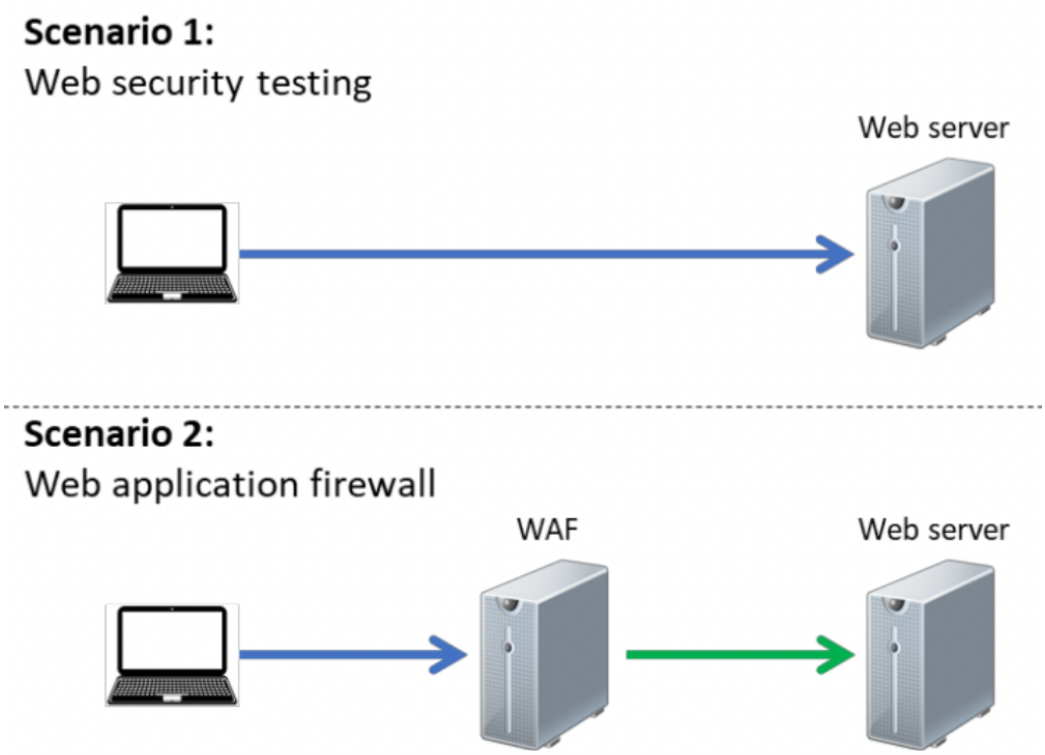
Web Application Security Exploration

Practical Assignment 3

Bruno Faria e Dylan Perdigão
2018295474, 2018233092
{brunofaria, dgp}@student.dei.uc.pt
Department of Informatics Engineering, University of Coimbra

May 2022



Contents

1 Introduction

The objective of this assignment is to explore the security of a Juice Shop [4] web application and to implement a firewall to secure this web application against application-layer attacks. The first phase is dedicated to exploring the Juice Shop security with *Web Security Testing Guide* [7]. The second phase aims to monitor, filter, and block *HTTP* traffic to the Juice Shop through the implementation of a *ModSecurity* web application firewall.

2 Structure

A Docker container was created with the Juice Shop Web Application for the first phase. The following code was executed in order to push the image and run the server:

```
docker pull bkimminich/juice-shop
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

The container runs on port 3000 and is accessible via `http://localhost:3000` or `http://192.168.93.1:3000`.

We installed OWASP ZAP to make some of our penetrations tests. To complement it, we also installed two add-ons, *Directory List v2.3* and *Advanced SQLInjection Scanner*.

For the second phase, we created a *docker-compose.yml* file in order to build the Docker containers (see listing 1).

```yaml
version: '3'
name: 'WAF'
services:
  juiceshop:
    image: bkimminich/juice-shop:latest
    ports:
      - 3000:3000
    container_name: juiceshop
  modsecurity:
    image: owasp/modsecurity-crs:apache
    ports:
      - 80:80
    container_name: modsecurity
    environment:
      - PROXY=1
      - UPSTREAM=http://192.168.1.129:3000
      - PARANOIA=3
      - BACKEND=http://192.168.1.129:3000
```

Listing 1: docker-compose.yml

Then, we installed the *modsecurity-crs* in docker and run it via the following commands:

```
docker pull owasp/modsecurity-crs:apache
docker compose up
```

This should result in the containers of the figure 1.

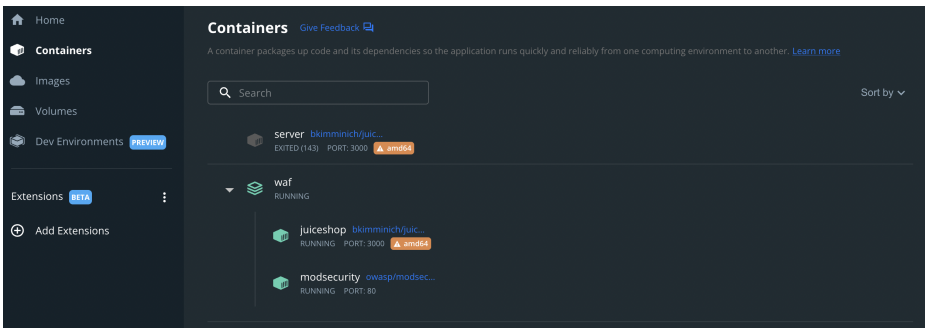


Figure 1: Docker containers

Note that the container must run into a real, local network. This time, we put the container running into the address `http://192.168.1.129`. The Juice Shop is still on port 3000, but now we access it via the proxy on port 80. We defined a *PARANOIA* level of 3.

3 Web application security testing

3.1 Information Gathering

A first approach is to find open ports on the Docker container containing the Juice Shop website. Using the *nmap* tool, more particularly on the following code:

```
nmap -Pn -sT -sV -p0-65535 192.168.93.1
nmap -Pn -sT -sV -p0-65535 192.168.1.129
```

Results of this command are showed on table 1

PORT	STATE	SERVICE
3000/tcp	open	ppp
5000/tcp	open	upnp
7000/tcp	open	afs3-fileserver
8080/tcp	open	http-proxy

Table 1: Opened Ports

With *OWASP ZAP* software, it is trivial to get access to website files. More particulary, we found a confidential document making a *GET* request to the url `http://localhost:3000/ftp/acquisitions.md` as we can see on figure 2. Using the same procedure, it is possible to find an Easter Egg [1] file at `http://localhost:3000/ftp/eastere.gg`

```
# Planned Acquisitions

> This document is confidential! Do not distribute!

Our company plans to acquire several competitors within the next year.
This will have a significant stock market impact as we will elaborate in
detail in the following paragraph:

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam
voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet
clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit
amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam
nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat,
sed diam voluptua. At vero eos et accusam et justo duo dolores et ea
```

Figure 2: Confidential file

At the beginning of the file, we have found a comment saying the name *Bjoern Kimminich* which is of one of the web application contributors:

[1] Easter Egg: `https://en.wikipedia.org/wiki/Easter_egg_(media)`

