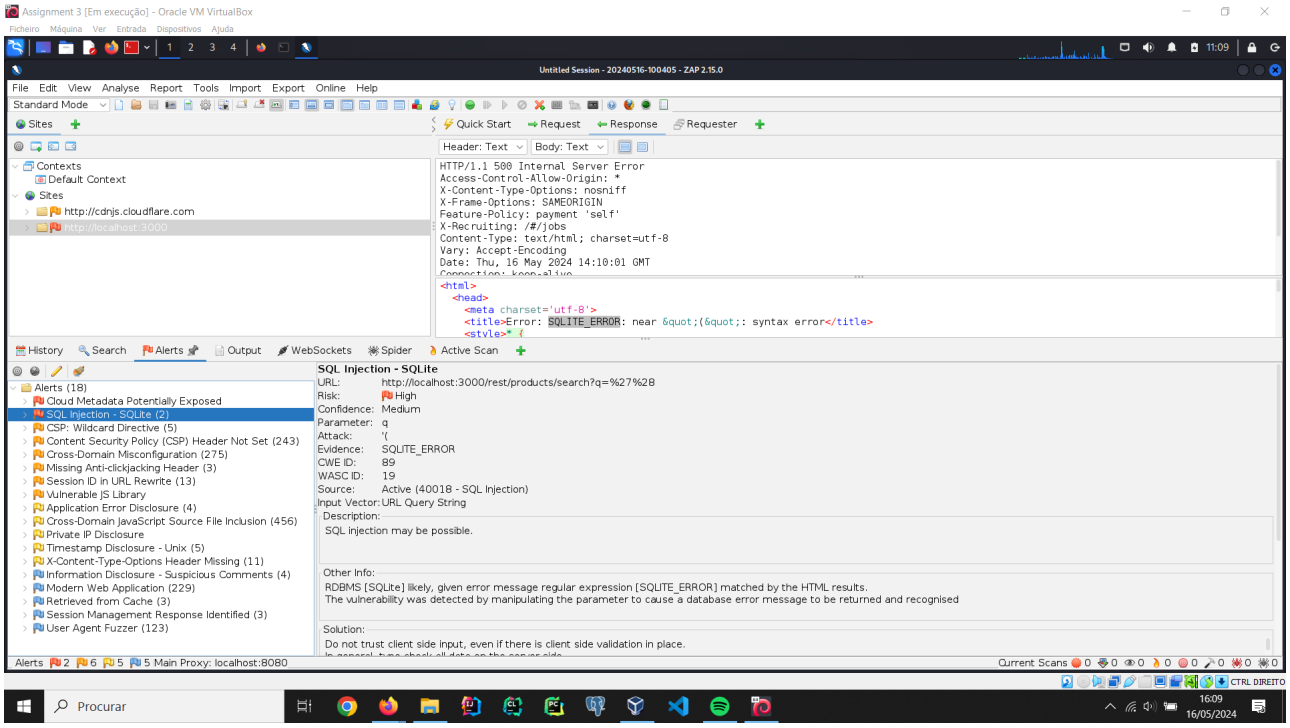

Figure 12: Scan with the High Strength / Default Threshold policy

The results are clear, as the average policy produced 18 alerts, and the highest policy produced 22 alerts. This confirms that the policies are indeed relevant in the search for alerts.

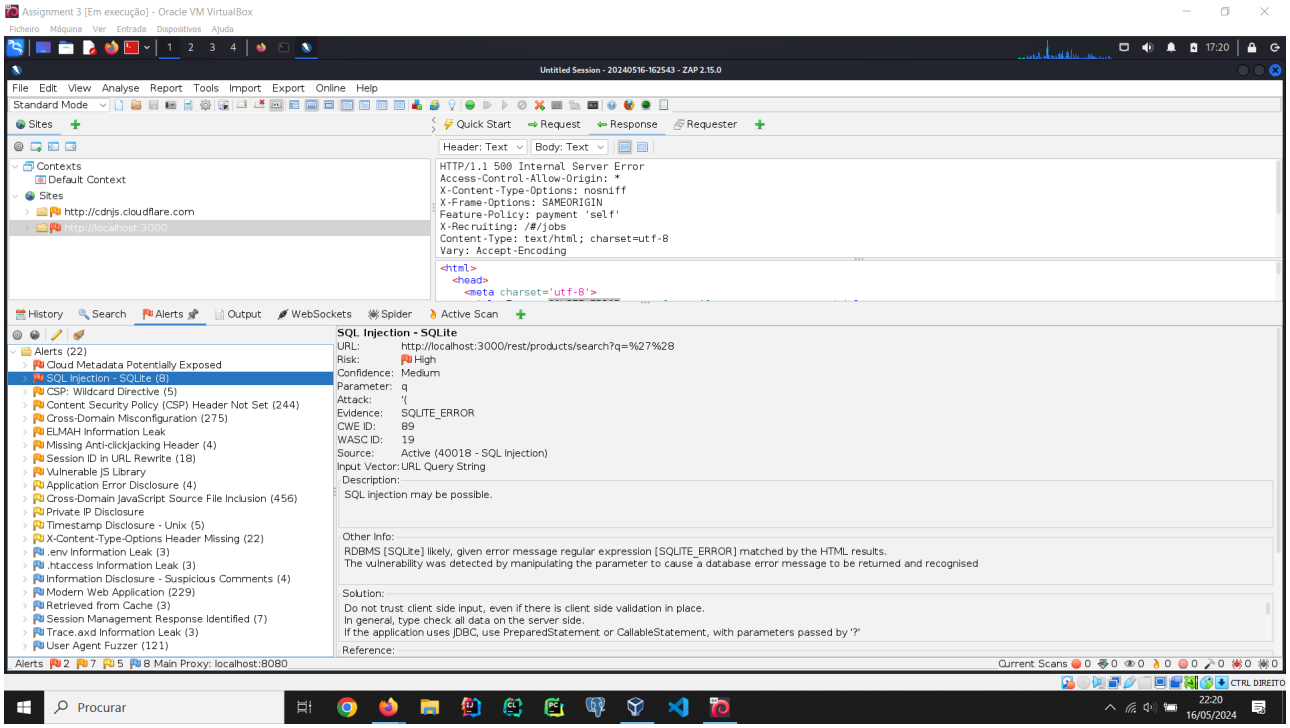

Figure 13: Scan with the Insane Strength / Low Threshold policy

2.1.3 - Automated Scan with new Add-Ons

Here, a few add-ons that could help detect new alerts were added, ranging from fuzzing to access control testing. Figures 14 and 15 show the add-ons included:

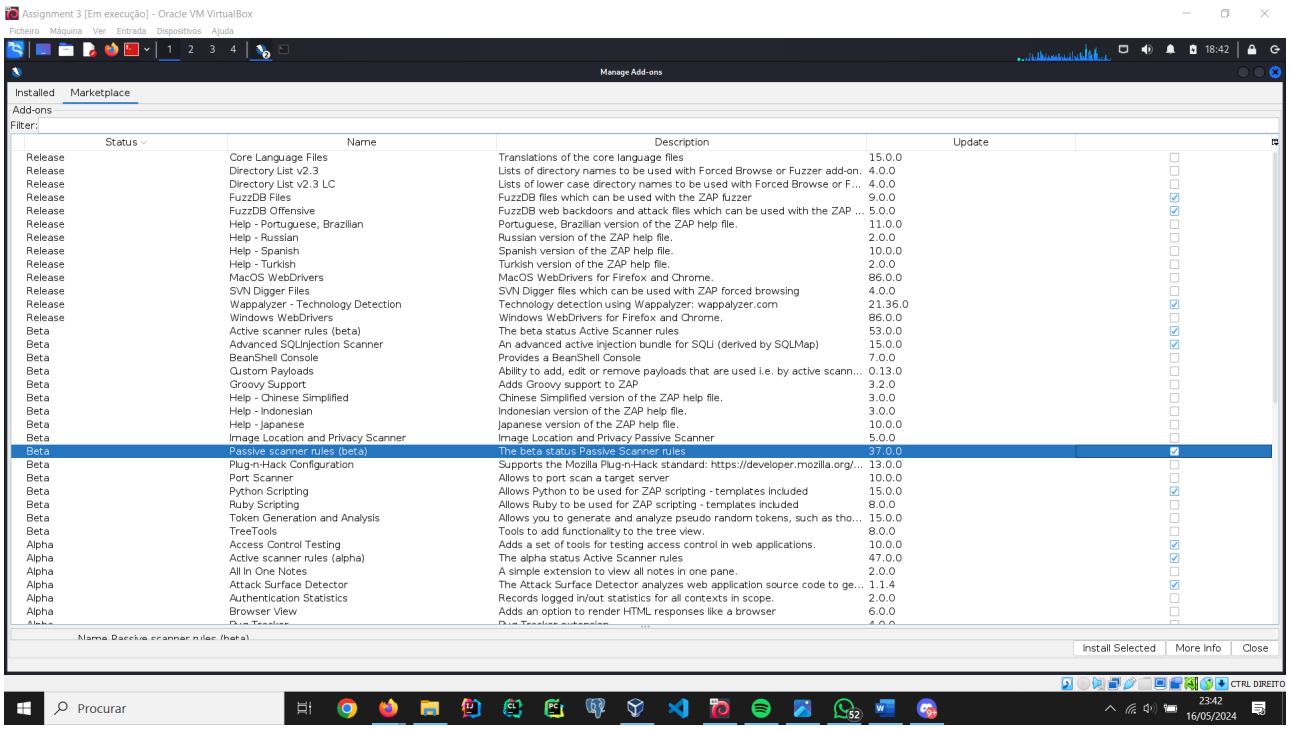

Figure 14: Add-ons included (1)

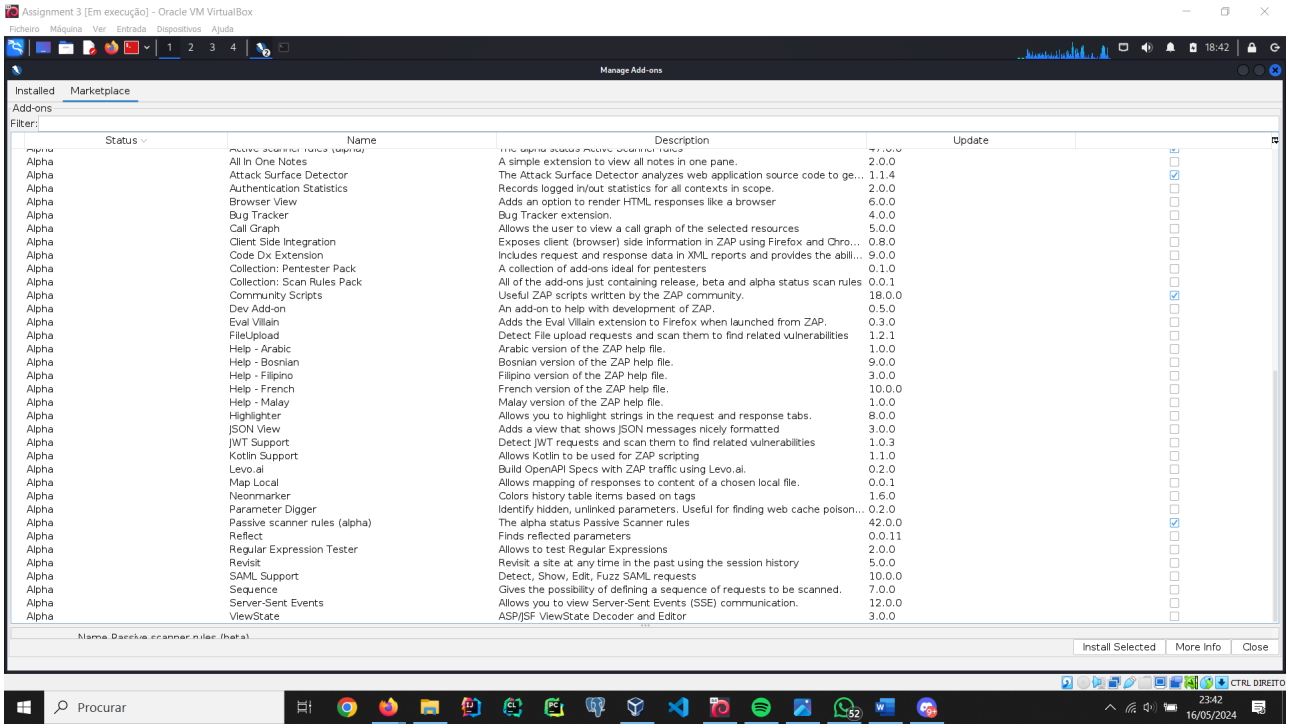

Figure 15: Add-ons included (2)

To test these, an automated scan was performed. Figure 16 displays the result:

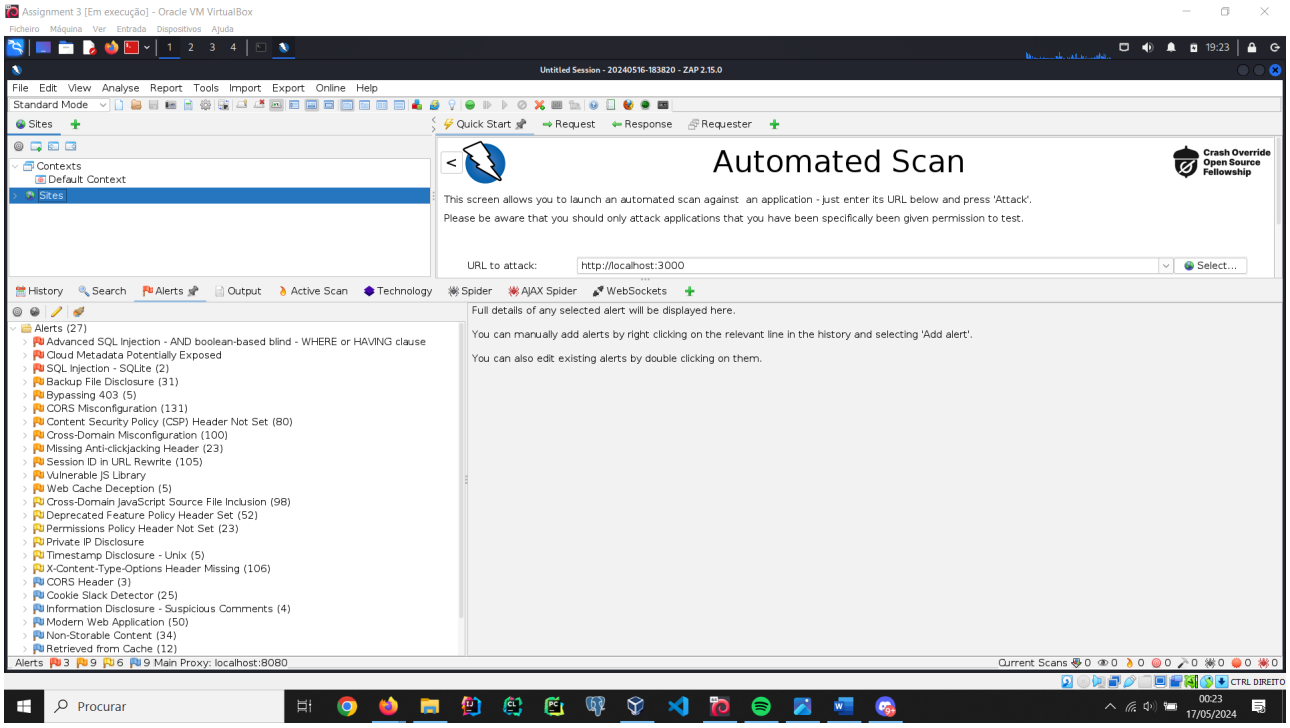

Figure 16: Automated Scan with add-ons

Compared with the default automated scan, this one is a lot better. 27 alerts were discovered, proving the efficiency of the new add-ons.

2.1.4 - Fuzz attacks to the login form

Now, two types of fuzz attacks are going to be performed on the login page, one SQL injection, and one brute force attack. For starters, one needs to identify the POST request of the login attempt, therefore, a purposefully incorrect login attempt must be made, as Figure 17 has:

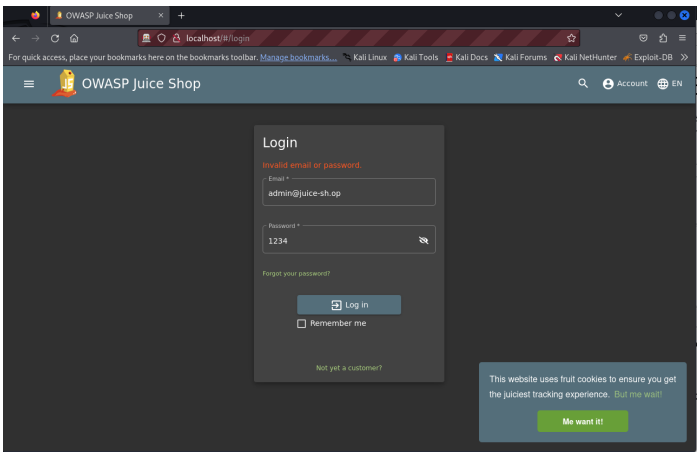

Figure 17: Incorrect login attempt

After identifying the request, the configurations of the fuzz can start. For the SQL injection, the username must be targeted, and the SQL injection payload must be selected in the file fuzzers section. Figures 18 and 19 have these configurations:

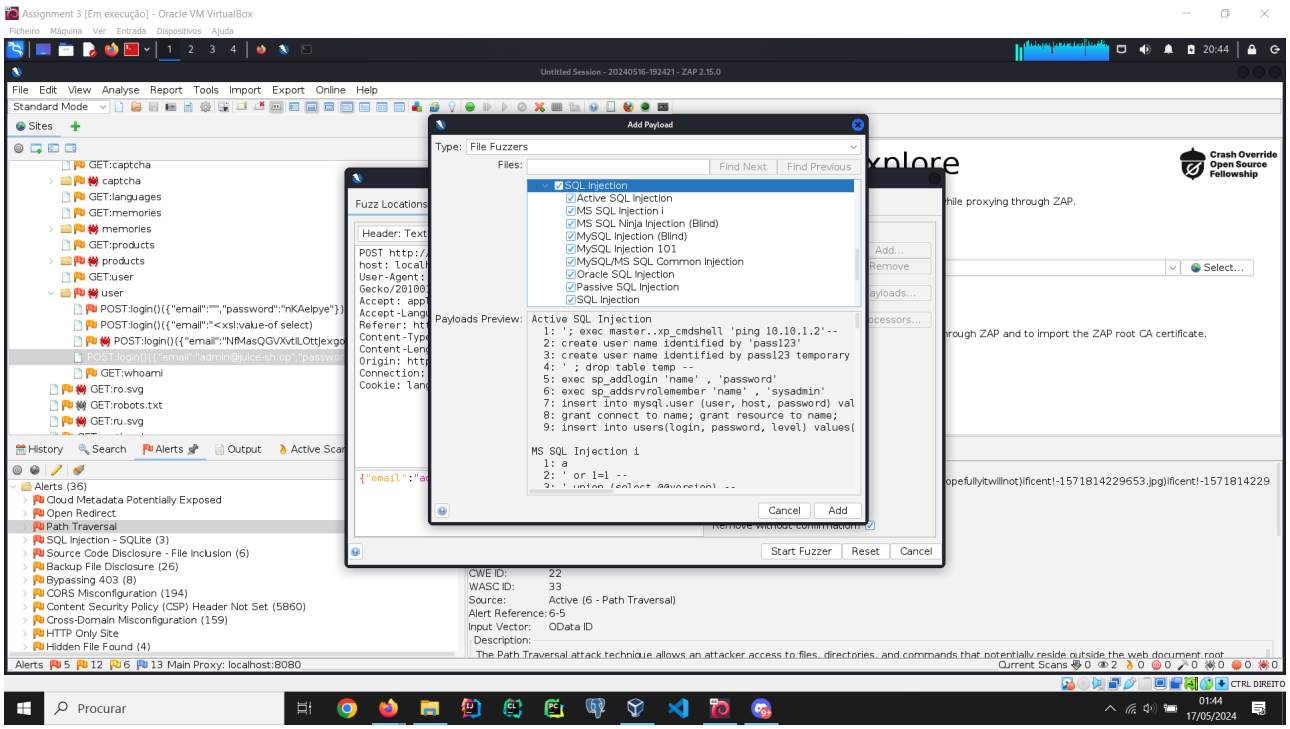

Figure 18: Selecting the SQL Injection payload

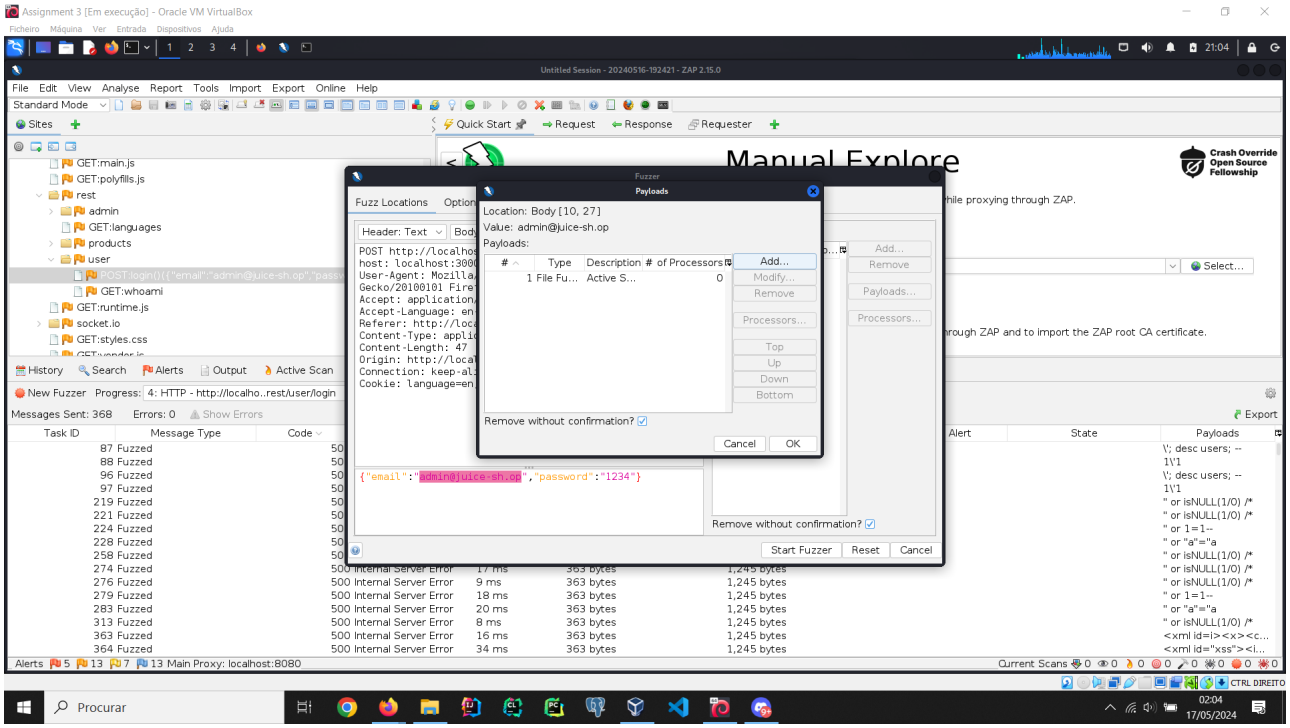

Figure 19: Targeting the username part

After that, the fuzz attack is performed. The results have different types of possible SQL Injections, and the one displayed in Figure 20 is going to be used in the next section as a way to manually exploit the web application:

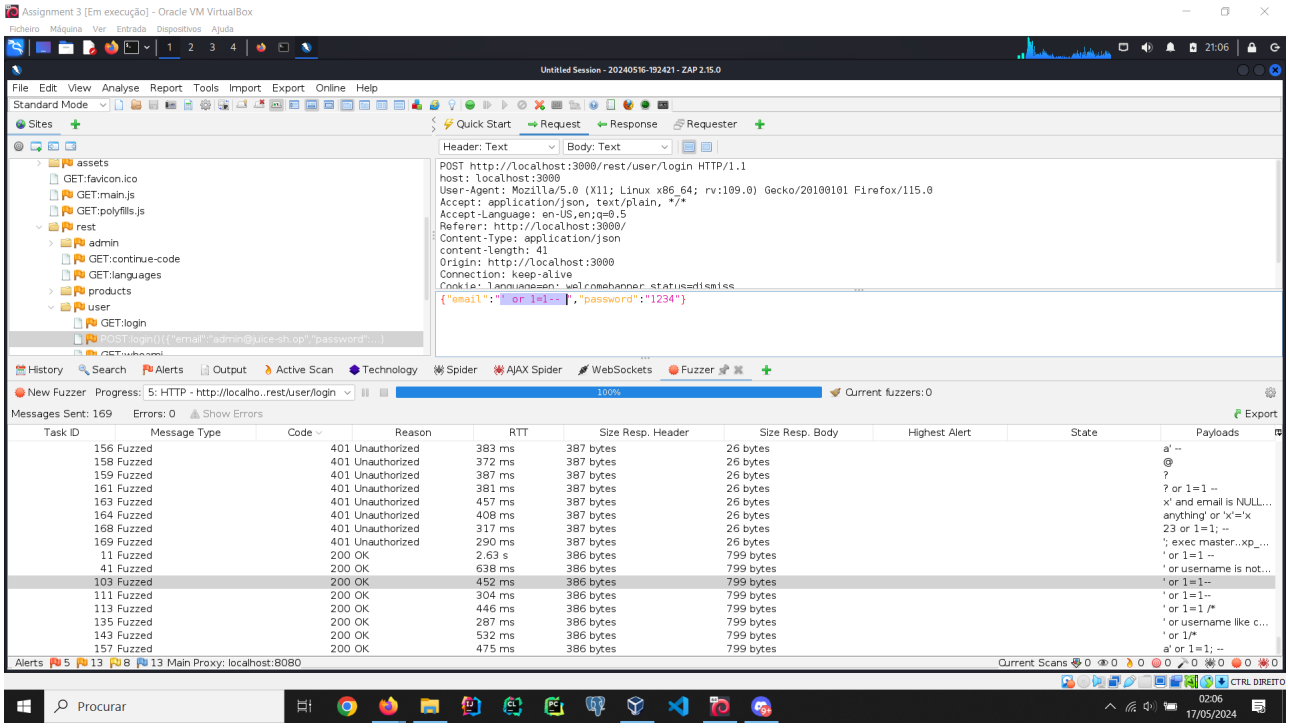

Figure 20: Results of the SQL Injection fuzzing scan

Moving on to the brute force attack, this time the password is targeted, because, of course, this process occurs by experimenting with as many passwords as possible in a short amount of time. Figure 21 has the wordlists payload being selected, while Figure 22 shows the password camp being targeted:

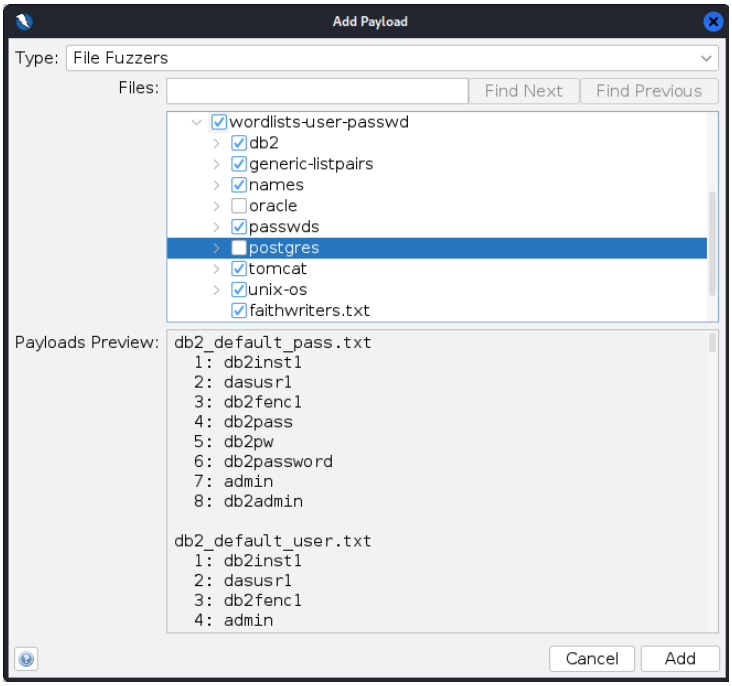

Figure 21: Selecting the wordlists payload

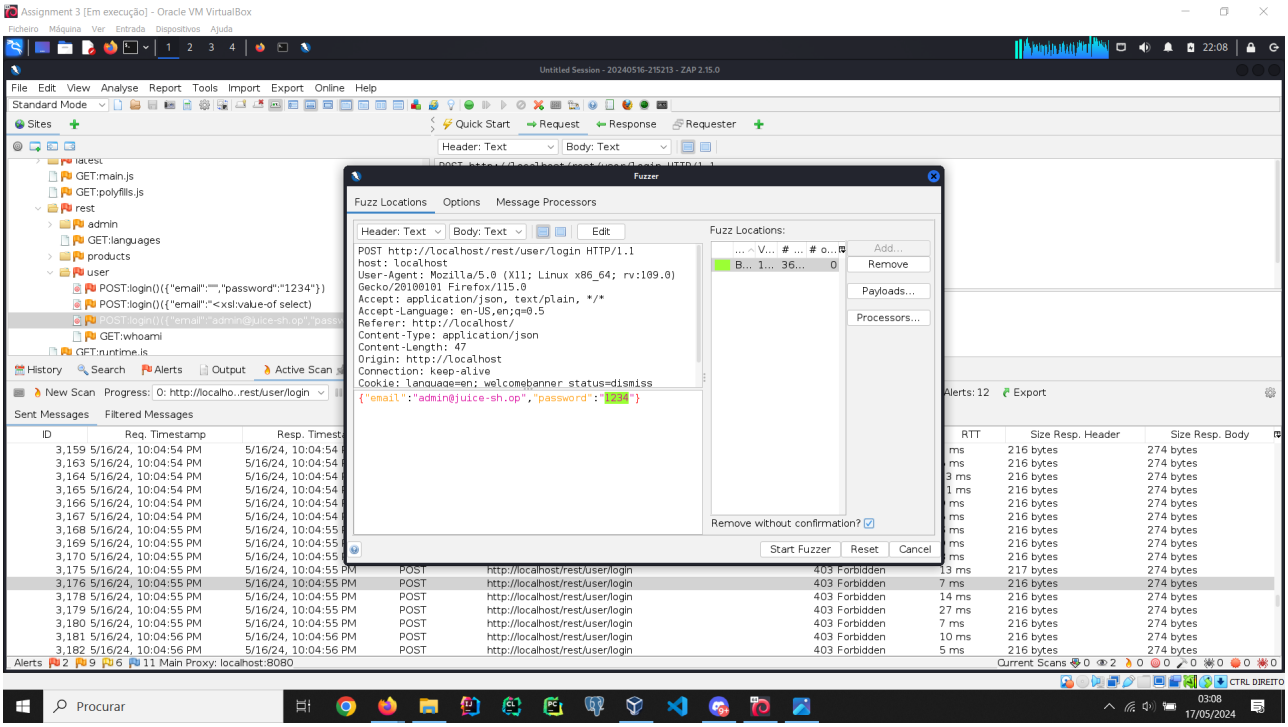

Figure 22: Targeting the password camp

The fuzz attack is now run. It ends up being successful halfway through, with the password **admin123** returning a 200 OK code. Figure 23 exhibits this result:

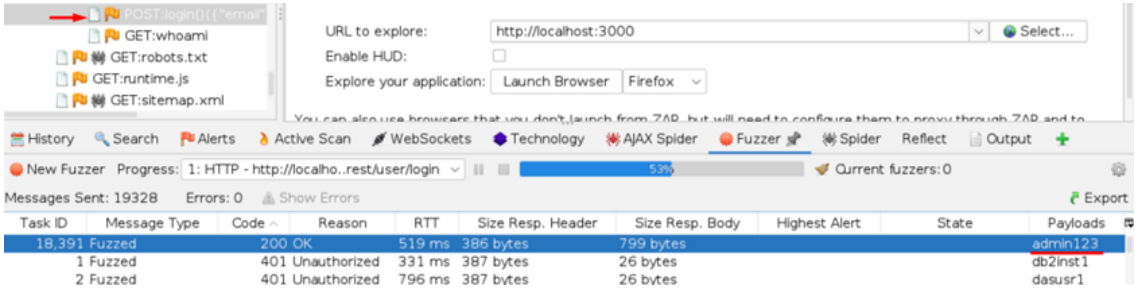

Figure 23: Results of the brute force fuzzing scan

2.1.5 - Manual tests to explore logged-in threats

From the results of Figures 20 and 23, two manual tests to explore the logged-in threats can be done. The first one is to use the SQL Injection to exploit the system and get inside, and this process is shown in Figures 24 and 25:

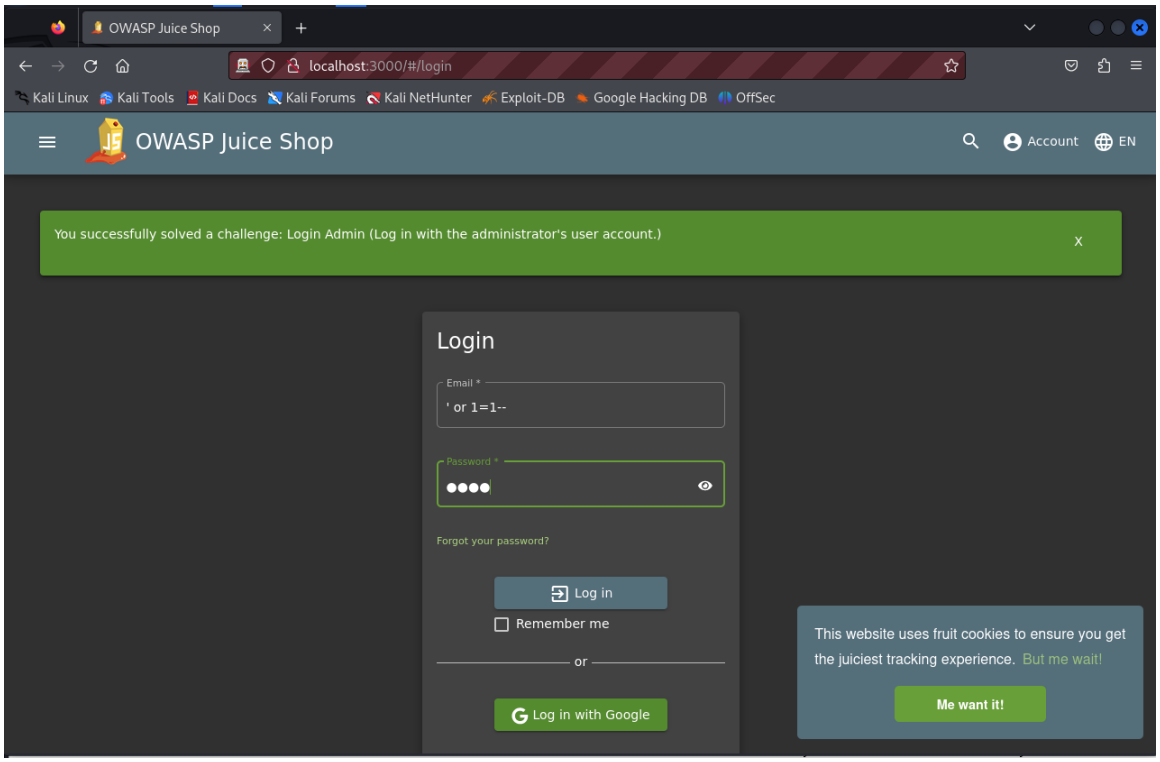

Figure 24: Exploiting the Login page with a SQL Injection (1)

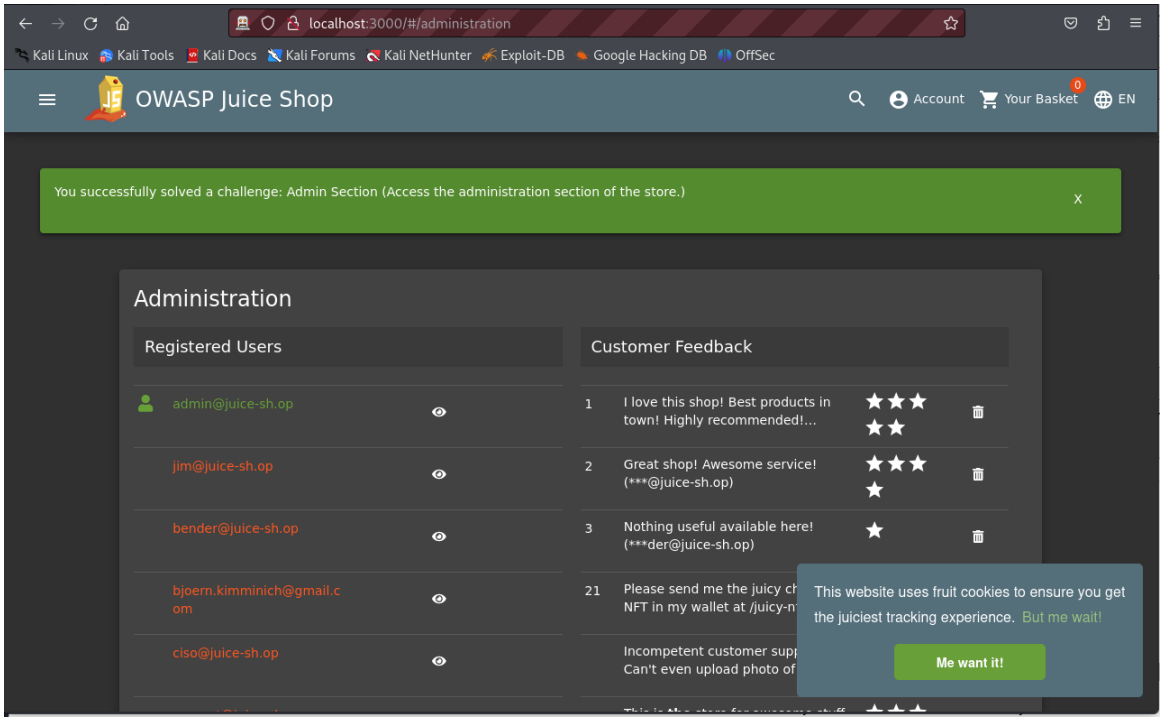

Figure 25: Exploiting the Login page with a SQL Injection (2)

Regarding the findings of the brute force attack, they can be used to just casually get inside the system, using the password discovered, like Figure 26 displays:

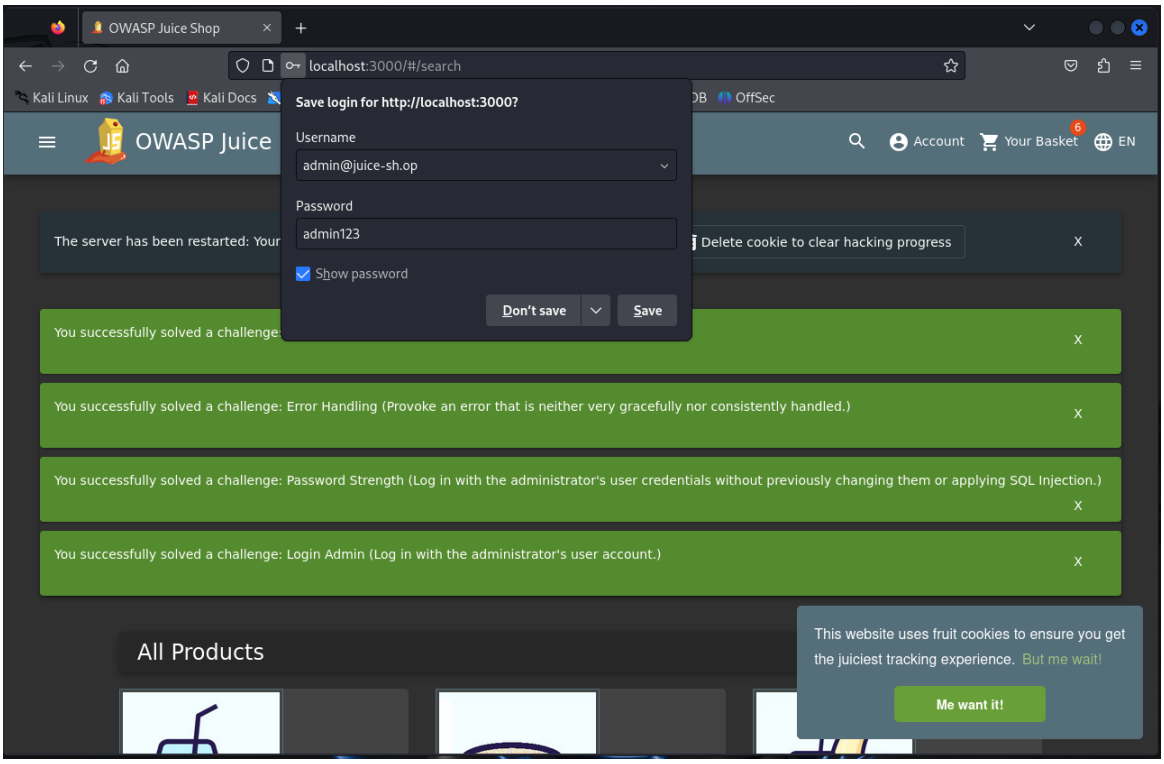

Figure 26: Using the password discovered in the brute force attack

2.1.6 - Active Scan to explore authenticated area

Finally, to end the OWASP ZAP tests and scans, an active scan is going to be performed to explore the authenticated area. Firstly, the Authentication Tester is going to be run as is shown in Figure 27, to help understand which parts of the authentication process ZAP can automatically handle:

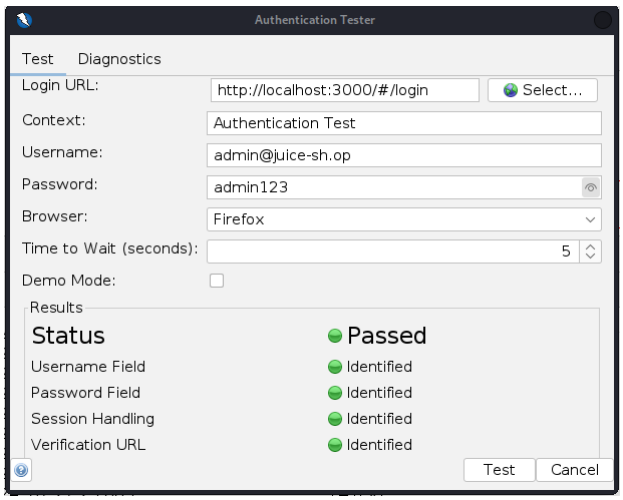

Figure 27: Running the Authentication Tester

As everything passed, ZAP identified all of the information required. Now, the authentication verification and session handling parts of the context need to be configured, to be running in the mode Auto-Detect. Figures 28 and 29 show the configurations applied:

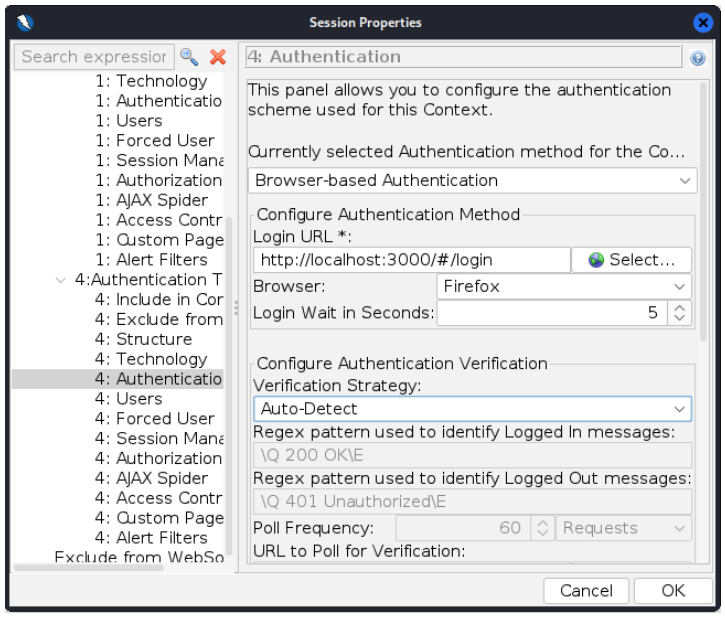

Figure 28: Configuring authentication verification

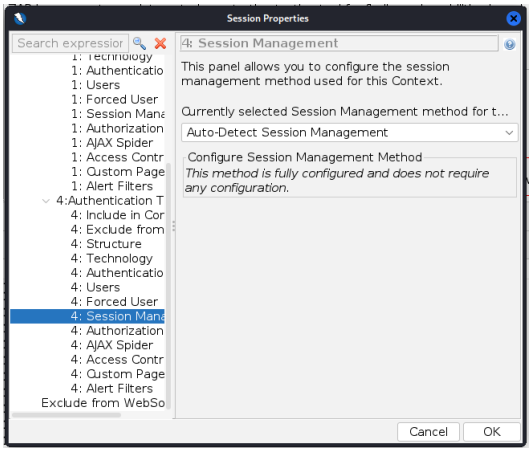

Figure 29: Configuring session handling

With that in place, the new context needs to be tested. An automation framework plan has to be created, configured, and run. The requester job is going to be selected. Then, the authenticated user is selected, and a suitable URL is added to the requests tab, on the requesters' job. Finally, the plan is executed, resulting in Figures 30 and 31:

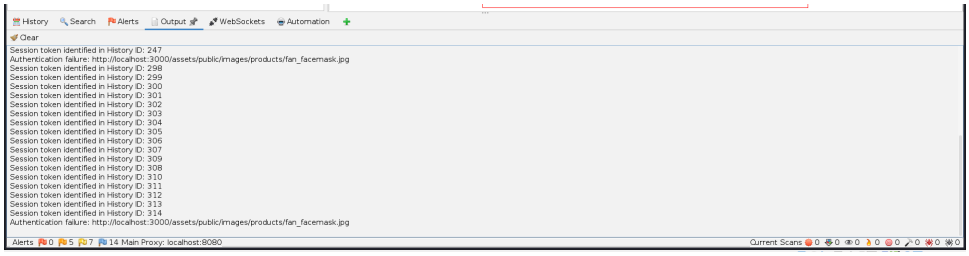

Figure 30: Output Tab

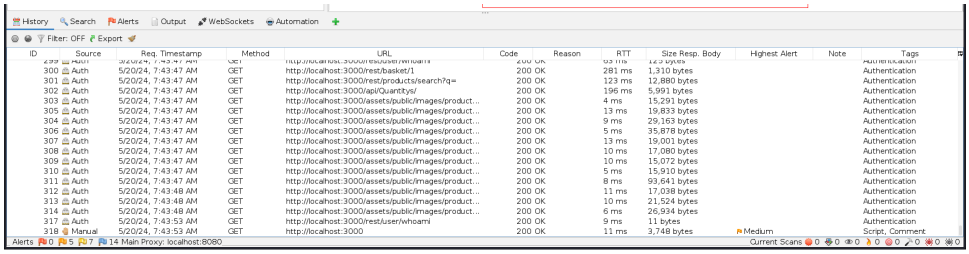

Figure 31: History Tab

While the History Tab shows the intended, which is the final message having a Source of 'Manual' and the last 'Auth' message being to the 'verification URL' and succeeding, the Output Tab on the other hand shows authentication errors, which is strange given the fact that when the plan is running, ZAP automatically tries to login and is successful.