```
<!--
  ~ Copyright (c) 2014-2022 Bjoern Kimminich & the OWASP Juice Shop contributors.
  ~ SPDX-License-Identifier: MIT
-->
```

Another point we want to talk is about the application showing the emails of product reviews as we can see on figure 3. The following emails were gathered:

- `bender@juice-sh.op`
- `uvogin@juice-sh.op`
- `mc.safesearch@juice-sh.op`
- `morty@juice-sh.op`
- `admin@juice-sh.op`
- `stan@juice-sh.op`
- `jim@juice-sh.op`
- `accountant@juice-sh.op`
- `bjoern@owasp.org`

These emails can be exploited in a future section for login with SQL injection, more particularly, the `admin@juice-sh.op`, which is an admin account, or `bjoern@owasp.org`, which is the account of the web app creator.

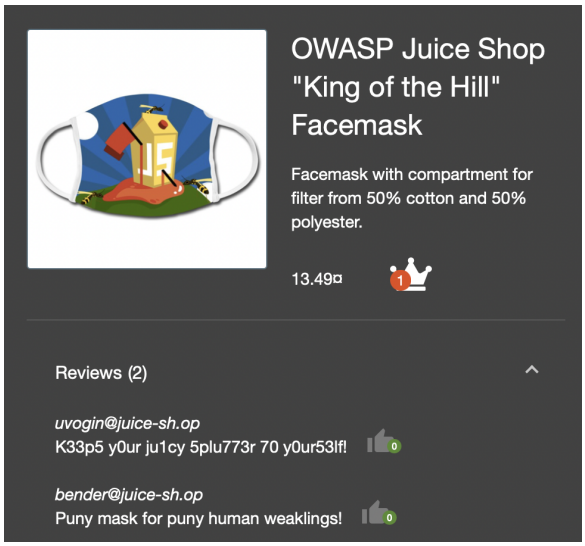


Figure 3: E-mails in a product review

3.2 Configuration and Deployment Management Testing

While exploring the HTTP requests, we found that writing reviews on behalf of someone else were possible. This was obtainable by taking a look at the request sent while writing a review and observe, as we can see in figure 4, that it passes the author as an argument.

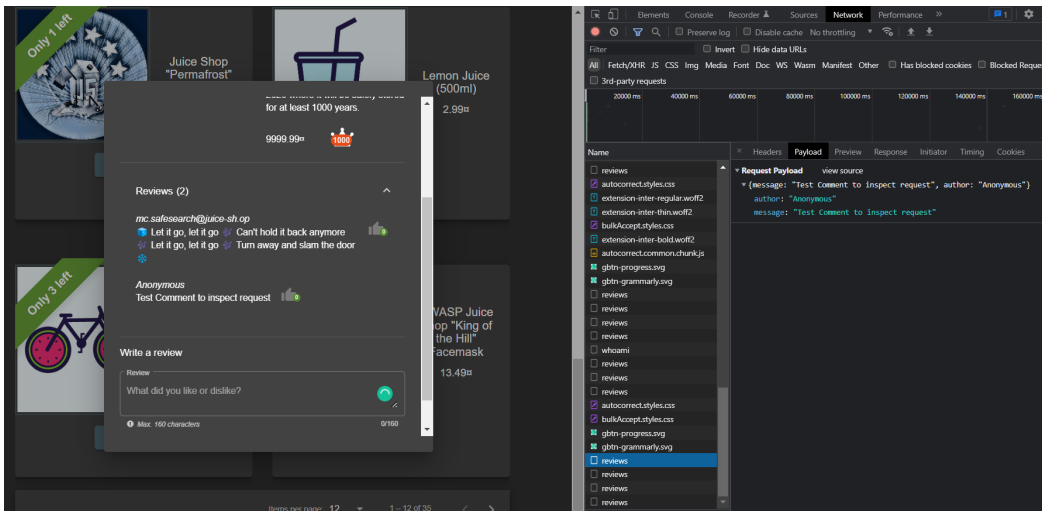


Figure 4: Request made while commenting

Then, we proceeded to try to replicate this *PUT* request by changing the author parameter to *"mc.safesearch@juice-sh.op"*. This request and response can be found in figure 5.

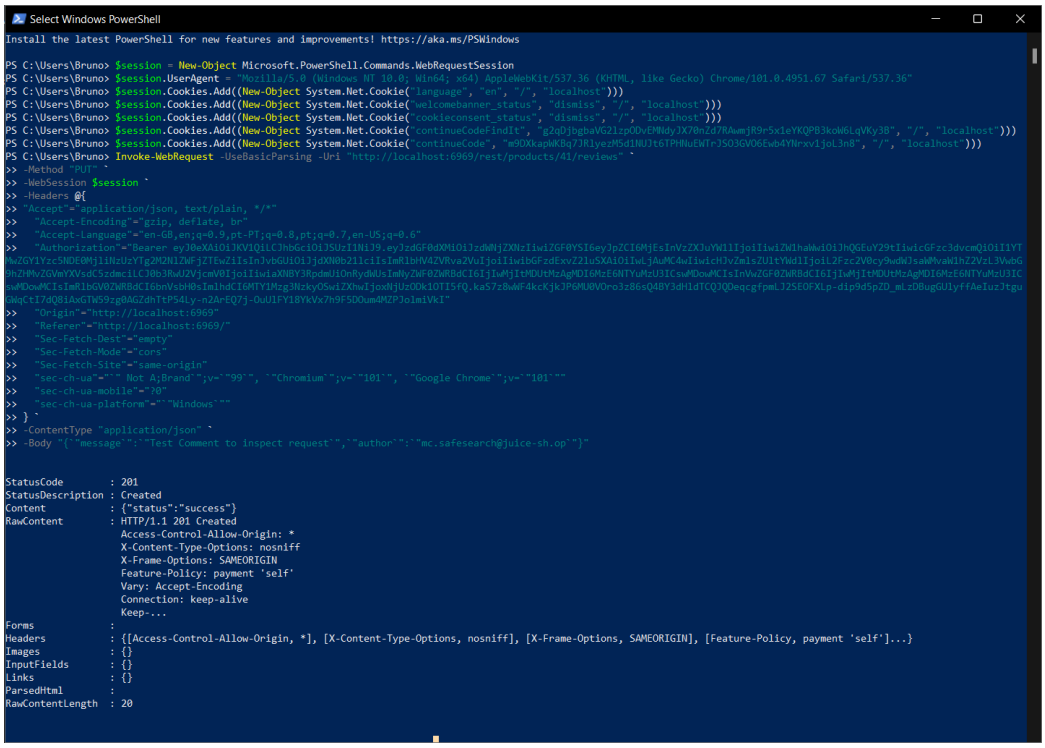


Figure 5: Request made to create comment in other's name

As we can see, the response obtained was "success", so it accepted the comment with someone else's email, as evident in figure 6

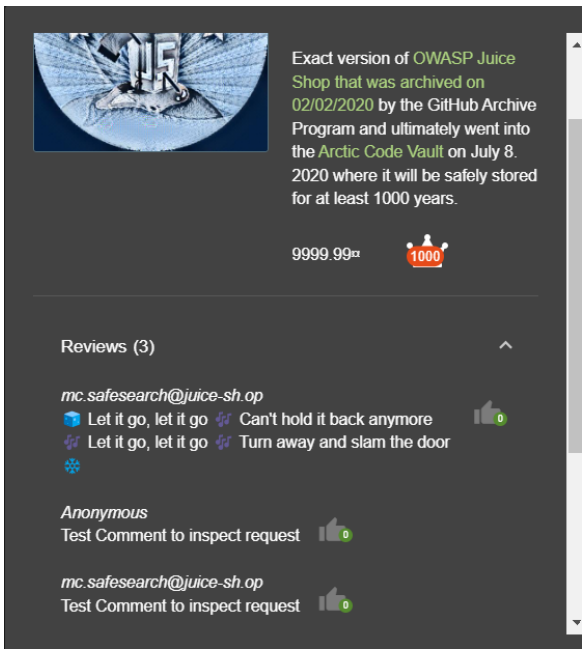


Figure 6: Comments after our *PUT* request

3.3 Identity Management Testing