But all in all, now the application has a context that handles authentication, like it was required.

2.2 - WSTG - Configuration and Deployment Management Testing

2.2.1 - Enumerate Infrastructure and Application Admin Interfaces

Administrator interfaces may be present in the application or on the application server to allow certain users to undertake privileged activities on the site. Tests should be undertaken to reveal if and how this privileged functionality can be accessed by an unauthorized or standard user. The test objective is to identify hidden administrator interfaces and functionality.

To begin with, admin interfaces are usually located at predictable URLs. Common paths are going to be used to try and discover any hidden or default admin interfaces. URLs ending with **/admin**, **/superuser**, **/manager**, for example, were tested. One gave away a hidden admin interface, that being the path **/administration**.

Next, the tool **dirb** will be used to try and enumerate directories and files. This tool can help identify hidden admin interfaces by brute-forcing common paths. Figure 32 shows the command run:

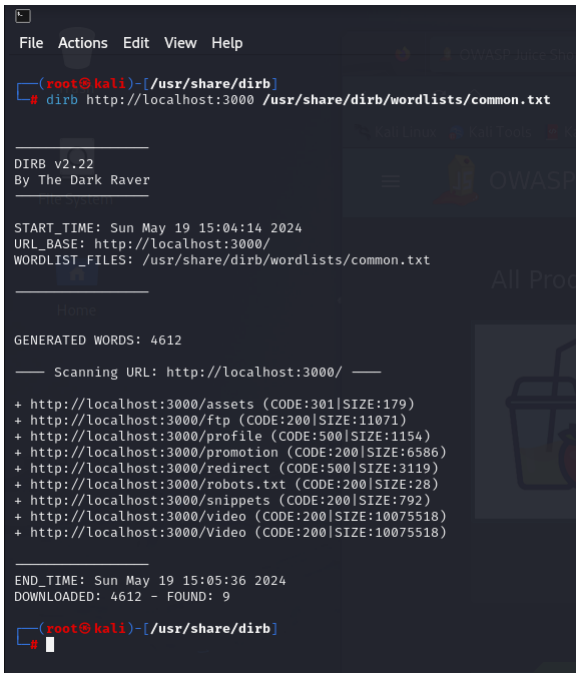

Figure 32: dirb command

None of these URLs usually indicate admin interfaces. But it might be useful to analyze those with the code 200 OK. For example, the robots.txt file often contains disallowed paths that might hint at hidden areas. After accessing it, it reveals that the **/ftp** path is supposed to be disallowed. And when visiting that path, what is found is displayed in Figure 33:

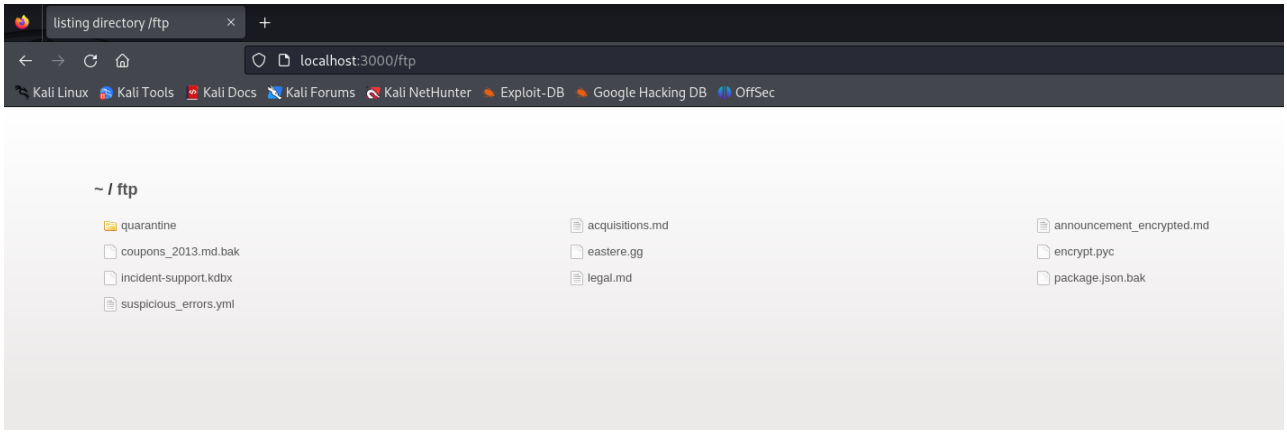

Figure 33: localhost:3000/ftp results

The directory listing at http://localhost:3000/ftp does not immediately appear to be a standard admin interface. However, it does provide access to potentially sensitive files, which could be useful for further exploration and exploitation.

2.2.2 - Test HTTP Methods

HTTP offers a number of methods that can be used to perform actions on the web server. While GET and POST are by far the most common methods that are used to access information provided by a web server, HTTP allows several other (and somewhat less known) methods. Some of these can be used for nefarious purposes if the web server is misconfigured. The objectives of this test are to enumerate supported HTTP methods, test for access control bypass, test XST vulnerabilities, and test HTTP method overriding techniques.

To enumerate supported HTTP methods, the http-methods Nmap script and the tool **curl** are going to be used. Figure 34 represents the commands run:

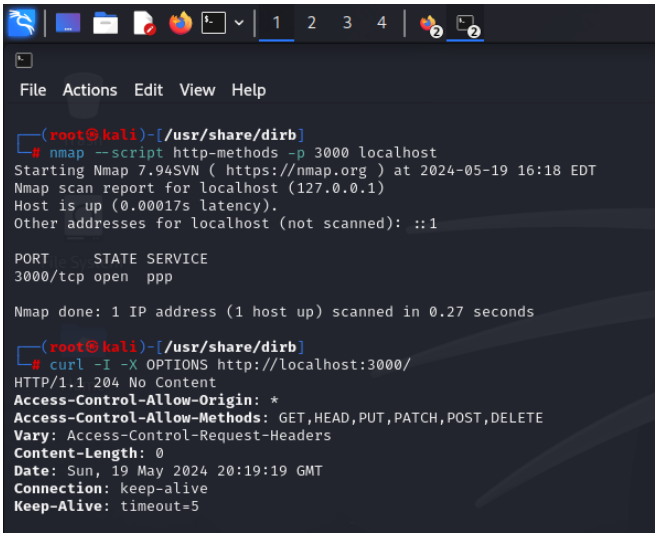

Figure 34: Nmap script and curl OPTIONS command

This output indicates the HTTP methods supported by the server. According to the Access-Control-Allow-Methods header, the following methods are supported: GET, HEAD, PUT, PATCH, POST, DELETE.

Next, to test for access control bypass, 5 curl commands will be formed, to try accessing a protected user authentication details endpoint, try posting feedback on the Juice Shop without proper authentication, try updating a resource, such as user details, try partially updating a resource, and try deleting a resource, such as a feedback entry. In the end, all of these returned information about the HTML page targeted, corresponding to 200 OK responses, as Figure 35 has, which indicates that access control mechanisms are not functioning correctly.

Figure 35: Trying to access a protected user authentication details endpoint

As for XST vulnerabilities, the desired is to check if the TRACE method is enabled and can be exploited for XST attacks. The check is shown in Figure 36:

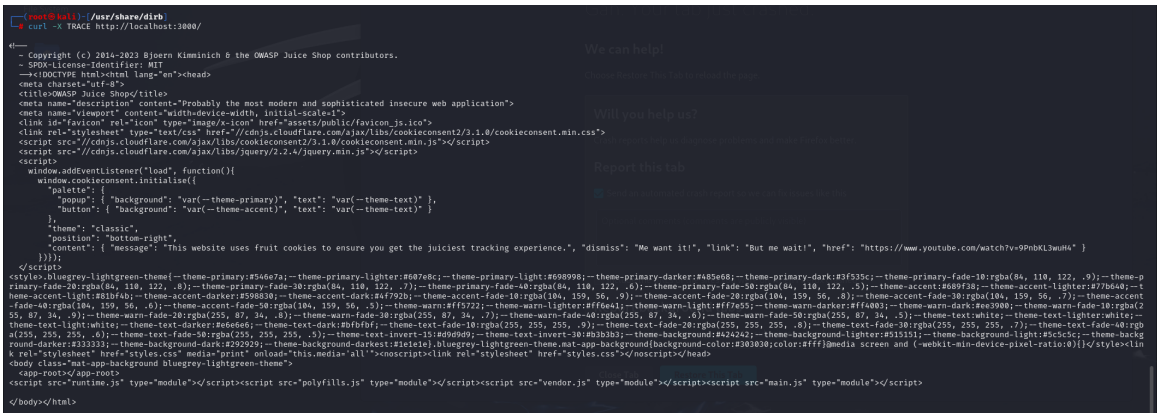

Figure 36: Checking if the TRACE method is enabled

The response includes a full HTML page that is reflective of the request. This indicates that the TRACE method is enabled on the server. The presence of the TRACE method can be a security concern because it may allow Cross-Site Tracing (XST) attacks, where malicious scripts can use the TRACE method to bypass same-origin policies and capture sensitive information such as cookies or authentication tokens.

Finally, to test HTTP method overriding techniques, headers like X-HTTP-Method-Override will be used to attempt to override the HTTP method. The test was made in the Feedback functionality, where a POST response corresponds to creating feedback, and the PUT response corresponds to updating feedback. Well, when the attempt to override is run, the response is similar to that of a PUT response, therefore meaning that the server accepts and correctly processes the overridden method.

2.2.3 - Test HTTP Strict Transport Security

The HTTP Strict Transport Security (HSTS) feature lets a web application inform the browser through the use of a special response header that it should never establish a connection to the specified domain servers using unencrypted HTTP. Instead, it should automatically establish all connection requests to access the site through HTTPS. It also prevents users from overriding certificate errors. The objective of this test is to review the HSTS header and its validity.

The presence of the HSTS header can be confirmed by examining the server's response through an intercepting proxy or by using curl as follows in Figure 37:

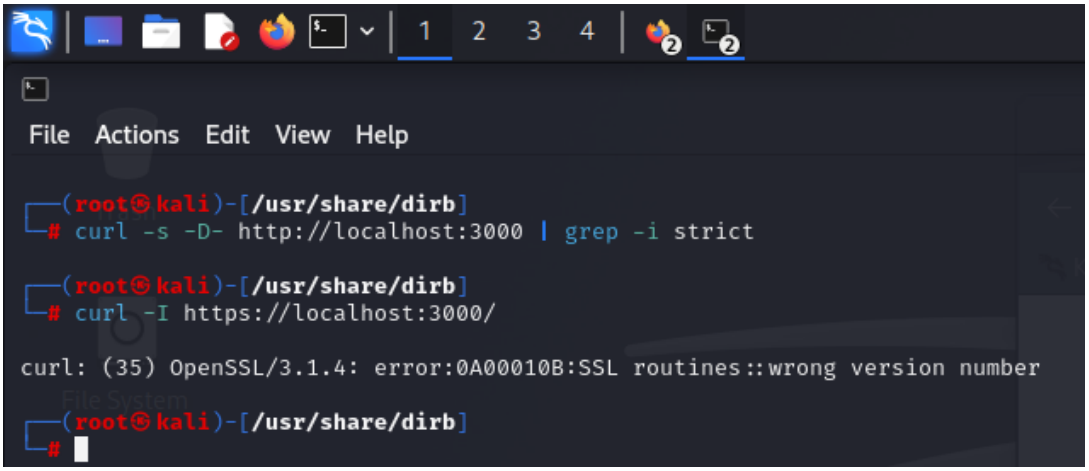

Figure 37: Confirming the presence of the HSTS header

The lack of output suggests that there is no Strict-Transport-Security header in the HTTP response. Since the application is not running on HTTPS, there will be no automatic redirection from HTTP to HTTPS, and thus no HSTS enforcement.

2.3 - WSTG - Identity Management Testing

2.3.1 - Test Role Definitions

Concerning user roles, some information is already possessed due to the ZAP analysis, which is the credentials of an admin account. But other roles might exist, so the first thing needed is to discover them. For that, **sqlmap** is going to be used to access the database of the web application to try and acquire some valuable information. The easiest way to access the databases is to form some kind of request to the database, like searching for a product. Therefore, the URL corresponding to that same search is the one used to access the database. Before trying to perform the sqlmap command, there is just one thing needed, which is to increase the **–level** and **–risk** options to allow sqlmap to perform a more thorough analysis, and discover that the parameter in the URL is injectable. In Figure 38, the database can already be seen:

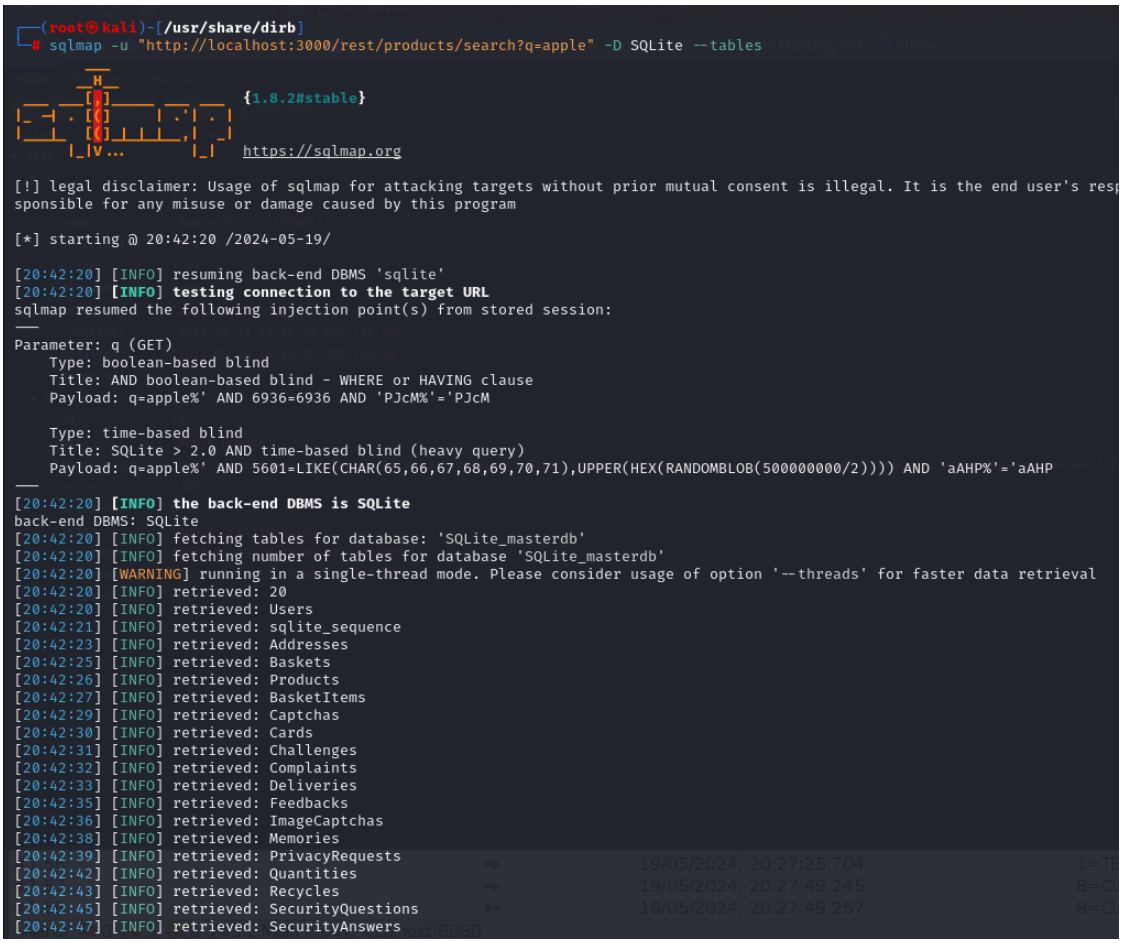

Figure 38: Discovering the database

One of the tables is named ”Users”. It must correspond to information about users. When accessing it, the following is discovered, shown in Figure 39:

```
Database: <current>
Table: Users
[15 entries]
+----+------------+-----------------------------+----------+----------------------------------+-------------+
| id | role       | email                       | isActive | password                         | lastLoginIp |
|    | deluxeToken                                                                              |             |
+----+------------+-----------------------------+----------+----------------------------------+-------------+
| 9  | admin      | J12934@juice-sh.op          | 1        | 0192023a7bbd73250516f069df1      |             |
|    | <blank>                                                                                  | undefined   |
| 15 | customer   | accountant@juice-sh.op      | 1        | e541ca7ecf72b8d1286474fc613      |             |
|    | <blank>                                                                                  | <blank>     |
| 1  | customer   | admin@juice-sh.op           | 1        | 0c36e517e3fa95aabf1bbffc674      |             |
|    | <blank>                                                                                  | <blank>     |
| 11 | admin      | amy@juice-sh.op             | 1        | 6edd9d726cbdc873c539e41ae87      |             |
|    | <blank>                                                                                  | <blank>     |
| 3  | deluxe     | bender@juice-sh.op          | 1        | 861917d5fa5f1172f931dc700d8      |             |
|    | d715c2c75d4a42d3825a050e0a0163c1959b51165373f17bd8eed7b1e05bf20d                         | <blank>     |
| 4  | admin      | bjoern.kimminich@gmail.com  | 1        | 3869433d74e3d0c86fd25562f83      |             |
|    | <blank>                                                                                  | <blank>     |
| 12 | customer   | bjoern@juice-sh.op          | 1        | f2f933d0bb0ba057bc8e33b8ebc      |             |
|    | <blank>                                                                                  | <blank>     |
| 13 | customer   | bjoern@owasp.org            | 1        | b03f4b0ba8b458fa0acdc02cdb9      |             |
|    | <blank>                                                                                  | <blank>     |
| 14 | admin      | chris.pike@juice-sh.op      | 1        | 3c2abc04e4a6ea8f1327d0aae37      |             |
|    | <blank>                                                                                  | <blank>     |
| 5  | admin      | ciso@juice-sh.op            | 1        | 9ad5b0492bbe528583e128d2a89      |             |
3KH | <blank>                                                                                  | <blank>     |
| 17 | customer   | demo                        | 1        | 030f05e45e30710c3ad3c32f00d      |             |
|    | <blank>                                                                                  | <blank>     |
| 19 | admin      | emma@juice-sh.op            | 1        | 7f311911af16fa8f418dd1a3051      |             |
|    | <blank>                                                                                  | <blank>     |
| 21 | deluxe     | ethereum@juice-sh.op        | 1        | 9283f1b2e9669749081963be046      |             |
|    | efe2f1599e2d93440d5243a1ffaf5a413b70cf3ac97156bd6fab9b5ddfcbe0e4                         | <blank>     |
| 2  | customer   | jim@juice-sh.op             | 1        | 10a783b9ed19ea1c67c3a27699f      |             |
|    | <blank>                                                                                  | <blank>     |
| 18 | accounting | john@juice-sh.op            | 1        | 963e10f92a70b4b463220cb4c5c      |             |
|    | <blank>                                                                                  | 123.456.789 |
```

Figure 39: Some of the ”Users” Table

There are 4 existing roles. One for admins, one for customers, one for deluxe members, and one for accountants. Now all that is left is to enter inside one account of each role to test if roles have access to the endpoints of the other roles. In order to do that, SQL Injections in OWASP ZAP were performed, the same way they were in section 2.1.4.

In the end, every role can only access their respective pages. If an admin tries to access the accounting page, the warning displayed in Figure 40 appears. The same goes for the other roles trying to access what does not correspond to them.

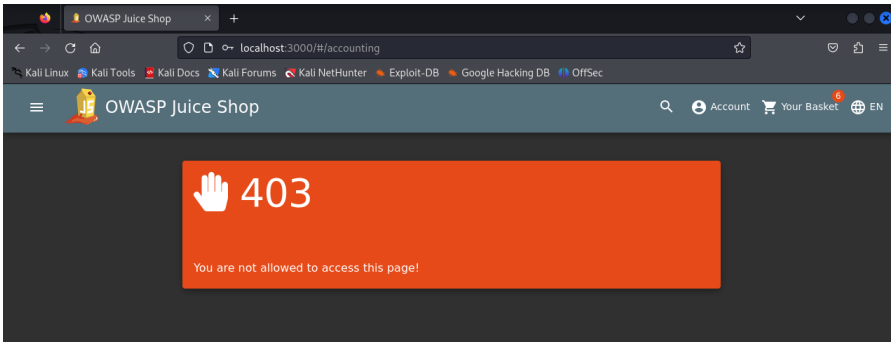

Figure 40: Admin cannot access endpoint ”accounting”

2.3.2 - Test User Registration Process

Some websites offer a user registration process that automates (or semi-automates) the provisioning of system access to users. The identity requirements for access vary from positive identification to none at all, depending on the security requirements of the system. Many public applications completely automate the registration and provisioning process because the size of the user base makes it impossible to manage manually. However, many corporate applications will provision users manually, so this test case may not apply. The goals of this test are to verify that the identity requirements for user registration are aligned with business and security requirements, and to validate the registration process.

First of all, one needs to go to the register page. Figure 41 provides an example of a successful registration on the web application:

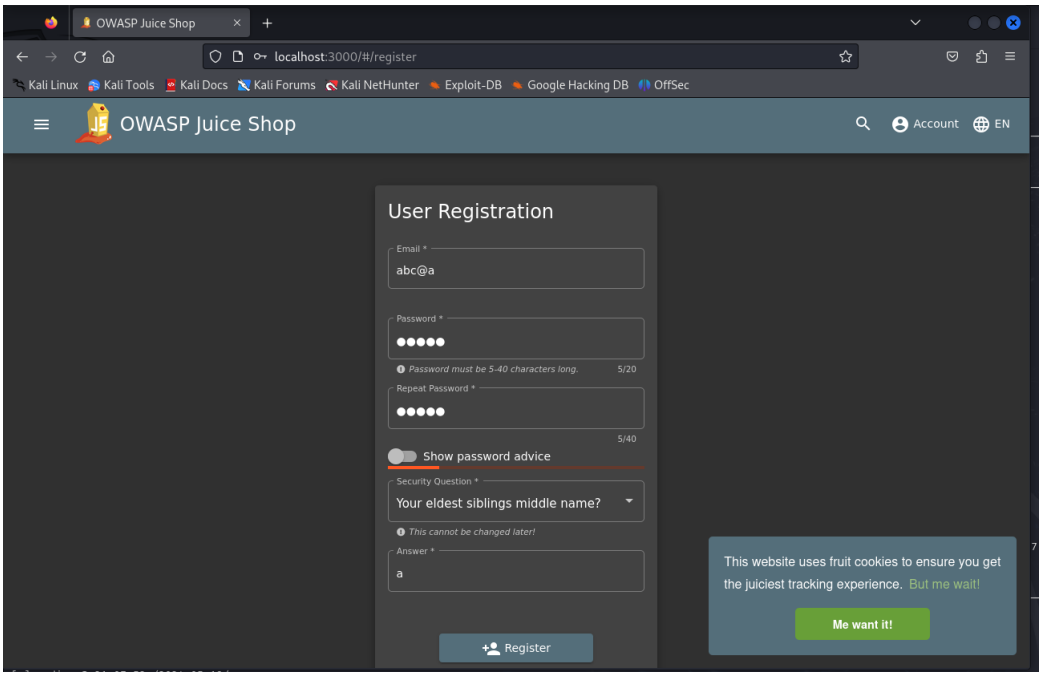

Figure 41: Registration example

One main aspect to consider is that the e-mail is accepted as long as the character ”@” is in the camp. The application does not require a real e-mail to accept the registration. Then, some questions need to be asked:

Can anyone register for access? - yes, anyone can go to the register page and create an account for access.

Are registrations vetted by a human prior to provisioning, or are they automatically granted if the criteria are met? - they are automatically granted if the criteria are met. Access becomes available there and then.

Can the same person or identity register multiple times? - yes, the same person can register multiple accounts, but the e-mail has to be different each time.

Can users register for different roles or permissions? - no, users can only register as customers.

What proof of identity is required for a registration to be successful? - no proof of identity is required. Only a security question must be filled out for when the application suspects that the account is being misused.

Are registered identities verified? - there is no mechanism of verification, every newly created account is at the same level as the others.

Can identity information be easily forged or faked? - yes, one can register with any e-mail, therefore faking identity information, as there is no verification mechanism in place.

2.3.3 - Test Account Provisioning Process

The provisioning of accounts presents an opportunity for an attacker to create a valid account without the application of the proper identification and authorization process. The aim of this test is to verify which accounts may provision other accounts and of what type.

The relevant questions are:

Is there any verification, vetting, and authorization of provisioning requests? - no.

Is there any verification, vetting, and authorization of de-provisioning requests? - no.

Can an administrator provision other administrators or just users? - just users.

Can an administrator or other user provision accounts with privileges greater than their own? - no.

2.3.4 - Testing for Account Enumeration and Guessable User Account

The scope of this test is to verify if it is possible to collect a set of valid usernames by interacting with the authentication mechanism of the application. This test will be useful for brute force testing, in which the tester verifies if, given a valid username, it is possible to find the corresponding password. The objectives of this test are to review processes that pertain to user identification and enumerate users where possible through response analysis.

To begin with, when testing for a valid user with a wrong password, the application will just return the same login screen, with a phrase saying ”invalid email or password”. This is a good practice, as an attacker cannot know which of the two is wrong.

As for testing for a nonexistent username, the procedure is equal. But if a user forgets the password, and tries to set a new one, if the email provided is not valid, the application will not let the request go through. Furthermore, the security question that needs to be given is displayed on the box. This is extremely dangerous, as it gives the attacker a context from where he can possibly get the answer. This part is illustrated in Figure 42:

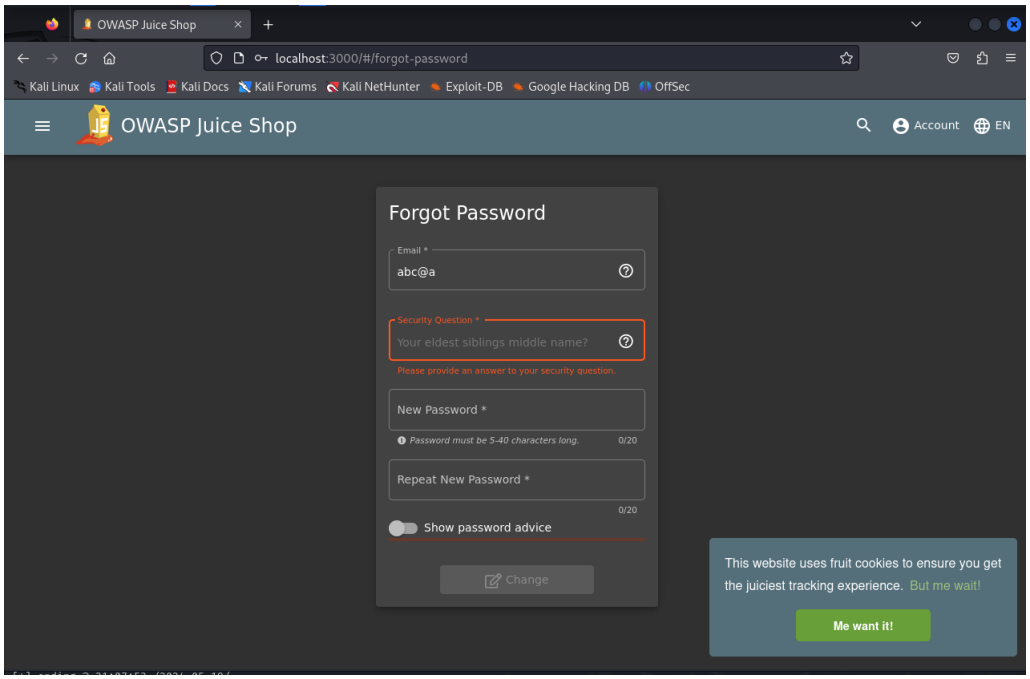

Figure 42: Forgot Password Page

Another aspect that must be mentioned is that, once an attacker has access to the admin account, he can easily get the list of all the other users, just by going to the administration page. Another way to list the users is by doing the procedure executed in 2.3.1, where the user roles were discovered with the tool sqlmap.

2.3.5 - Testing for Weak or Unenforced Username Policy

This section does not apply, since there are no usernames. The authentication is done with the e-mail and a password. Just one note, in the registration of a new account, the application actually has a mechanism that provides the user an insight into the strength of the password chosen, as well as suggestions to make the password as strong as possible, which are good security practices. The problem is that it ends up falling short in the obligations it sets for the password, as it only needs to be 5 characters long, and does not mandate a composition of more than just letters. The suggestions should be obligations. This just makes users choose an easy password to remember, instead of being forced to choose a secure one, despite usually being more complex to remember.

2.4 - WSTG - Authentication Testing

2.4.1 - Testing for Credentials Transported over an Encrypted Channel

Testing for credentials transport verifies that web applications encrypt authentication data in transit. This encryption prevents attackers from taking over accounts by sniffing network traffic. Web applications use HTTPS to encrypt information in transit for both client-to-server and server-to-client communications.

Since the OWASP Juice Shop was developed to only work with HTTP, this point is non-applicable.

2.4.2 - Testing for Default Credentials

Often these applications, once installed, are not properly configured and the default credentials provided for initial authentication and configuration are never changed. These default credentials are well known by penetration testers and, unfortunately, also by malicious attackers, who can use them to gain access to various types of applications.

This was the case for the admin account in this application. Its password was pretty basic, ”admin123”. Likewise, if a fuzzing brute force attack was made on other accounts, the likelihood of success would be very high, as passwords only need to be 5 characters long, without any other restriction. Performing manual tries on other popular default credentials did not have success, so brute force attacks are definitely the way in order to discover more credentials.

2.4.3 - Testing for Weak Lock Out Mechanism

Account lockout mechanisms are used to mitigate brute force attacks. They require a balance between protecting accounts from unauthorized access and protecting users from being denied authorized access. Accounts are typically locked after 3 to 5 unsuccessful attempts and can only be unlocked after a predetermined period of time, via a self-service unlock mechanism, or intervention by an administrator.

This topic will also be short. As the fuzzing brute force attack was successful, this evidently means that there is absolutely no lockout mechanism. An attacker can try to enter an account with the amount of tries they want. This represents a serious security problem, as brute force attacks can be performed quite easily.

2.4.4 - Testing for Bypassing Authentication Schema

In computer security, authentication is the process of attempting to verify the digital identity of the sender of a communication. A common example of such a process is the log on process. Testing the authentication schema means understanding how the authentication process works and using that information to circumvent

the authentication mechanism. The objective of this test is to ensure that authentication is applied across all services that require it.

Starting with direct page request and parameter modification, when an attempt is made to directly access a protected page through the address bar in the browser to bypass authentication, nothing happens, the user stays in the home page. Regarding SQL Injection, it was already seen by the tests made in the OWASP ZAP section that the application is indeed vulnerable to different types of SQL Injection, with the authentication phase being completely bypassed. See 2.1.4.

2.4.5 - Testing for Vulnerable Remember Password

Credentials are the most widely used authentication technology. Due to such a wide usage of username-password pairs, users are no longer able to properly handle their credentials across the multitude of used applications. So now, applications usually provide a remember me functionality that allows the user to stay authenticated for long periods of time, without asking the user again for their credentials. The objective of this test is to validate that the generated session is managed securely and does not put the user's credentials in danger.

To test this, two logins are to be performed, one with the remember me checkbox active, and another with it inactive. Then, using the programmer tools, in the tab local storage, some conclusions might be able to be taken. Figures 43 and 44 show the results with remember me on and off, respectively:

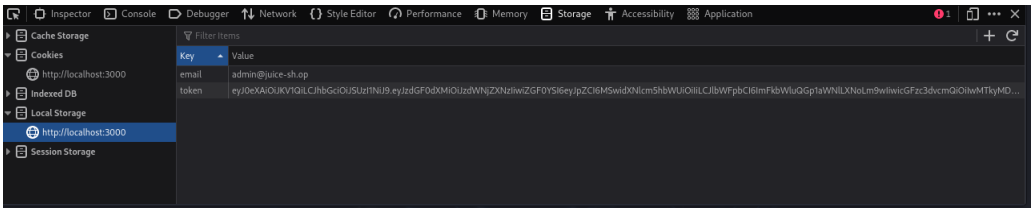

Figure 43: Logging in with remember me on

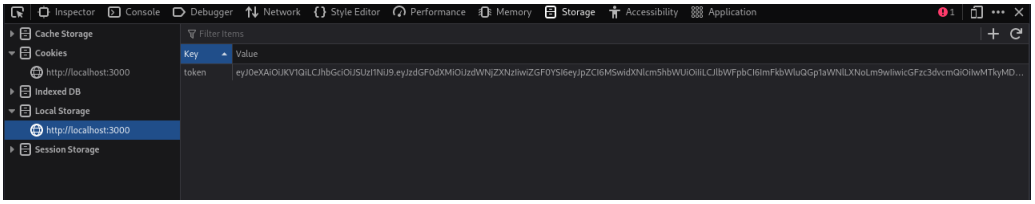

Figure 44: Logging in with remember me off

As one can see, the email becomes visible in the local storage when logging in with remember me activated. This is bad, as the local storage is accessible by any JavaScript running on the same domain, making it vulnerable to XSS attacks. Also, data in local storage persists until explicitly deleted, meaning sensitive information could remain on a device for an extended period.

2.4.6 - Testing for Browser Cache Weaknesses

Browsers can store information for purposes of caching and history. Checks are made to ensure that the application correctly instructs the browser to not retain sensitive data. The objective of this test is to review if the application stores sensitive information on the client-side and review if access can occur without authorization.

Firstly, browser history is going to be looked at, by navigating to pages with sensitive information of a user, logging out, and then trying to go Back to see if the application lets it happen or not. What happens is not what is desired. Once logged out, going back to the user profile still works, and the user information is being displayed to someone no longer authenticated, like Figure 45 exhibits:

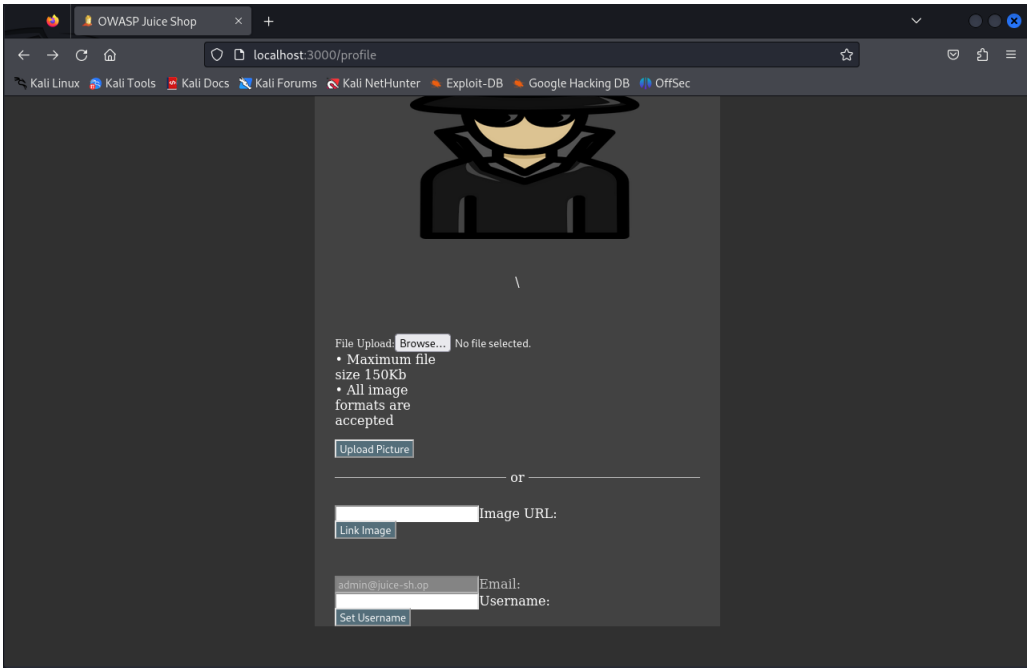

Figure 45: Accessing the admin profile while logged out

Moving on to browser cache, cache storage never stores anything while a person browses through the application. Cache validity in the responses given from the website is always set to 0, meaning it does not store data. This shows a good implementation of browser cache.

2.4.7 - Testing for Weak Password Policy

The most prevalent and most easily administered authentication mechanism is a static password. The password represents the keys to the kingdom, but is often subverted by users in the name of usability. In each of the recent high-profile hacks that have revealed user credentials, it is lamented that the most common passwords are still: 123456, password, and qwerty. The goal of this test is to determine the resistance of the application against brute force password guessing using available password dictionaries by evaluating the length, complexity, reuse, and aging requirements of passwords.

Once again, questions need answers:

What characters are permitted and forbidden for use within a password? Is the user required to use characters from different character sets such as lower and uppercase letters, digits and special symbols? - the user is only required to have a minimum length of 5 characters. There are no forbidden characters.

How often can a user change their password? How quickly can a user change their password after a previous change? - whenever they feel like doing so. No limitation is present.

When must a user change their password? - the application does not force the user to change their password. The first password can stay on for an undefined amount of time.

How often can a user reuse a password? - whenever the user wants. The system does not detect reused passwords.

How different must the next password be from the last password? - it can be the same password again. The system does not detect this.

Is the user prevented from using his username in the password? - no.

What are the minimum and maximum password lengths that can be set? - minimum of 5 characters, maximum of 40.

Is it possible set common passwords such as Password1 or 123456? - yes.

2.4.8 - Testing for Weak Security Question Answer

Often called "secret" questions and answers, security questions and answers are often used to recover forgotten passwords (see Testing for weak password change or reset functionalities, or as extra security on top of the password. The aim of this test is to determine the complexity and how straightforward the questions are and assess possible user answers and brute force capabilities.

Security questions are, in fact, implemented by the application, and required when a user is registering his account. 14 different questions exist, but there is no limitation or obligation that the user must obey when filling in his security question. Therefore, these answers might be weak and susceptible to brute force attacks. Moreover, the user cannot create his own question, and there is no limit set for the amount of tries a user can have at getting the answer right when changing password.

2.4.9 - Testing for Weak Password Change or Reset Functionalities

The password change and reset function of an application is a self-service password change or reset mechanism for users. This self-service mechanism allows users to quickly change or reset their password without an administrator intervening.

There is no password reset function in the Juice Shop application. As for password change, it does in fact request the current password in order to allow the change to happen.

2.4.10 - Testing for Weaker Authentication in Alternative Channel

Even if the primary authentication mechanisms do not include any vulnerabilities, it may be that vulnerabilities exist in alternative legitimate authentication user channels for the same user accounts.

This topic is non-applicable to the OWASP Juice Shop, as it does not implement alternative authentication user channels.

2.5 - WSTG - Authorization Testing

2.5.1 - Testing Directory Traversal File Include

Many web applications use and manage files as part of their daily operation. Using input validation methods that have not been well designed or deployed, an aggressor could exploit the system in order to read or write files that are not intended to be accessible. In particular situations, it could be possible to execute arbitrary code or system commands. The goals of this test are to identify injection points that pertain to path traversal and assess bypassing techniques and identify the extent of path traversal.

For starters, one needs to look at endpoints that accept filenames as input. The one that comes up pretty quickly is the **/complain** endpoint, where a user can make a complaint to the shop and include files in that same complaint. From there, a test was made by sending a file on a complaint to see if it ended up on the /ftp endpoint, where files are stored by the system as discovered earlier, but nothing reached that location.

Therefore, a zip file is going to be made, including a random .txt file, and using the tool curl, a path traversal will be attempted a bit blindly. Figure 46 illustrates this event:

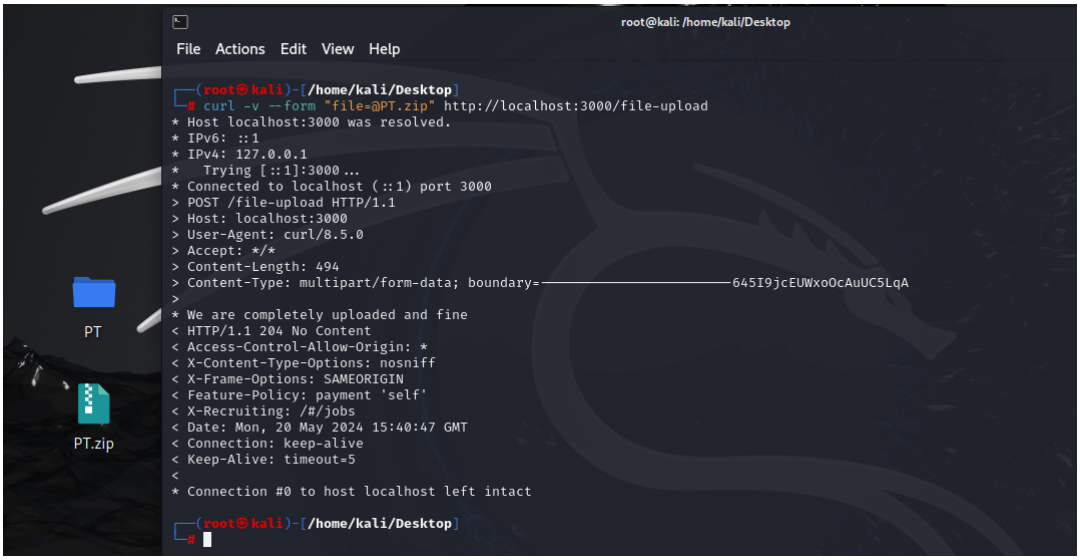

Figure 46: Path Traversal attempt (1)

Still, nothing appeared in the ftp location. So, **../** will keep on being added to the file inside the test zip, as Figure 47 displays:

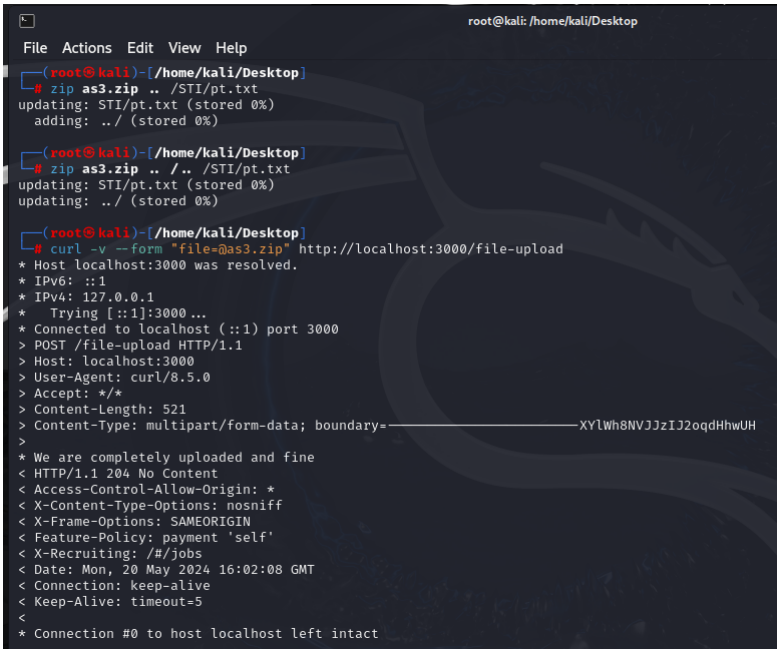

Figure 47: Path Traversal attempt (2)

Fortunately, the file did not end up in the /ftp endpoint after all of these efforts, confirming that the Juice Shop is indeed secure against Path Traversal.