In this section, we searched for analyzing form errors. We retried to create the same account using the login form and creating an account with random credentials. One element that takes our attention is the error message that indicates the account with this email already exists (shown in figure 7).

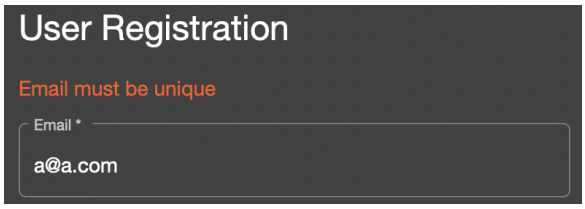


Figure 7: Login error message for existing email

3.4 Authentication Testing

As said on the *WSTG*, we can look for the default admin-user password. We have made a little search on the browser with the string "Juice Shop default password", and we obtained the result of figure 8. The password is *admin123* and the email is `admin@juice-sh.op`.

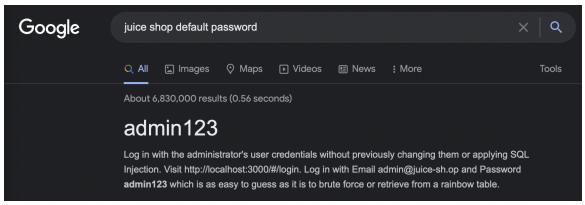
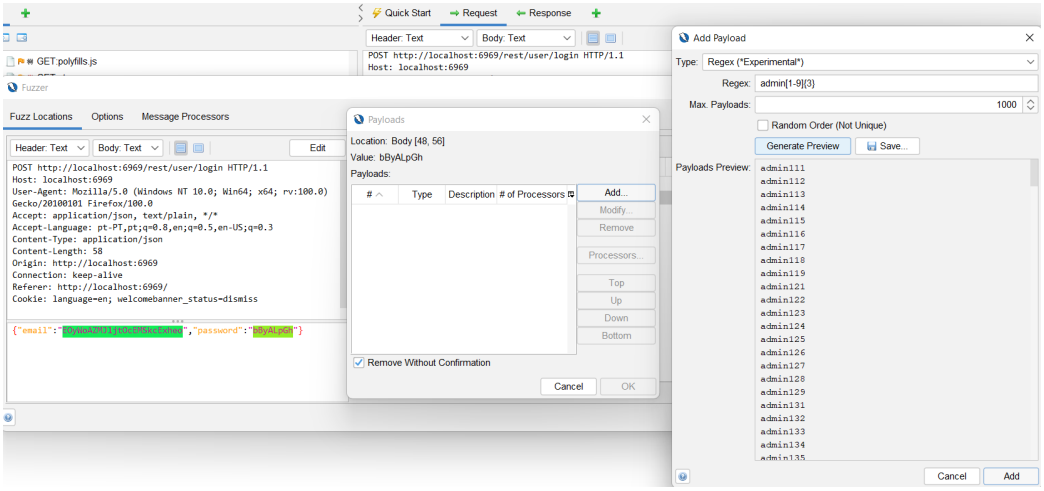


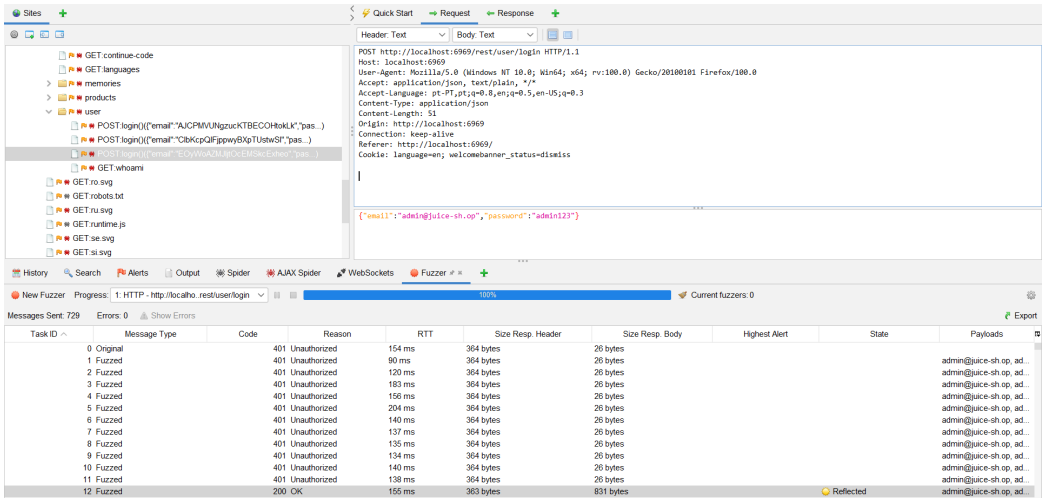
Figure 8: Default password for the Juice Shop administrator account

Having the credentials, we tried to iterate 3, 4, and 5 times to log in with the wrong password, and we noticed the Shop is vulnerable to brute-force attacks because there is no lockout mechanism

or *CAPTCHA* to avoid it. Knowing this, we tried to brute-force the password using OWASP ZAP Fuzz attack as shown in figure 9. This was only for demonstration since we already knew the admin's password.



(a) Fuzzer configuration



(b) Fuzzer results

Figure 9: The Fuzzer screens for brute-forcing admins's password

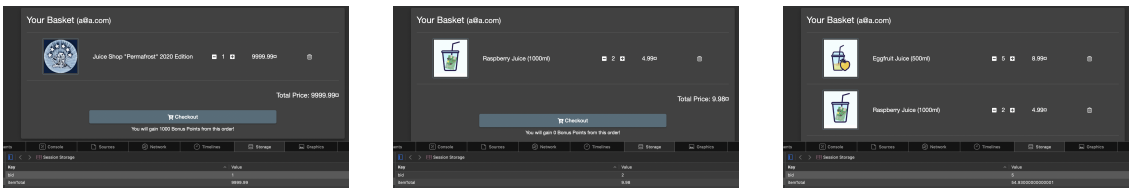
During the account creation, we also noticed that it is possible to use weak passwords, like "*12345*", "*password*", or "*qwerty*" which is a bad point for account security which can be easily brute-forced using the previous technique.

3.5 Authorization Testing

Regarding authorization testing, we also found some vulnerabilities. One example is mentioned in section 3.1, were we easily found a confidential file as shown in figure 2.

3.6 Session Management Testing