2.5.2 - Testing for Bypassing Authorization Schema

This kind of test focuses on verifying how the authorization schema has been implemented for each role or privilege to get access to reserved functions and resources. The objective of this test is to assess if horizontal or vertical access is possible.

The majority of the requirements of this test were already fulfilled in section 2.3.1 - User Roles. As for additional information, the application is vulnerable to a series of unauthorized accesses, like, for example, once again, accessing personal information while not logged out, or even being able to upload files as a mere visitor, using the ”go back” mechanism. All of these represent bypasses of authorization.

2.5.3 - Testing for Privilege Escalation

This section describes the issue of escalating privileges from one stage to another. During this phase, the tester should verify that it is not possible for a user to modify their privileges or roles inside the application in ways that could allow privilege escalation attacks. The goals are to identify injection points related to privilege manipulation and fuzz or otherwise attempt to bypass security measures.

Again, this point was already achieved in the OWASP ZAP testing, by successfully performing SQL Injections and brute force attacks that consequently gave away a privilege escalation. Another way to reach privilege escalation is by changing the role of a user in registration. Figure 48 illustrates the command needed to discover this fact:

Figure 48: Discovering how to change role of a user

The image shows that there is a ”role” field in the payload, which can be manipulated by intercepting the registration requests, and adapted to whichever role is desired.

2.5.4 - Testing for Insecure Direct Object References

Insecure Direct Object References (IDOR) occur when an application provides direct access to objects based on user-supplied input. As a result of this vulnerability, attackers can bypass authorization and access resources in the system directly, for example, database records or files.

This test can be verified in the User Roles section, 2.3.1. The parameter **q** of the URL http://localhost:3000/rest/products/search?q= is so easily manipulated to the point that it can include SQL Injections targeting back-end code.

2.6 - WSTG - Session Management Testing

2.6.1 - Testing for Session Management Schema

One of the core components of any web-based application is the mechanism by which it controls and maintains the state for a user interacting with it. To avoid continuous authentication for each page of a website or service, web applications implement various mechanisms to store and validate credentials for a pre-determined timespan. These mechanisms are known as Session Management. The test objectives are to gather session tokens, for the same user and for different users where possible, analyze and ensure that enough randomness exists to stop session forging attacks, and modify cookies that are not signed and contain information that can be manipulated.

In 2.4.5, when testing the remember me feature, one could also see in the local storage the authentication session tokens of the users performing the login. Using the website **https://www.base64decode.org/**, part of the token can be decoded, as one can verify in Figure 49:

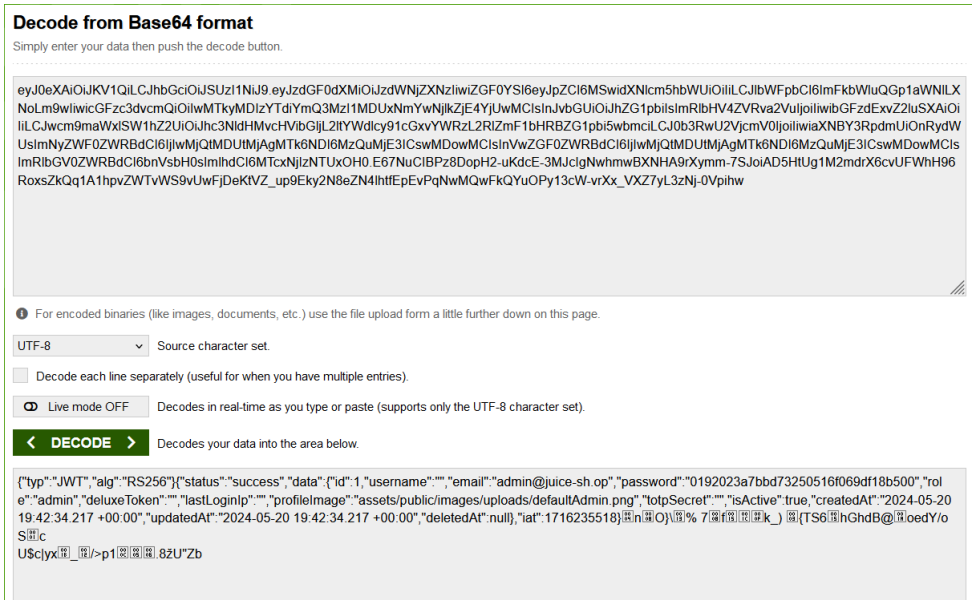

Figure 49: Decoding the session token

There is clearly a lot of information available once the token is decoded, like the email, the password string, and the user role, among others. This shows a lack of security in the sessions of the application.

2.6.2 - Testing for Cookies Attributes

Web Cookies (referred to as cookies) are often a key attack vector for malicious users (typically targeting other users) and the application should always take due diligence to protect cookies. The aim of this test is to ensure that the proper security configuration is set for cookies.

By accessing the web developer tools, one is able to see the cookies gathered as a user browses through the application. This is shown by Figure 50:

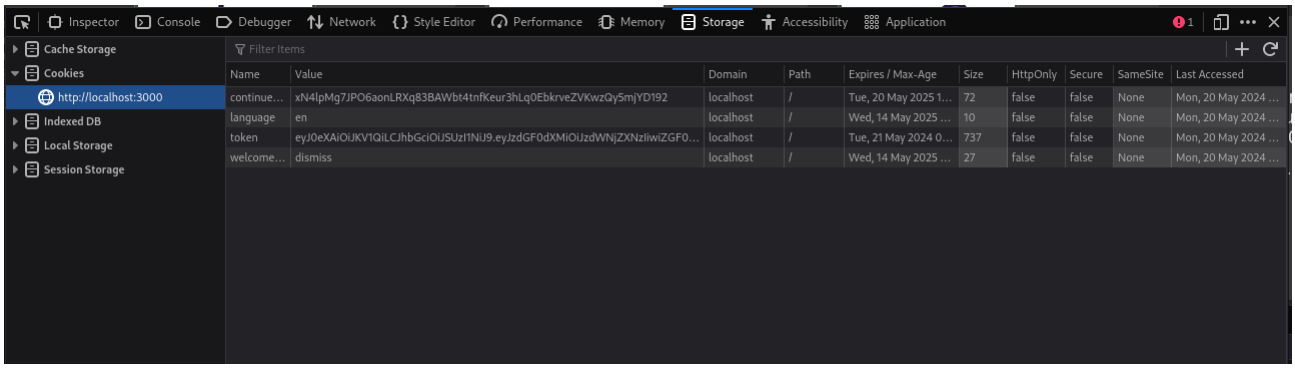

Figure 50: Accessing the cookies

These cookies contain important tabs:

Secure - attribute that indicates to the browser to send the cookies only if those are transmitted over a safe channel. As every one of these is set to False, cookies are subject to attacks.

HttpOnly - prevents session leakage attacks. Every attribute is set to false, so the application is vulnerable to these attacks.

Expires - gives the expiration date of the cookies. It is not recommended to have this as a high value, because the cookies will be stored for that duration of time. The application, in fact, has long expiration dates.

SameSite - aims to ensure that cookies are not sent in cross-site requests. The values are set to None, which means that the Juice Shop is vulnerable to cross-origin information leakage.

2.6.3 - Testing for Session Fixation

Session fixation is enabled by the insecure practice of preserving the same value of the session cookies before and after authentication. This typically happens when session cookies are used to store state information even before login, e.g., to add items to a shopping cart before authenticating for payment. The goals of this test are to analyze the authentication mechanism and its flow, and to force cookies and assess the impact.

Well, after logging out of the application, two main things happen. The session tokens disappear, meaning that the information present in them is no longer vulnerable, which is a good thing. On the other hand, some of the cookies stay present, meaning that the information about possible attacks on the cookies is still available, as is the risk associated.

2.6.4 - Testing for Exposed Session Variables

The Session Tokens (Cookie, SessionID, Hidden Field), if exposed, will usually enable an attacker to impersonate a victim and access the application illegitimately. It is important that they are protected from eavesdropping at all times, particularly whilst in transit between the client browser and the application servers.

The OWASP Juice Shop does not reutilize session tokens or IDs, which is a good practice. Also, the methods GET and POST are used correctly when requested, despite giving away some information about the application.

2.6.5 - Testing for Cross Site Request Forgery

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unintended actions on a web application in which they are currently authenticated. With a little social engineering help (like sending a link via email or chat), an attacker may force the users of a web application to execute actions of the attacker's choosing.

A successful CSRF exploit can compromise end user data and operation when it targets a normal user. If the targeted end user is the administrator account, a CSRF attack can compromise the entire web application. The goal is to determine whether it is possible to initiate requests on a user's behalf that are not initiated by the user.

To test CSRF, a malicious form is going to be created to try and change the username of a logged-in user. Figure 51 shows the form used:

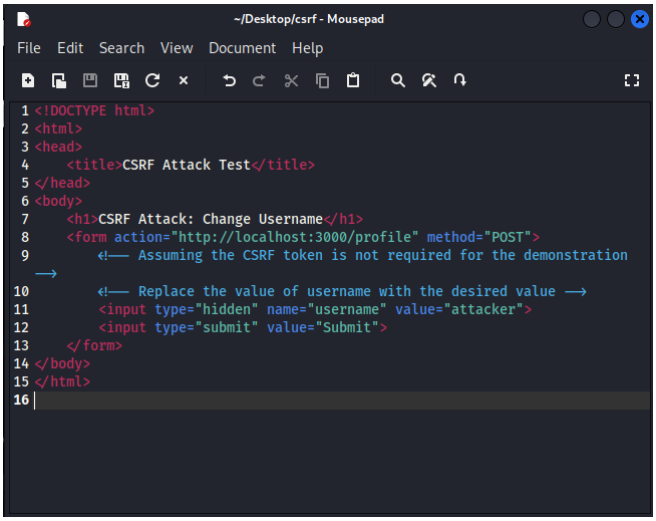

Figure 51: Malicious HTML form to change the username

The username set was just ”admin”. The HTML script was opened on the internet and run. Figure 52 contains the result:

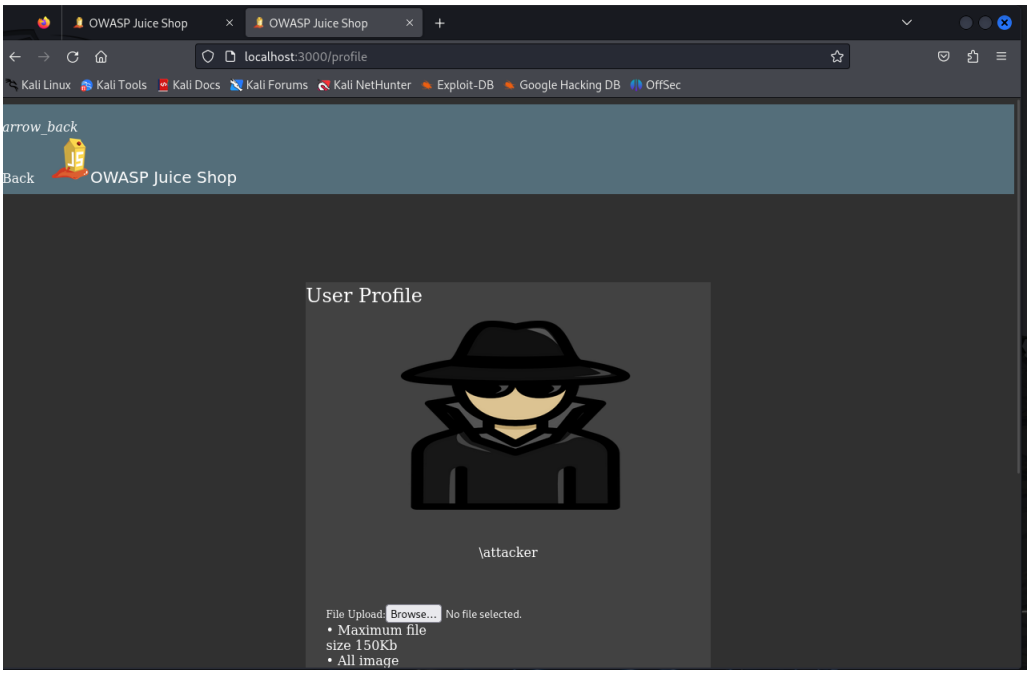

Figure 52: Result of the CSRF attack

Now it is confirmed. The application is vulnerable to CRSF attacks.

2.6.6 - Testing for Logout Functionality

Session termination is an important part of the session lifecycle. Reducing to a minimum the lifetime of the session tokens decreases the likelihood of a successful session hijacking attack. This can be seen as a control against preventing other attacks like Cross Site Scripting and Cross Site Request Forgery. Such attacks have been known to rely on a user having an authenticated session present. Not having a secure session termination only increases the attack surface for any of these attacks. The objective of this test is to analyze the session timeout and if the session is properly killed after logout.

As already seen in 2.6.3, after a user logs out, the session tokens disappear, meaning the log-out occurs in a safe manner. Regarding the logout interface, the button does not appear immediately. Instead one needs to click on the account button first, to open up a series of options, among which is the log-out button. As for server-side termination, some cookies disappear once a user exits the application, and others stay.

2.6.7 - Testing Session Timeout

All applications should implement an idle or inactivity timeout for sessions. This timeout defines the amount of time a session will remain active in case there is no activity by the user, closing and invalidating the session upon the defined idle period since the last HTTP request received by the web application for a given session ID. The most appropriate timeout should be a balance between security (shorter timeout) and usability (longer timeout) and heavily depends on the sensitivity level of the data handled by the application.

At this point of the assignment, there is no evidence that the system possesses a session timeout mechanism. Therefore, this section is going to be marked as inconclusive. The good practice should be to have a session timeout that balances security and usability, as said above.

2.6.8 - Testing for Session Puzzling

Session Variable Overloading (also known as Session Puzzling) is an application-level vulnerability which can enable an attacker to perform a variety of malicious actions. This vulnerability occurs when an application uses the same session variable for more than one purpose. An attacker can potentially access pages in an order unanticipated by the developers so that the session variable is set in one context and then used in another.

The only session variable that exists in the Juice Shop is the session token that identifies the session of a user. As this token is used on various pages of the application, possible attackers could intercept requests made, for example, when a user submits a complaint, and get hold of the session token. It must be said though that the best way to test this type of vulnerability is via a gray-box approach.

2.6.9 - Testing for Session Hijacking

An attacker who gets access to user session cookies can impersonate them by presenting such cookies. This attack is known as session hijacking. When considering network attackers, i.e., attackers who control the network used by the victim, session cookies can be unduly exposed to the attacker over HTTP. To prevent this, session cookies should be marked with the Secure attribute so that they are only communicated over HTTPS. The objectives are to identify vulnerable session cookies, hijack vulnerable cookies, and assess the risk level.

To begin with, a session token is going to be captured in the web developer tools. Then, said token is going to be set in a new window, like Figure 53 displays:

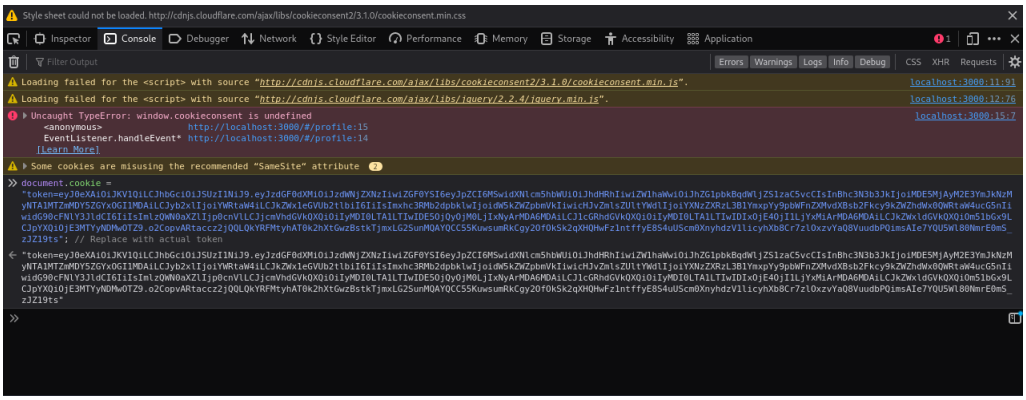

Figure 53: Setting a captured session toke on a new window

With this process executed, a different user can now hijack the session of the user whose token was stolen. The application is vulnerable to session hijacking.

2.7 - WSTG - Input Validation Testing

2.7.1 - Testing for Reflected Cross Site Scripting

Reflected Cross-site Scripting (XSS) occur when an attacker injects browser executable code within a single HTTP response. The injected attack is not stored within the application itself; it is non-persistent and only impacts users who open a maliciously crafted link or third-party web page. The attack string is included as part of the crafted URI or HTTP parameters, improperly processed by the application, and returned to the victim.

The easiest way to perform this attack is once again using the URL **http://localhost:3000/#/search?q=**, where following the ”q” will be a script corresponding to the attack. Figure 54 shows an XSS attack with a payload of an image tag with OnError event:

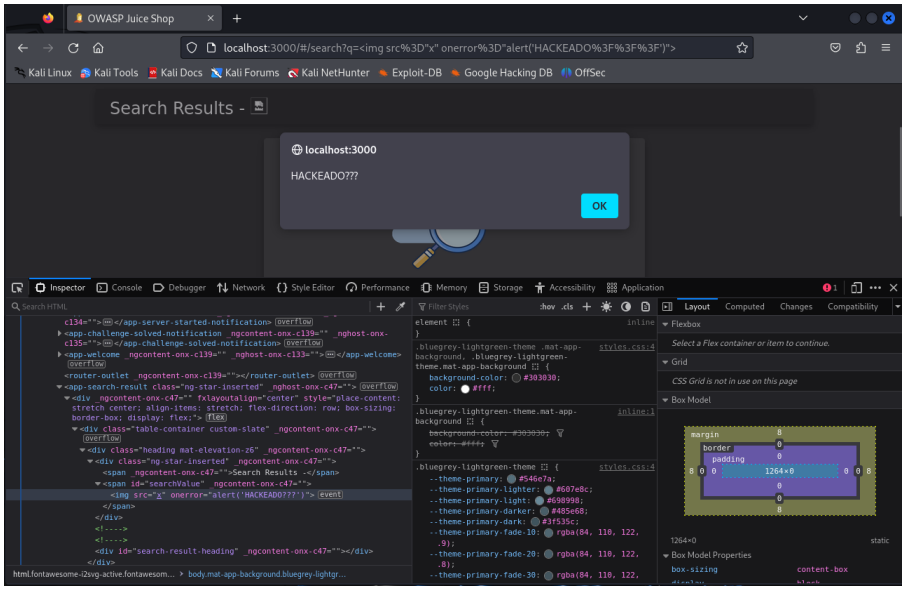

Figure 54: XSS attack

2.7.2 - Testing for Stored Cross Site Scripting

Stored Cross-site Scripting (XSS) is the most dangerous type of Cross Site Scripting. Web applications that allow users to store data are potentially exposed to this type of attack. Stored XSS occurs when a web application gathers input from a user which might be malicious, and then stores that input in a data store for later use. The input that is stored is not correctly filtered. As a consequence, the malicious data will appear to be part of the website and run within the user's browser under the privileges of the web application.

An attempt to perform this attack was made in the /contact endpoint, where everyone can send feedback to the system. Figure 55 contains the attempt:

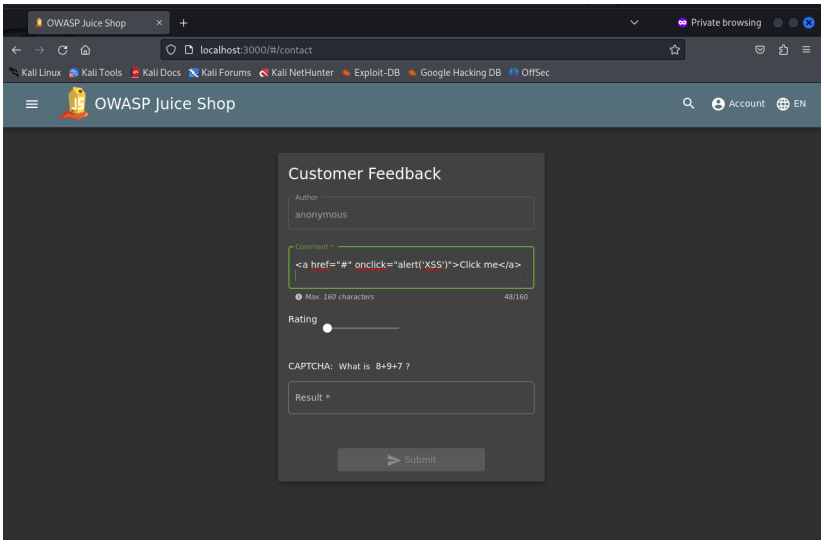

Figure 55: Stored XSS attack attempt

The results were not conclusive, as on the administration page a ”click me” button appeared stored along with the feedback, but if a user clicks on it, it is redirected to the home page, and no pop-up appears. Figure 56 contains the feedback stored, with the ”click me” button. Still, there are suspicions that this vulnerability might exist, so the test is considered inconclusive.

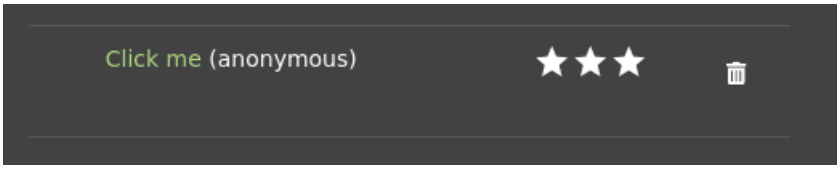

Figure 56: Stored ”click me” button

2.7.3 - Testing for HTTP Verb Tampering

Non-Applicable. Section covered in 2.2.2.

2.7.4 - Testing for HTTP Parameter Pollution

HTTP Parameter Pollution tests the application response to receiving multiple HTTP parameters with the same name; for example, if the parameter username is included in the GET or POST parameters twice.

This topic has already been dealt with during the making of the assignment. Some examples of HTTP Parameter Pollution are the URLs **http://localhost:3000/rest/products/search?q=** and **http://localhost:3000/search?q=**.

Also if search parameters are duplicated, like in the URL **http://localhost:3000/#/search?q=apple&q=banana**, nothing supposedly happens, as the page that is returned is the list of all products available.

2.7.5 - Testing for SQL Injection

At this stage of the assignment, it is already known that the SQL type used by the OWASP Juice Shop is SQLite corresponding to a local database, therefore, only section ”2.7.5.3 - Testing for SQL Server” applies here.

2.7.5.3 - Testing for SQL Server

SQL injection vulnerabilities occur whenever input is used in the construction of an SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers and execute SQL code under the privileges of the user used to connect to the database.

Most of the topics to be addressed here have already been discovered elsewhere. OWASP ZAP discovered right in the beginning that the application is vulnerable to several SQL Injections. Besides, ”2.3.1 - User Roles” uses the tool sqlmap to discover the application database, containing all tables with information such as user information, product information, complaints, and others.

2.8 - WSTG - Testing for Error Handling

2.8.1 - Testing for Improper Error Handling

All types of applications (web apps, web servers, databases, etc.) will generate errors for various reasons. Developers often ignore handling these errors, or push away the idea that a user will ever try to trigger an error purposefully (e.g. sending a string where an integer is expected). When the developer only considers the happy path, they forget all other possible user input the code can receive but can't handle.

An example of present improper error handling, in the form of a stack trace, is when trying to access an inexistent file in the **/ftp** endpoint, which gives away information. This is illustrated by Figure 57:

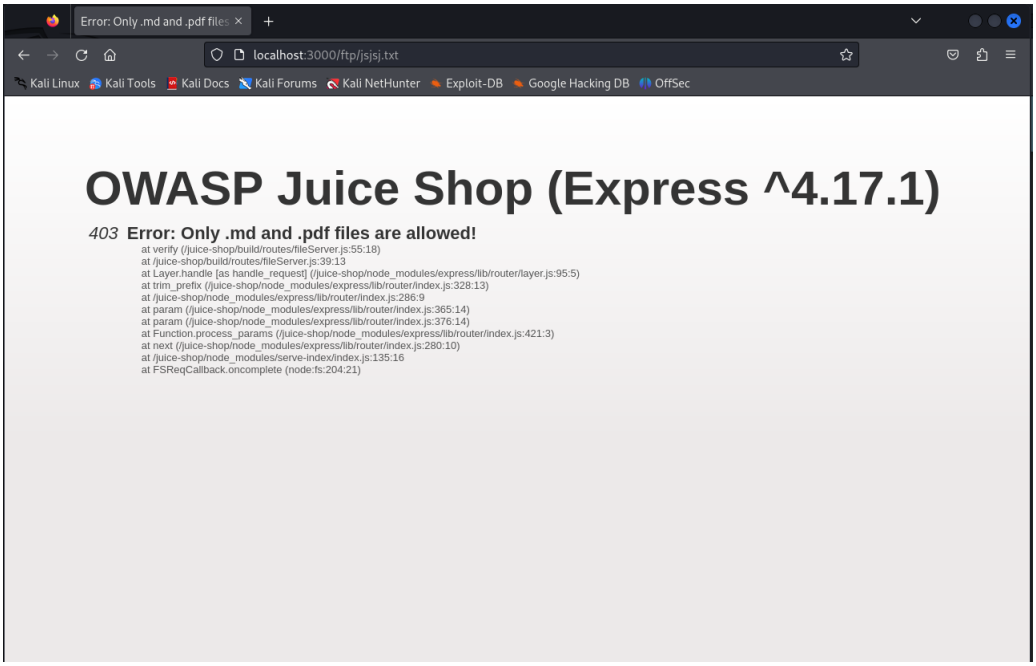

Figure 57: Improper Error Handling

2.8.2 - Testing for Stack Traces

Non-applicable. The section is covered in 2.8.1.

3 - Web application security firewall

Now for the third phase of the assignment, the same tests previously run are applied again to the web application, but this time, it has the WAF configured to protect it from most of these attacks. From the results, some conclusions will be made about the effectiveness of the WAF.

3.1 - OWASP ZAP penetration tests

Starting once more with the OWASP ZAP tool, the same tests are going to be applied, with the same purpose as before.

3.1.1 - Automated Scan

Figure 58 contains the results of the automated scan performed, with the same configurations of the scan made in 2.1.1:

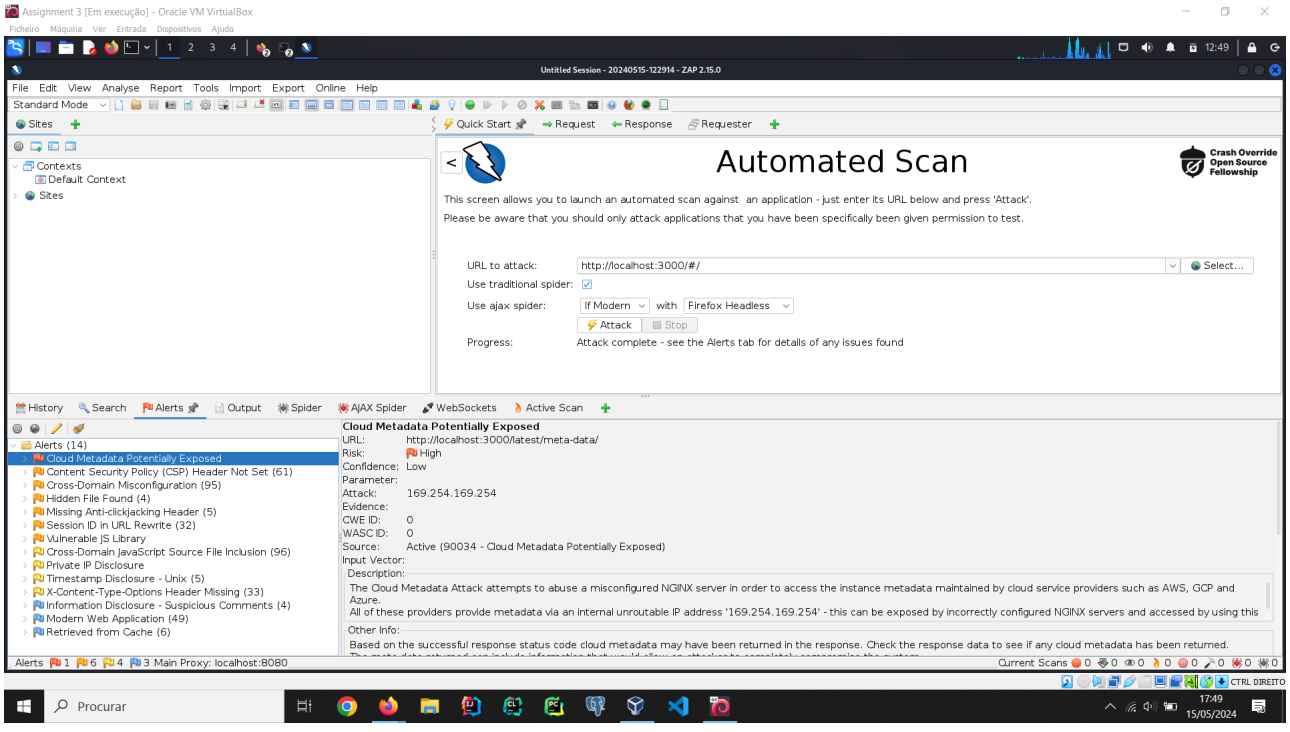

Figure 58: Automated Scan to localhost:80

13 alerts were found, one less than without the WAF. All in all, this indicates that the WAF probably suppressed one of the alerts previously found, but the results were very similar.

3.1.2 - Active Scan exploring the most effective policies

Once again, the same policies were applied to the scans, this time with the WAF implemented. The results are illustrated by Figures 59 and 60:

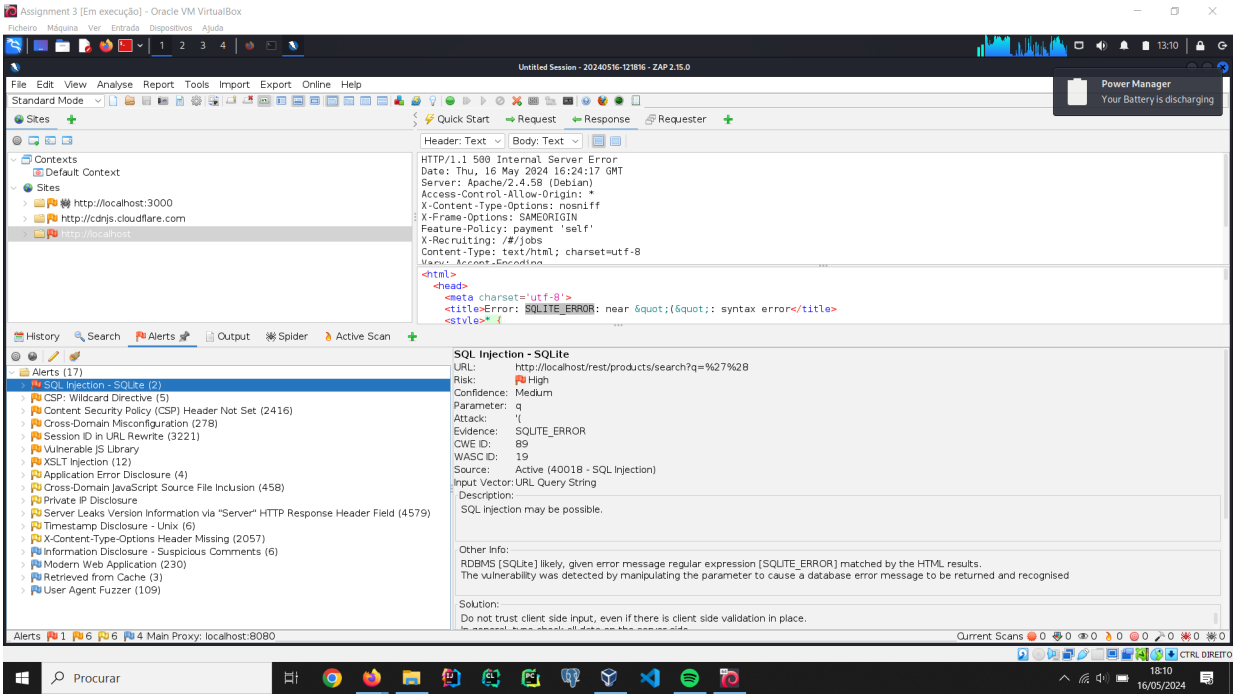

Figure 59: WAF - Scan with the High Strength / Default Threshold policy

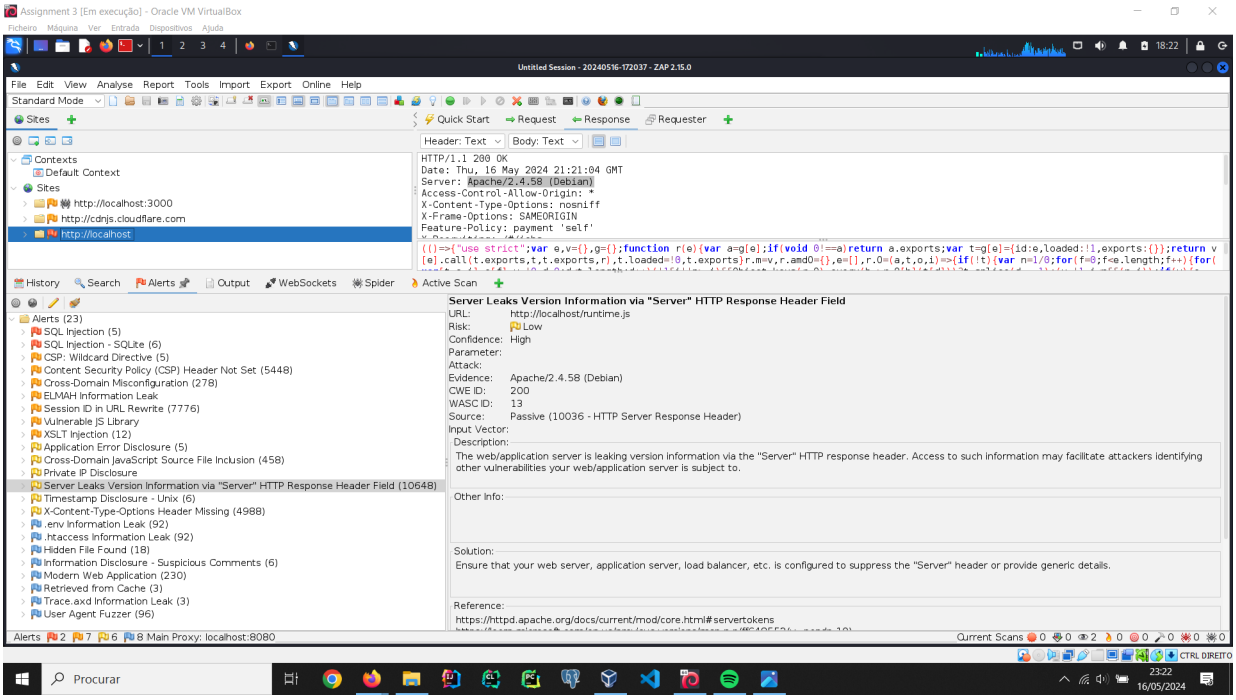

Figure 60: WAF - Scan with the Insane Strength / Low Threshold policy

The results are the same as before. The average policy only detects 17 alerts while the highest policy detects 23. As for differences between the WAF and no WAF components, the alerts are pretty much the same.

3.1.3 - Automated Scan with new Add-Ons

Using the same add-ons, the automated scan was performed on the application protected with the WAF, pictured by Figure 61:

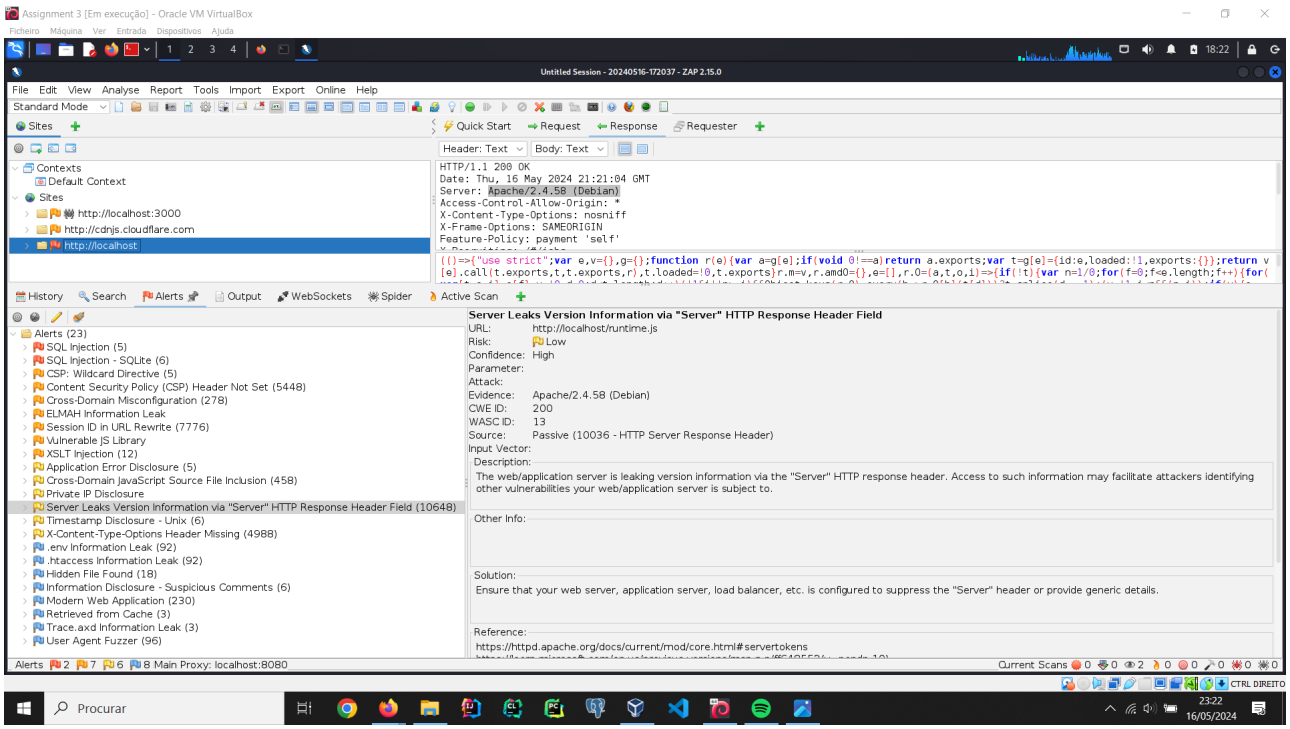

Figure 61: WAF - Automated Scan with add-ons

These results show even more alerts than in the version without the WAF. This is probably due to the fact that when in port 80, the application continues to be crawled even after the scan. Nonetheless, the add-ons were once again successful.

3.1.4 - Fuzz attacks to the login form

Again, SQL Injection and brute force attack fuzz scans are to be performed. Starting with the SQL Injection, Figures 62, 63, and 64 have the results of the scan:

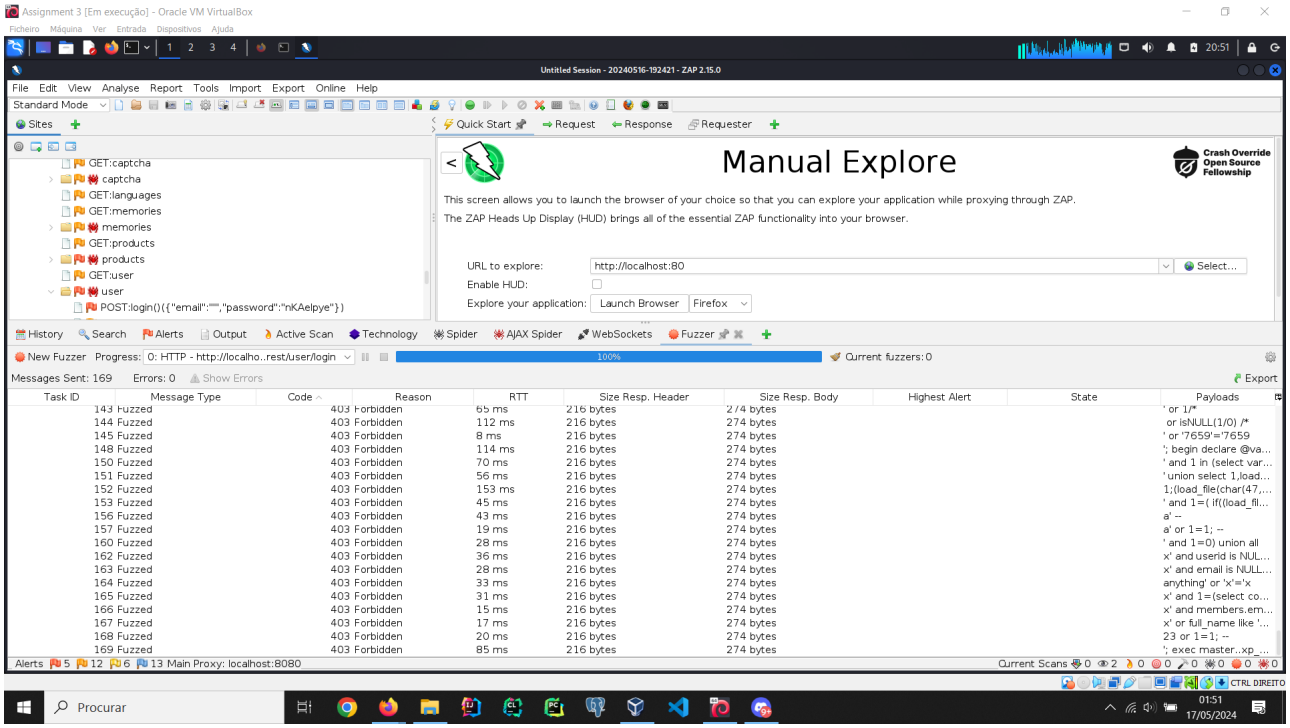

Figure 62: WAF - Results of the SQL Injection fuzzing scan (1)

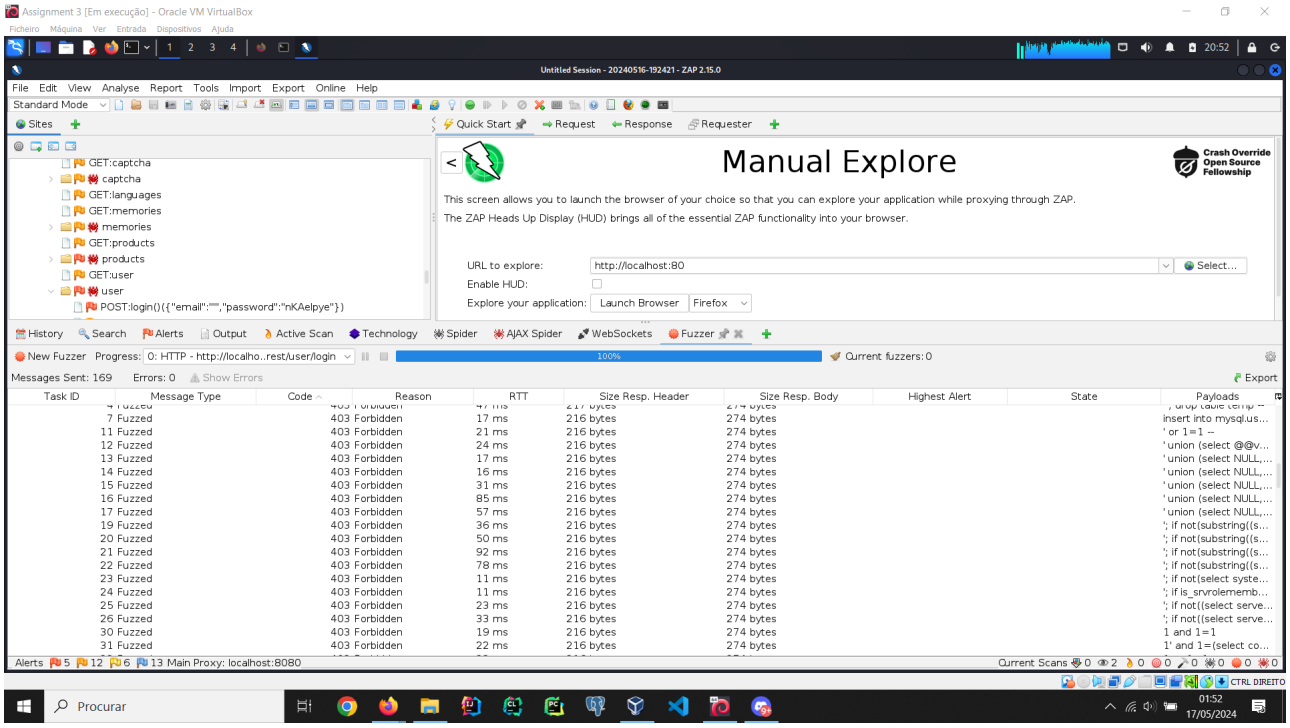

Figure 63: WAF - Results of the SQL Injection fuzzing scan (2)

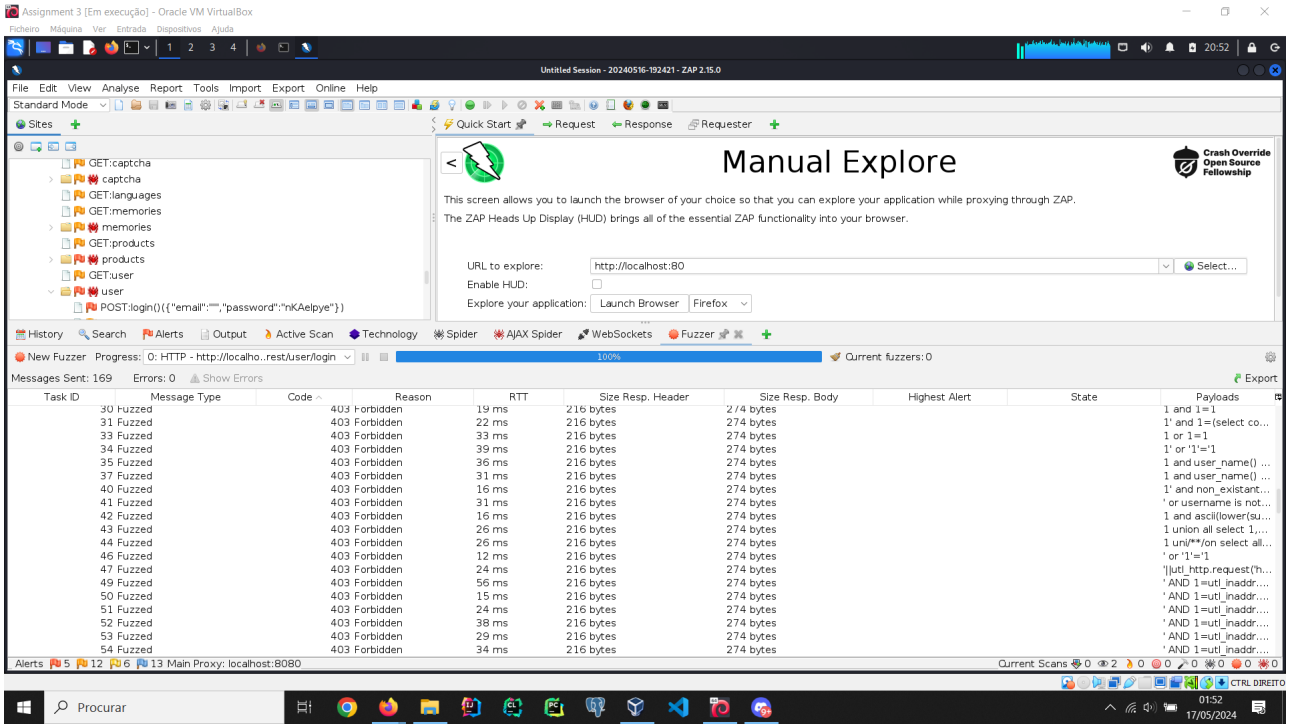

Figure 64: WAF - Results of the SQL Injection fuzzing scan (3)

The WAF worked as intended. Every payload returned a **403 Forbidden** code, meaning that the WAF defended against all the injections.