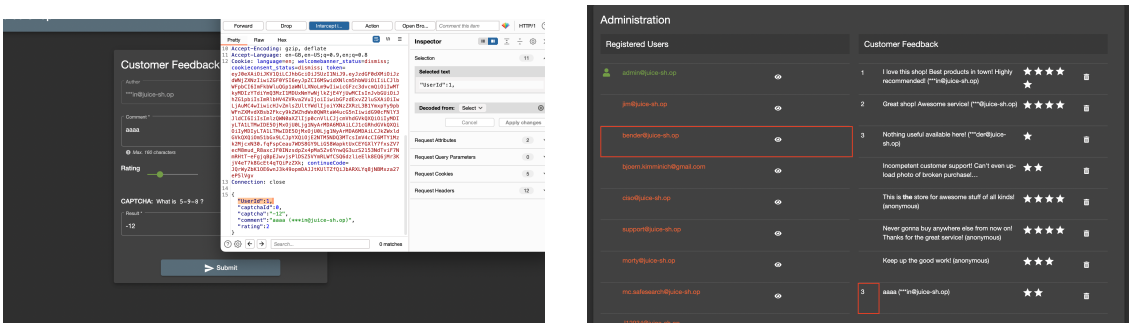
User's baskets are stored by session variables, which the browser inspector element can modify. For example, we have logged in with the `a@a.com` account, which has an empty basket. Changing the value of the session variable *bid* seems to change the ID of the basket account, as we can see in figure 10.



(a) $bid = 1$ (b) $bid = 2$ (c) $bid = 5$

Figure 10: Some user's baskets accessed by the user `a@a.com`

It is also possible to send feedback to another account. For that, we used *Burp Suite* software [2] to change the ID of the account sending the feedback. The `a@a.com` account is used. The request is intercepted and we can edit the "*UserId*" value (figure 11a). To see the result, we go to the administration panel at `http://localhost:3000/#/administration` where the feedback what sent by the `bender@juice-sh.op` account (user n°3).



(a) Editing the value with Burp Suite (b) Administration panel with new feedback

Figure 11: Changing the user's feedback with `a@a.com` account

3.7 Input Validation Testing

Another vulnerability found is that it is possible to inject SQL [5] into the login form. With that, we have access to every account, only putting the string "$'--$" after a user email. It bypasses every random password submitted in the form - for example, the admin's password as exemplified in figure 12.

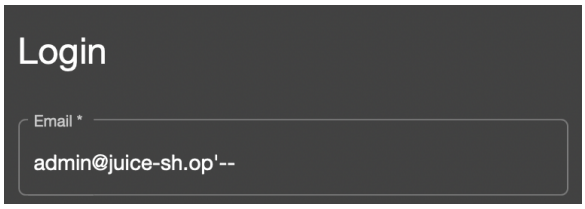


Figure 12: SQL injection attack to bypass admin's password

A second vulnerability regarding the input validation is observed using a technique called Null Byte Injection [6]. However, since we are testing in the web application, this is encoded as *%2500*. Using this technique, we can access the remaining files from `http://localhost:3000/ftp`, which includes some backups and the Easter Egg file.