As for the brute force attack, things are a little different. Figure 65 pictures the scenario:

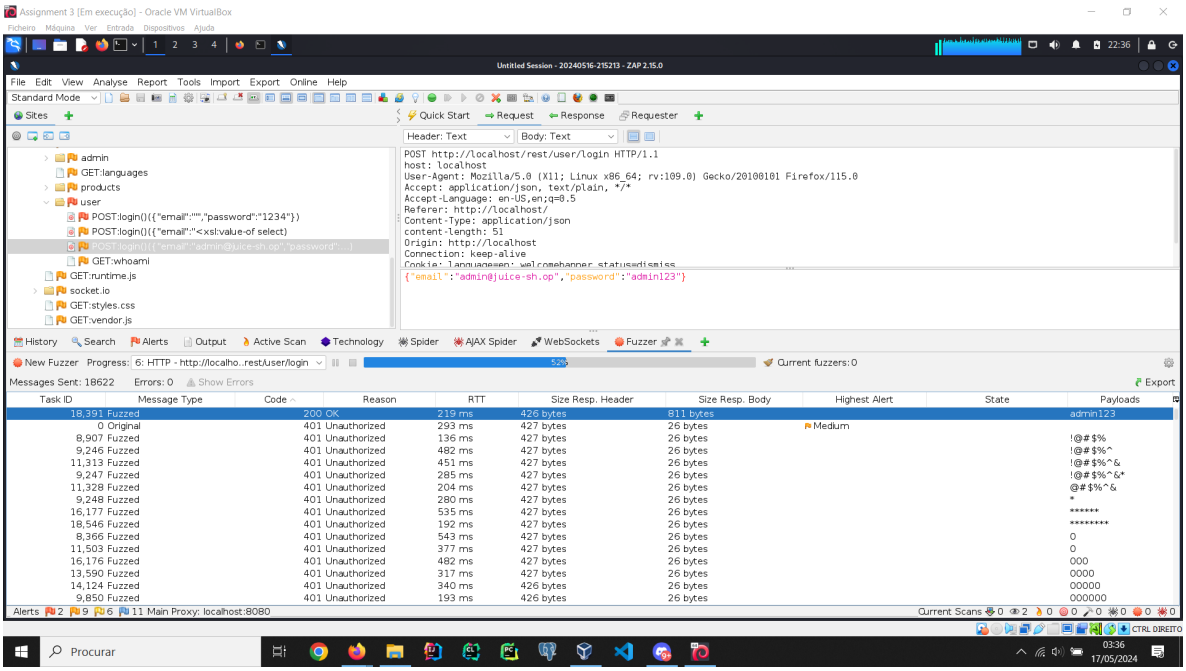

Figure 65: WAF - Results of the brute force fuzzing scan

This time the WAF could not protect the web application. The same password, **admin123**, was found, meaning the attacker can successfully enter the system in both versions of the web application.

3.1.5 - Manual tests to explore logged-in threats

Not much to say here, as the SQL Injection was successfully stopped by the WAF, and the results of the brute force attack are already displayed in section 2.1.5. Nothing changes here, once the attacker has the password he can get inside.

3.1.6 - Active Scan to explore authenticated area

As for the last part of the OWASP ZAP tests concerning the WAF-protected web application, the same active scan with the same context applied was run. The results were equal to those of 2.1.6, so visit that section for more information.

3.2 - WSTG - Configuration and Deployment Management Testing

3.2.1 - Enumerate Infrastructure and Application Admin Interfaces

The goal is to see if the WAF blocks the dirb command and, eventually, the access to the **/ftp** path. Figures 66 and 67 provide the answers to both questions, respectively:

```
┌──(root㉿kali)-[/usr/share/dirb]
└─# dirb http://localhost:80 /usr/share/dirb/wordlists/common.txt

-----------------
DIRB v2.22
By The Dark Raver
-----------------

START_TIME: Sun May 19 15:27:12 2024
URL_BASE: http://localhost:80/
WORDLIST_FILES: /usr/share/dirb/wordlists/common.txt

-----------------

GENERATED WORDS: 4612

---- Scanning URL: http://localhost:80/ ----
+ http://localhost:80/.config (CODE:403|SIZE:274)
+ http://localhost:80/_vti_bin/_vti_adm/admin.dll (CODE:403|SIZE:274)
+ http://localhost:80/_vti_bin/_vti_aut/author.dll (CODE:403|SIZE:274)
+ http://localhost:80/_vti_bin/shtml.dll (CODE:403|SIZE:274)
+ http://localhost:80/akeeba.backend.log (CODE:403|SIZE:274)
+ http://localhost:80/assets (CODE:301|SIZE:179)
+ http://localhost:80/awstats.conf (CODE:403|SIZE:274)
+ http://localhost:80/development.log (CODE:403|SIZE:274)
+ http://localhost:80/ftp (CODE:200|SIZE:11071)
+ http://localhost:80/global.asa (CODE:403|SIZE:274)
+ http://localhost:80/global.asax (CODE:403|SIZE:274)
+ http://localhost:80/main.mdb (CODE:403|SIZE:274)
+ http://localhost:80/production.log (CODE:403|SIZE:274)
+ http://localhost:80/profile (CODE:500|SIZE:1154)
+ http://localhost:80/promotion (CODE:200|SIZE:6586)
+ http://localhost:80/redirect (CODE:500|SIZE:3119)
+ http://localhost:80/robots.txt (CODE:200|SIZE:28)
+ http://localhost:80/snippets (CODE:200|SIZE:792)
+ http://localhost:80/spamlog.log (CODE:403|SIZE:274)
+ http://localhost:80/video (CODE:200|SIZE:10075518)
+ http://localhost:80/Video (CODE:200|SIZE:10075518)
+ http://localhost:80/WS_FTP.LOG (CODE:403|SIZE:274)

-----------------
END_TIME: Sun May 19 15:29:11 2024
DOWNLOADED: 4612 - FOUND: 22
```

Figure 66: WAF - dirb command

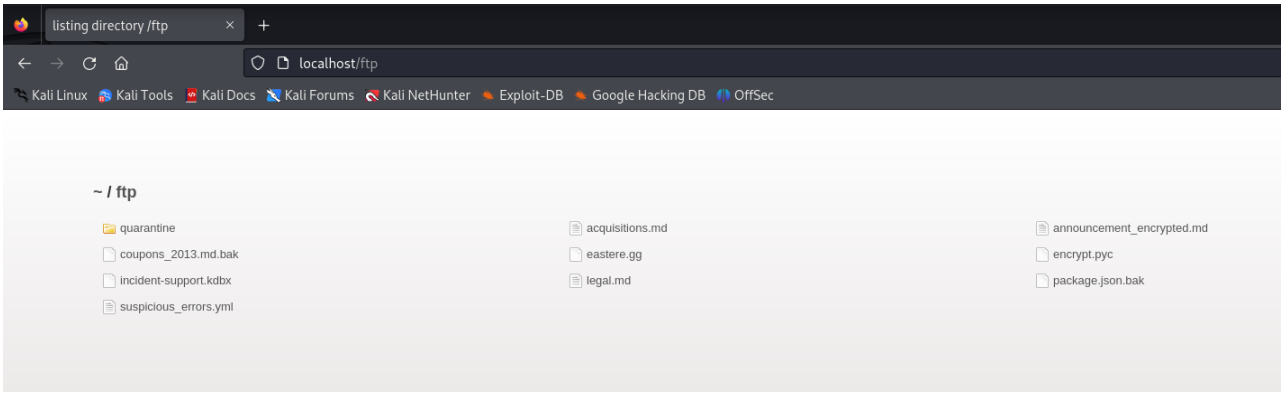

Figure 67: localhost/ftp results

The results remain the same, the WAF does not protect the application in this case.

3.2.2 - Test HTTP Methods

Regarding enumerating supported HTTP methods, the results are the same. The information that the methods GET, HEAD, PUT, PATCH, POST, DELETE are supported continues to appear when the commands are run.

As for testing for Access Control Bypass, the 5 tests were once again run. The GET, POST, and PUT methods still returned the HTML page, meaning the WAF does not work against these requests. As for the PATCH and DELETE requests, Figure 68 has the answers:

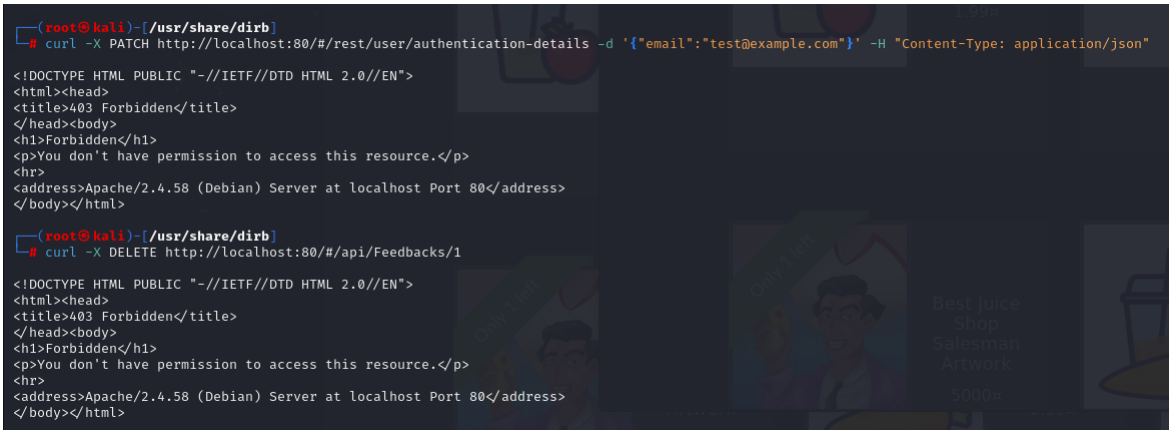

Figure 68: WAF - PATCH and DELETE requests

The WAF works for these, as the content returned as being of Forbidden nature.

Things continue to change when the subject is testing XST (Cross-Site Tracing) Vulnerabilities. Before, the application was vulnerable, but now it returns a 403 code Forbidden, meaning the WAF is protecting the OWASP Juice Shop against unauthorized gathering of information. Figure 69 counts as proof:

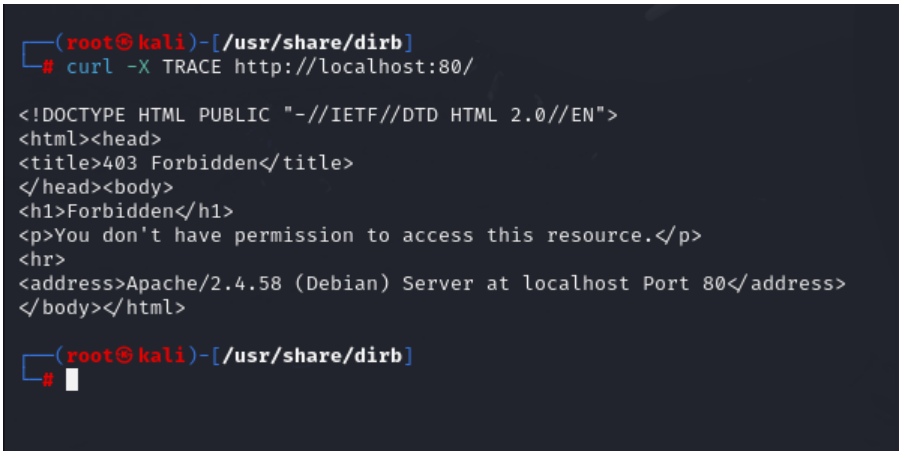

Figure 69: WAF - Checking if the TRACE method is enabled

Ending up with testing HTTP method overriding techniques, the same results apply. When the attempt to override is run, the response remains similar to that of a PUT response, therefore meaning that the server accepts and correctly processes the overridden method.

3.2.3 - Test HTTP Strict Transport Security

The same conclusions apply. Since the application is not running on HTTPS, there will be no automatic redirection from HTTP to HTTPS, and thus no HSTS enforcement.

3.3 - WSTG - Identity Management Testing

The issues identified in section 2.3 cannot be detected and solved with ModSecurity (with OWASP CRS), so no action is required here.

3.4 - WSTG - Authentication Testing

3.4.1 - Testing for Credentials Transported over an Encrypted Channel

Non-applicable. See 2.4.1.

3.4.2 - Testing for Default Credentials

Nothing changes here. The WAF could not repel the brute force attack made on the admin account, so probably if a fuzzing brute force attack was made on other accounts, the likelihood of success would be very high, as once again passwords only need to be 5 characters long, without any other restriction. See 2.4.2.

3.4.3 - Testing for Weak Lock Out Mechanism

The same as 2.4.3 applies here. As the fuzzing brute force attack was successful, this evidently means that there is absolutely no lockout mechanism, and the WAF does not provide protection to this vulnerability. An attacker can try to enter an account with the amount of tries they want. This represents a serious security problem, as brute force attacks can be performed quite easily.

3.4.4 - Testing for Bypassing Authentication Schema

Direct page request and parameter modification were already in place. As for SQL Injection, the picture is different. As one can see in 3.1.4, the WAF does its job and denies any type of fuzzing SQL Injection attack made by ZAP. And when trying to access the application with some of the queries discovered in 2.1.4, this happens, as exhibited by Figure 70:

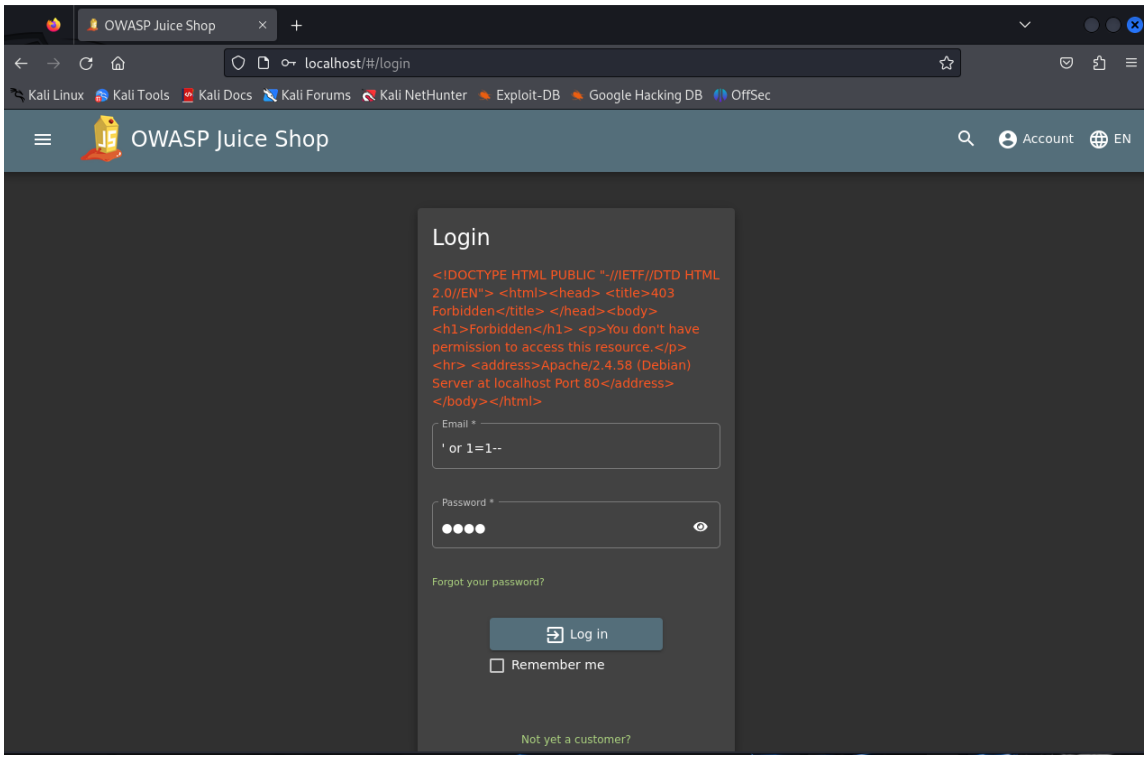

Figure 70: WAF - Trying to execute a SQL Injection in the application

The web application is now secure against attempts to bypass the authentication schema, thanks to the WAF.

3.4.5 - Testing for Vulnerable Remember Password

The same problem of 2.4.5 exists. The email becomes visible in the local storage when logging in with remember me activated, leading to potential unauthorized acquisition of information.

3.4.6 - Testing for Browser Cache Weaknesses

Testing browser history once more, what is discovered is that by navigating to pages with sensitive information of a user, logging out, and then trying to go Back is still allowed by the application, meaning the WAF has no effect here.

3.4.7 - Testing for Weak Password Policy

Everything written in 2.4.7 stands here.

3.4.8 - Testing for Weak Security Question Answer

The topic 2.4.8 has all the information relevant to this one.

3.4.9 - Testing for Weak Password Change or Reset Functionalities

Same as 2.4.9.

3.4.10 - Testing for Weaker Authentication in Alternative Channel

This topic is non-applicable to the OWASP Juice Shop, as it does not implement alternative authentication user channels.

3.5 - WSTG - Authorization Testing

3.5.1 - Testing Directory Traversal File Include

Since the application is not vulnerable to Path Traversal, there is nothing to test here, everything from 2.5.1 applies.

3.5.2 - Testing for Bypassing Authorization Schema

The same issues of 2.5.2 are applicable.

3.5.3 - Testing for Privilege Escalation

As already seen, with the WAF the application resists SQL Injection attempts, not letting the privilege escalation happen. On the other hand, it is still vulnerable to this by way of brute force attacks or intercepting the registration requests to change the role of the user.

3.5.4 - Testing for Insecure Direct Object References

Once again, the WAF protects against SQL Injection. But that does not mean that the parameter **q** of the URL **http://localhost:3000/rest/products/search?q=** magically becomes secure. It can still be modified to perform unauthorized searches of information in the application.

3.6 - WSTG - Session Management Testing

3.6.1 - Testing for Session Management Schema

As the token is still visible even with the presence of the WAF, nothing ends up changing, as the token is still available to be decoded, leaving information to be accessed by unauthorized parties.

3.6.2 - Testing for Cookies Attributes

The cookies are the same, and the tabs are the same. Therefore, visit 2.6.2 for more information.

3.6.3 - Testing for Session Fixation

Everything said in 2.6.3 applies here.

3.6.4 - Testing for Exposed Session Variables

Once more, 2.6.4 is the reference when it comes to this section.

3.6.5 - Testing for Cross Site Request Forgery

The CSRF attack was also performed on localhost:80. The results were the same as the results of 2.6.5, the CSRF attack was successful, therefore, the WAF does not protect against this threat.

3.6.6 - Testing for Logout Functionality

Same as 2.6.6.

3.6.7 - Testing Session Timeout

The content of 2.6.7 applies here.

3.6.8 - Testing for Session Puzzling

Visit 2.6.8 for information about this topic.

3.6.9 - Testing for Session Hijacking

Session Hijacking was also tested in localhost:80. The same results were achieved, injecting a session token of a user into the browser of another user proved to be possible. The WAF does not protect against session hijacking attacks.

3.7 - WSTG - Input Validation Testing

3.7.1 - Testing for Reflected Cross Site Scripting

Here, the XSS attack is also successful. The WAF does not give protection against XSS exploits. Figure 71 proves this:

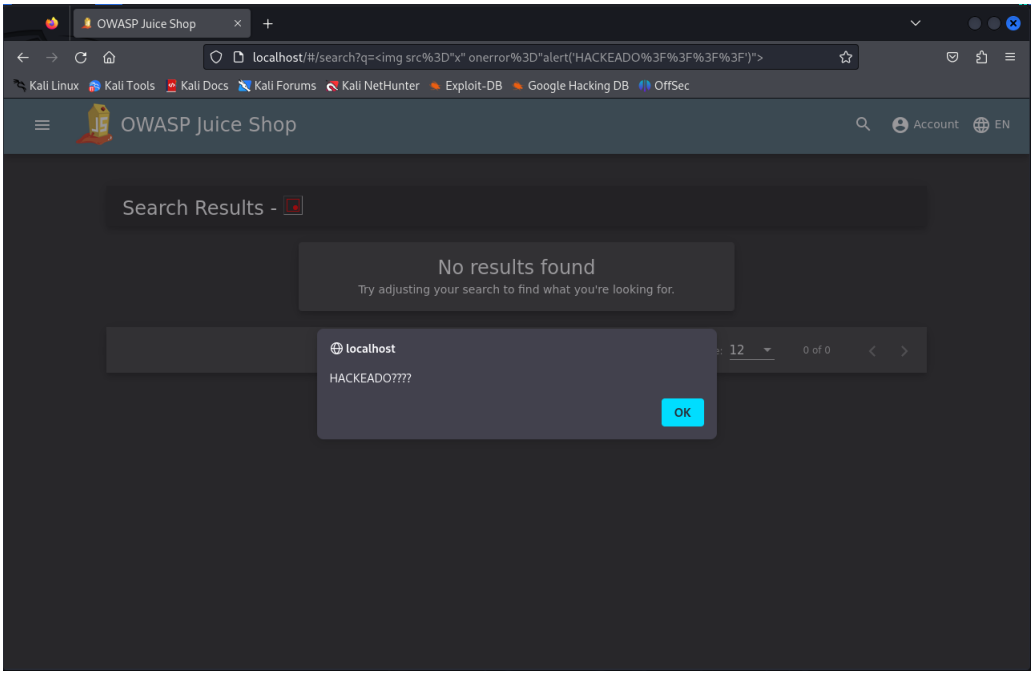

Figure 71: WAF - XSS attack

3.7.2 - Testing for Stored Cross Site Scripting

Unlike 2.7.2, here some other conclusions are reached. When the Stored XSS attack attempt is performed, the following appears, exhibited by Figure 72:

Figure 72: WAF - Stored XSS attack

Furthermore, nothing is stored on the administration page. This means that the WAF successfully repels Stored XSS attacks, a huge accomplishment.

3.7.3 - Testing for HTTP Verb Tampering

Non-Applicable. See 3.2.2 for more information.

3.7.4 - Testing for HTTP Parameter Pollution

Here, the logic is the same as 2.7.4. Just one note, those URLs mentioned, if targeted for SQL Injection attacks, are going to be secure because of the WAF, as it was already proven numerous times that the WAF blocks SQL Injections.

3.7.5 - Testing for SQL Injection

See 2.7.5.

3.7.5.3 - Testing for SQL Server

Once more, the WAF protects the application against SQL Injections.

3.8 - WSTG - Testing for Error Handling

3.8.1 - Testing for Improper Error Handling

A similar process to 2.8.1, the improper error handling still happens with the WAF in place.

3.8.2 - Testing for Stack Traces

Non-applicable. Visit the previous topic for information, 3.8.1.

4 - Keycloak as identity provider

The final phase of this assignment corresponds to configuring Keycloak to enable SSO on an application.

4.1 - Brute Force Attack on the admin console of Keycloak

The first step required is to perform a brute force attack on the admin console. The tools Hydra and OWASP ZAP were selected to assist in this procedure. Regarding Hydra, a list of the 100,000 most common passwords was gathered, and then the following command was run:

hydra -l admin -P /home/leonardo/Downloads/10-million-password-list-top-10000.txt localhost http-form-post "/realms/master/protocol/openid-connect/auth:username=admin&password= PASS^:F=Invalid user credentials"

Figure 73 shows the result of this attack:

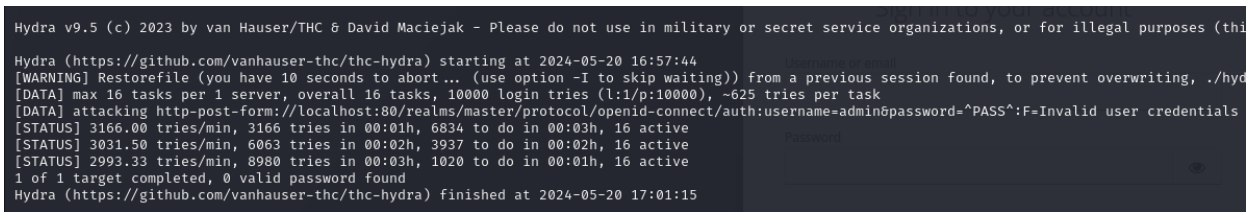

Figure 73: Brute force attack on the admin console (Hydra)

No valid passwords were found. Maybe a stronger wordlist should have been used, but the time it would take to execute the process would not have been feasible.

As for the use of ZAP, a brute force fuzz attack was performed, similar to what was made in section 2.1.4. The results can be seen in Figure 74:

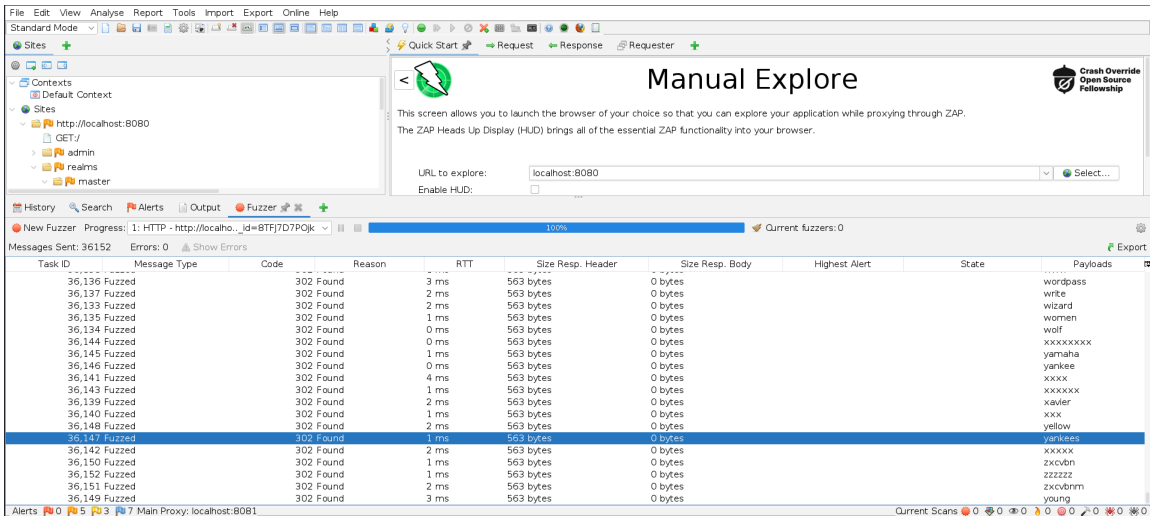

Figure 74: Brute force attack on the admin console (ZAP)

The findings provided by ZAP do not make a lot of sense, as all the payloads return the code 302 Found, indicating that a specific URL has been moved temporarily to a new location. Therefore, not much can be concluded here.

4.2 - Brute Force Attack on users accounts

Next, a brute force attack is going to be made targeting user accounts. This attack is similar to the previous one, differing in the fact that now a dictionary of usernames is used, to simulate the possible user accounts. Once again, Hydra is used, and the command below was executed:

hydra -L /home/leonardo/Downloads/usernames.txt -P /home/leonardo/Downloads/10-million-password-list-top-10000.txt localhost http-form-post "/realms/myrealm/protocol/openid-connect/auth:username=^USER^&password=^PASS^:F=Invalid user credentials"

Conclusions can be taken from Figure 75:

```
Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2024-05-20 17:46:19
[WARNING] Restorefile (you have 10 seconds to abort... (use option -I to skip waiting)) from a previous session found, to prevent overwriting, ./hydra
[DATA] max 16 tasks per 1 server, overall 16 tasks, 50005 login tries (l:5/p:10001), ~3126 tries per task
[DATA] attacking http-post-form://localhost:80/realms/myrealm/protocol/openid-connect/auth:username=^USER^&password=^PASS^:F=Invalid user credentials
[STATUS] 3369.00 tries/min, 3369 tries in 00:01h, 46636 to do in 00:14h, 16 active
[STATUS] 3324.33 tries/min, 9973 tries in 00:03h, 40032 to do in 00:13h, 16 active
[STATUS] 3334.00 tries/min, 23338 tries in 00:07h, 26667 to do in 00:08h, 16 active
[STATUS] 3374.42 tries/min, 40493 tries in 00:12h, 9512 to do in 00:03h, 16 active
1 of 1 target completed, 0 valid password found
[WARNING] Writing restore file because 3 final worker threads did not complete until end.
[ERROR] 3 targets did not resolve or could not be connected
[ERROR] 0 target did not complete
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2024-05-20 18:00:40
```

Figure 75: Brute force attack on the user accounts

Once again, no valid passwords were found. But despite failing to obtain credentials in both of these tests, Keycloak is going to be hardened either way.

4.3 - Hardening the security of Keycloak

Four different processes will be performed in order to strengthen the security of Keycloak. To begin with, it is required that the login in the admin console is protected. Figure 76 displays the configurations made to reach that goal:

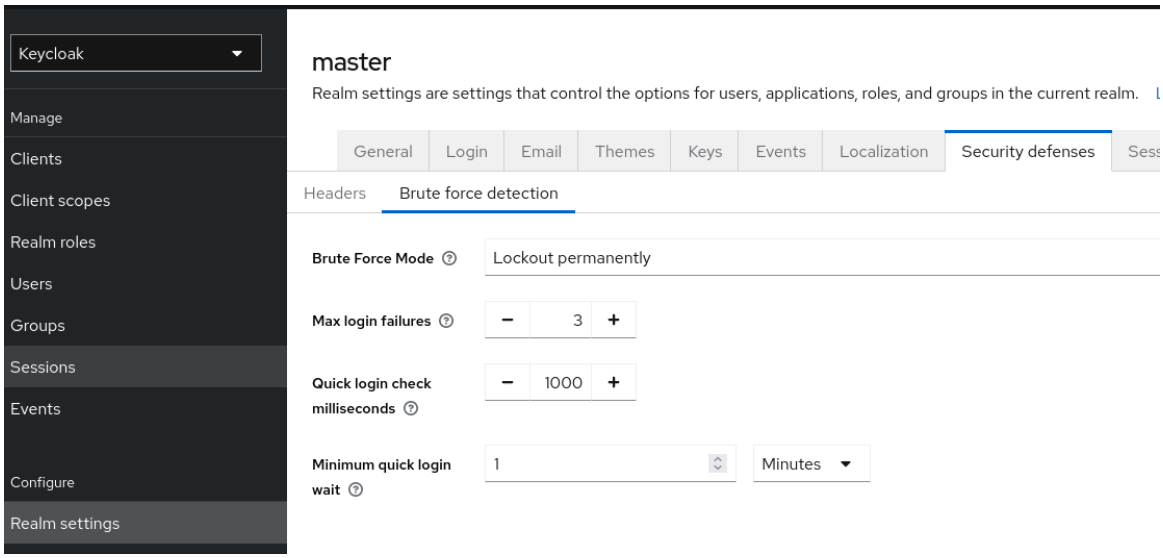

Figure 76: Protecting the login in the admin console

Looking at these decisions, to sum up, if a brute force entry attempt is detected, the user will be locked out permanently. Then, only 3 login failures are allowed at max. These contribute to making the login aspect of the admin console safer, mainly against adversaries that might use brute force attacks.

Moving on to the second measure to harden Keycloak, the aim is to improve the security of the SSO session, by reducing the lifetime session. Figure 77 illustrates the decisions taken:

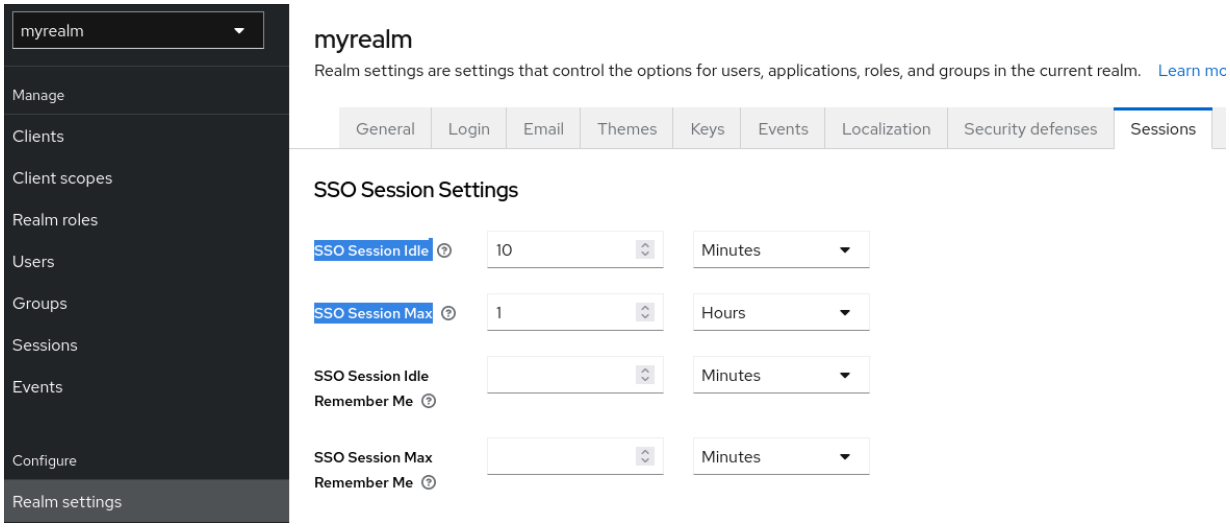

Figure 77: Hardening the security of SSO sessions

As one can observe, the SSO Session Idle attribute was set to 10 minutes, while the SSO Session Max attribute was configured to 1 hour. This means that if a user does not interact with the application within 10 minutes, he will be automatically disconnected, and the maximum time a session can have is 1 hour.

Next up is hardening the security of Keycloak to protect against brute force attacks. An idea can be to implement a WAF capable of dealing with these attacks, blocking them, unlike the one used in the previous phase of this assignment. In the end, some more brute-force-oriented measures were put in place, a procedure shown in Figure 78:

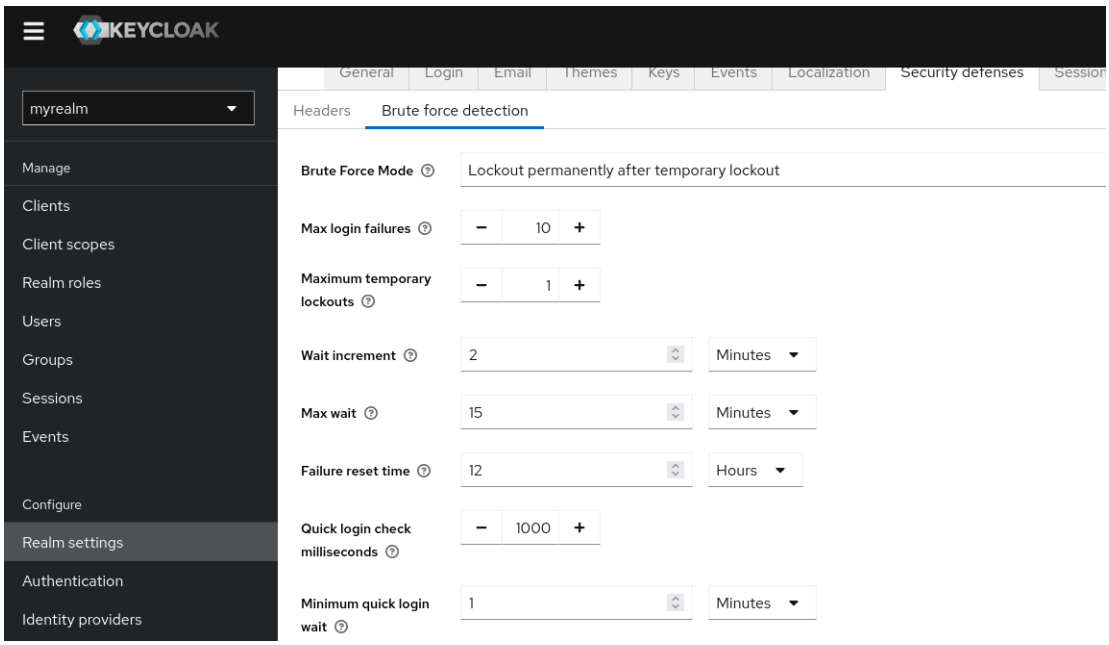

Figure 78: Hardening the security of Keycloak to protect against brute force attacks

Finally, the security of the client in Keycloak must also be dealt with. Figure 79 shows password policies implemented, while Figure 80 treats user events settings:

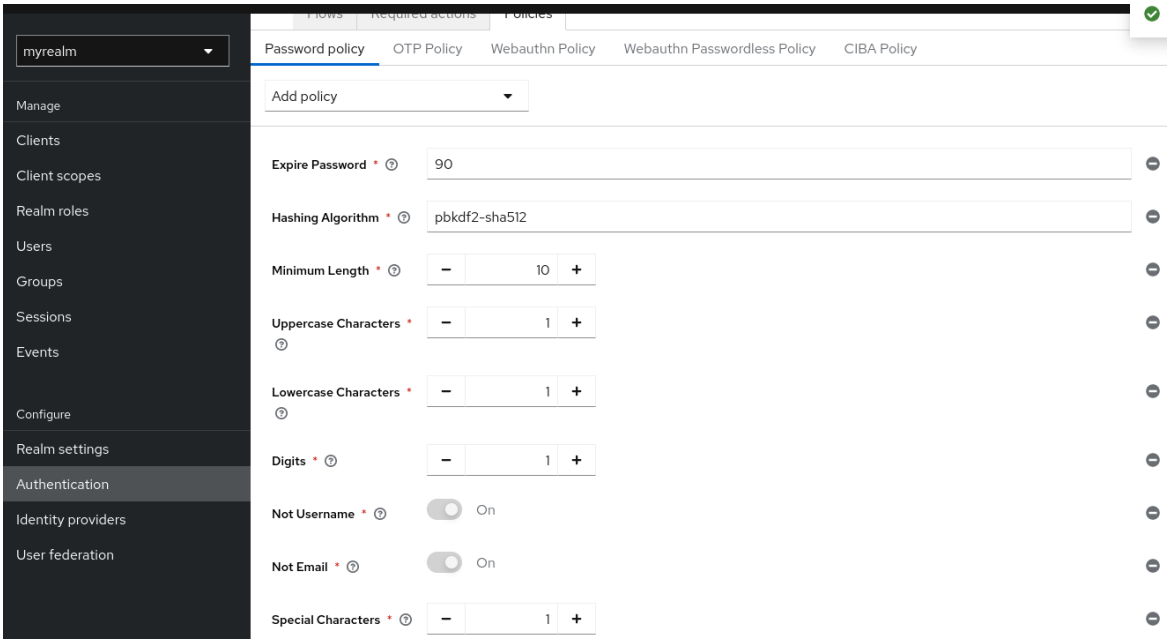

Figure 79: Hardening the security of the client - Password Policies

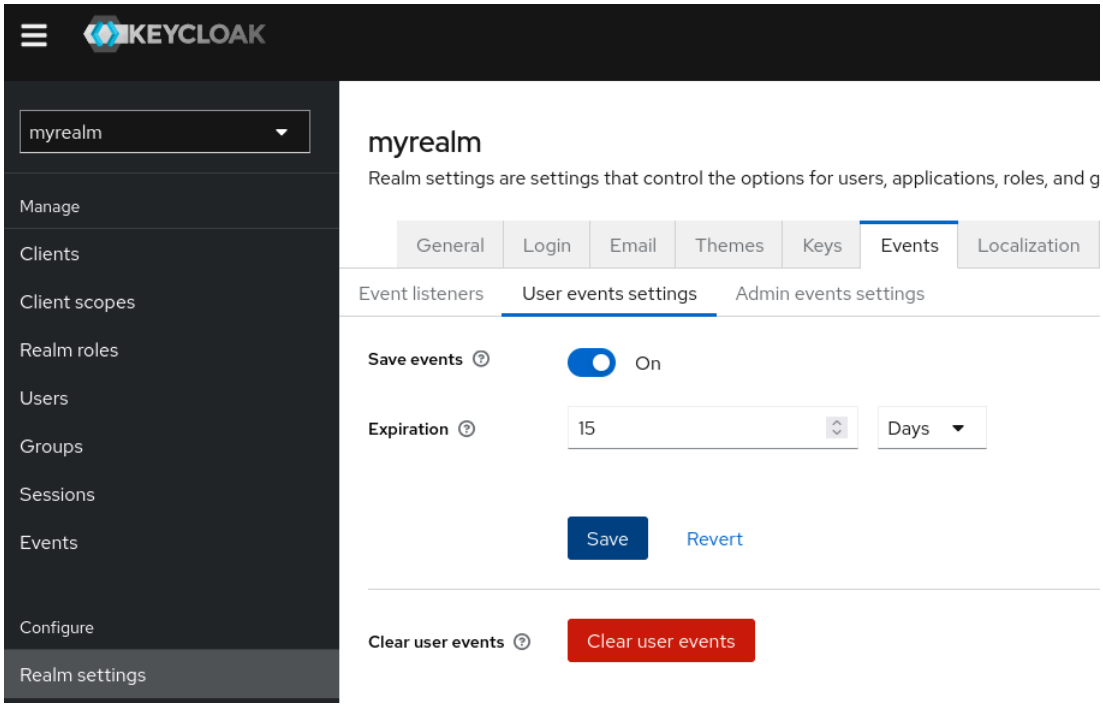

Figure 80: Hardening the security of the client - User Events Settings

Conclusion

This assignment focused mainly on one topic: Web Application Security Testing. Additionally, Keycloak was also a concept of some relevance. A lot of things were learned, and experience was acquired mainly with the tool OWASP ZAP. Technical skills like researching and problem-solving are also considered to have been improved by this assignment. Concepts like WAF, HTTP, Identity Management, Authentication, Authorization, Session Management, Input Validation, and Error Handling were also greatly worked on.

Some tests might not have been well executed or might have been left incomplete or without a proper result, but all in all, it is considered that the main requirements were reached. It must be mentioned that this WAF was not very successful, as it still left a significant number of vulnerabilities open to adversaries. Future work in this area can include this type of testing with a whole different bunch of web applications, that can be safer or not than the OWASP Juice Shop, and the different WAFs, that can be stronger or not than the ModSecurity OWASP WAF.