3.8 Testing for Error Handling

While trying the SQL Injection mentioned in section 3.7, we first tried "$'$" as email. This raised an Internal Server Error (status 500) as seen in figure 13

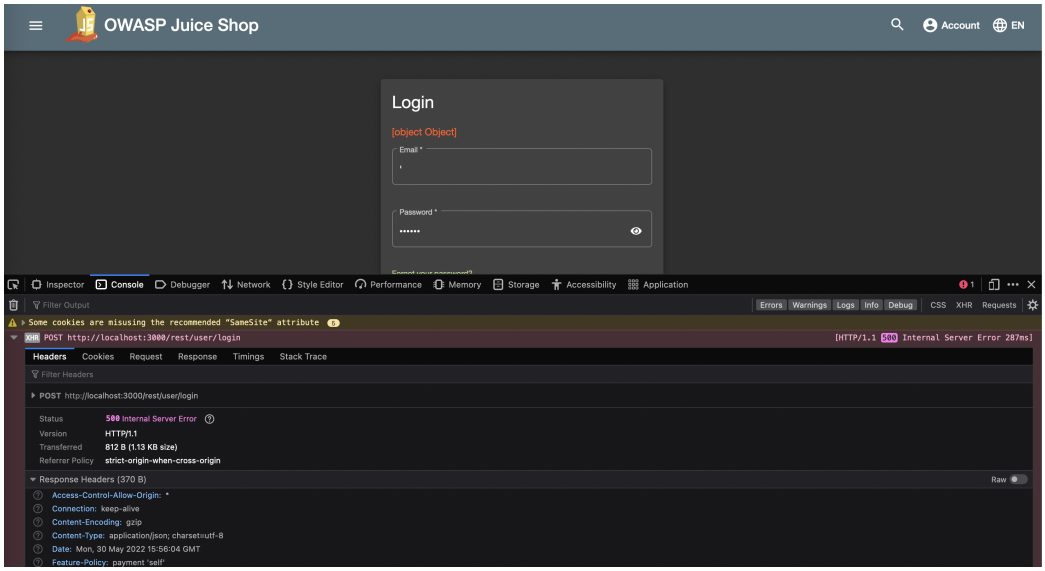


Figure 13: Internal Server Error 500

Performing an active scan to explore the authentication area, we can find the same "'" problem, as we can verify in figure 14, among many others that throw Internal Server Error.

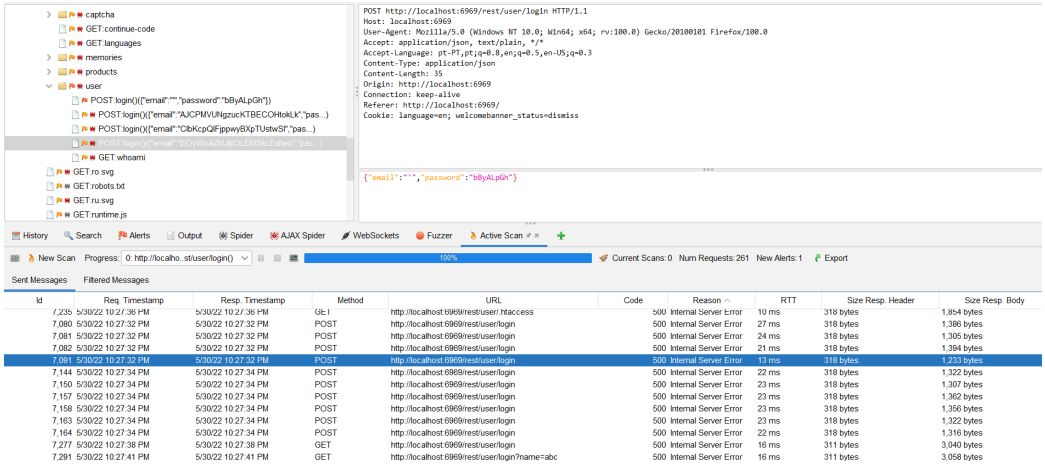


Figure 14: Internal Server Error 500 using OWASP ZAP active scan in login area

This error type is way more evident using OWASP ZAP's automated scan, where we can find many cases where we get Internal Server Error as evident in figure 15.

Id	Req. Timestamp	Resp. Timestamp	Method	URL	Code	Reason	RTT	Size Resp. Header	Size Resp. Body
6,738	5/30/22 7:42:23 PM	5/30/22 7:42:23 PM	GET	http://localhost:6969/ftp/quarantine/juicy_malware_macos_64.url/	500	Internal Server Error	18 ms	311 bytes	1,072 bytes
6,739	5/30/22 7:42:23 PM	5/30/22 7:42:23 PM	GET	http://localhost:6969/ftp/quarantine/juicy_malware_windows_64.exe.url/	500	Internal Server Error	18 ms	311 bytes	1,084 bytes
6,740	5/30/22 7:42:23 PM	5/30/22 7:42:23 PM	GET	http://localhost:6969/ftp/suspicious_errors.yml/	500	Internal Server Error	18 ms	311 bytes	1,040 bytes
6,744	5/30/22 7:42:24 PM	5/30/22 7:42:24 PM	GET	http://localhost:6969/rest/	500	Internal Server Error	20 ms	311 bytes	3,020 bytes
6,745	5/30/22 7:42:24 PM	5/30/22 7:42:24 PM	GET	http://localhost:6969/rest/admin/	500	Internal Server Error	19 ms	311 bytes	3,032 bytes
6,749	5/30/22 7:42:25 PM	5/30/22 7:42:25 PM	GET	http://localhost:6969/rest/products/	500	Internal Server Error	19 ms	311 bytes	3,038 bytes
6,750	5/30/22 7:42:25 PM	5/30/22 7:42:25 PM	GET	http://localhost:6969/rest/products/6/	500	Internal Server Error	20 ms	311 bytes	3,042 bytes
6,753	5/30/22 7:42:26 PM	5/30/22 7:42:26 PM	GET	http://localhost:6969/rest/user/	500	Internal Server Error	19 ms	311 bytes	3,030 bytes
6,754	5/30/22 7:42:26 PM	5/30/22 7:42:26 PM	GET	http://localhost:6969/rest/user/login/	500	Internal Server Error	19 ms	311 bytes	3,042 bytes
6,928	5/30/22 7:43:08 PM	5/30/22 7:43:08 PM	POST	http://localhost:6969/rest/user/login	500	Internal Server Error	21 ms	318 bytes	1,196 bytes
6,931	5/30/22 7:43:08 PM	5/30/22 7:43:08 PM	POST	http://localhost:6969/rest/user/login	500	Internal Server Error	25 ms	318 bytes	1,196 bytes
6,934	5/30/22 7:43:08 PM	5/30/22 7:43:08 PM	POST	http://localhost:6969/rest/user/login	500	Internal Server Error	23 ms	318 bytes	1,196 bytes
6,936	5/30/22 7:43:08 PM	5/30/22 7:43:08 PM	POST	http://localhost:6969/rest/user/login	500	Internal Server Error	25 ms	318 bytes	1,196 bytes
6,938	5/30/22 7:43:08 PM	5/30/22 7:43:08 PM	POST	http://localhost:6969/rest/user/login	500	Internal Server Error	19 ms	318 bytes	1,196 bytes
6,941	5/30/22 7:43:08 PM	5/30/22 7:43:08 PM	POST	http://localhost:6969/rest/user/login	500	Internal Server Error	26 ms	318 bytes	1,197 bytes
6,944	5/30/22 7:43:08 PM	5/30/22 7:43:08 PM	POST	http://localhost:6969/rest/user/login	500	Internal Server Error	20 ms	318 bytes	1,197 bytes
7,077	5/30/22 7:43:29 PM	5/30/22 7:43:29 PM	POST	http://localhost:6969/rest/user/login	500	Internal Server Error	19 ms	318 bytes	1,192 bytes
7,080	5/30/22 7:43:29 PM	5/30/22 7:43:29 PM	POST	http://localhost:6969/rest/user/login	500	Internal Server Error	26 ms	318 bytes	1,192 bytes
7,087	5/30/22 7:43:30 PM	5/30/22 7:43:30 PM	GET	http://localhost:6969/api/Challenges/?name=%00	500	Internal Server Error	25 ms	380 bytes	81 bytes
7,134	5/30/22 7:43:48 PM	5/30/22 7:43:48 PM	GET	http://localhost:6969/rest/products/search?q=%00	500	Internal Server Error	24 ms	311 bytes	986 bytes
7,164	5/30/22 7:43:49 PM	5/30/22 7:43:49 PM	GET	http://localhost:6969/rest/user/.htaccess	500	Internal Server Error	4 ms	318 bytes	1,854 bytes
7,168	5/30/22 7:43:49 PM	5/30/22 7:43:49 PM	GET	http://localhost:6969/api/.htaccess	500	Internal Server Error	19 ms	311 bytes	3,038 bytes
7,170	5/30/22 7:43:50 PM	5/30/22 7:43:50 PM	GET	http://localhost:6969/api/Quantitys/.htaccess	500	Internal Server Error	19 ms	311 bytes	3,095 bytes
7,258	5/30/22 7:43:53 PM	5/30/22 7:43:53 PM	GET	http://localhost:6969/rest/admin/.htaccess	500	Internal Server Error	18 ms	311 bytes	3,050 bytes
7,259	5/30/22 7:43:53 PM	5/30/22 7:43:53 PM	GET	http://localhost:6969/rest/.htaccess	500	Internal Server Error	27 ms	311 bytes	3,038 bytes
7,260	5/30/22 7:43:54 PM	5/30/22 7:43:54 PM	GET	http://localhost:6969/rest/products/6/.htaccess	500	Internal Server Error	16 ms	311 bytes	3,060 bytes
7,261	5/30/22 7:43:54 PM	5/30/22 7:43:54 PM	GET	http://localhost:6969/rest/products/.htaccess	500	Internal Server Error	24 ms	311 bytes	3,056 bytes

Figure 15: Internal Server Error 500 using OWASP ZAP automated scan

3.9 Testing for Weak Cryptography

The first sign of Weak Cryptography is verified in section 3.7 when we tried SQL Injection. Then, better analyzing the request made to the API, we verify that the password is sent without any encryption, as we can see in figure 16.

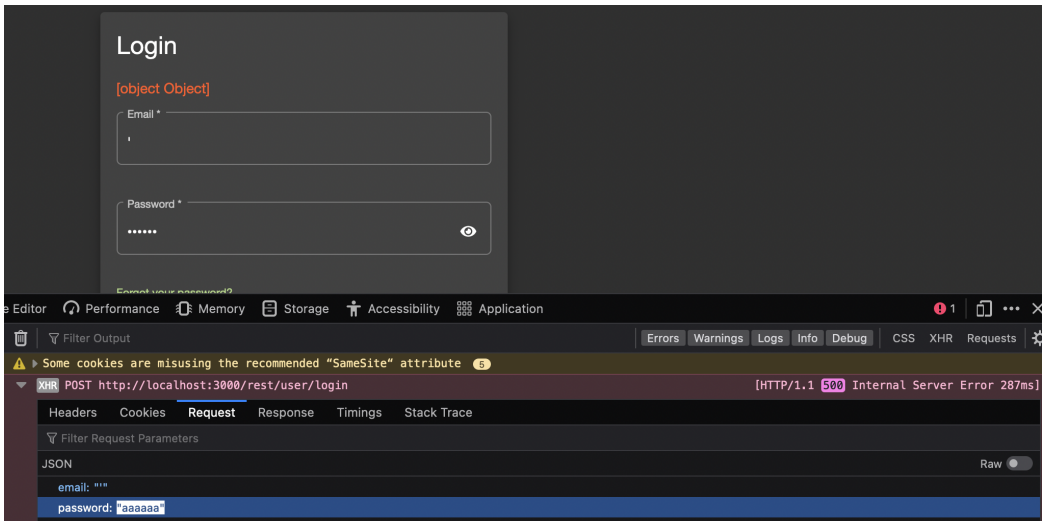
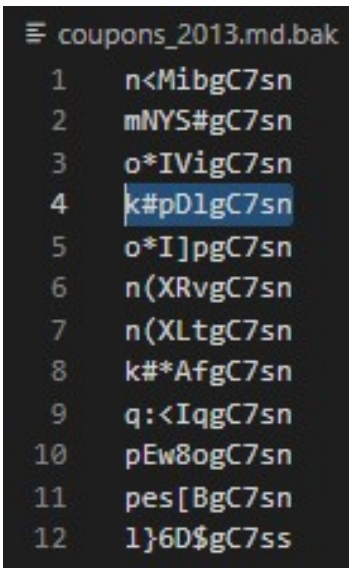
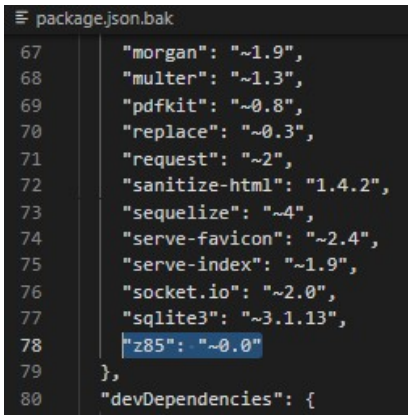


Figure 16: Password not being encrypted while leaving front-end

Another form of weak Cryptography can be detected in trying to forge a coupon. As we mentioned in section 3.7 we found some backup files (figure 17).



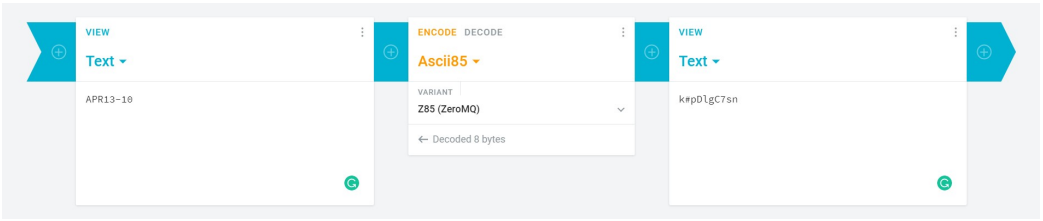
(a) cupouns_2013.md.bak



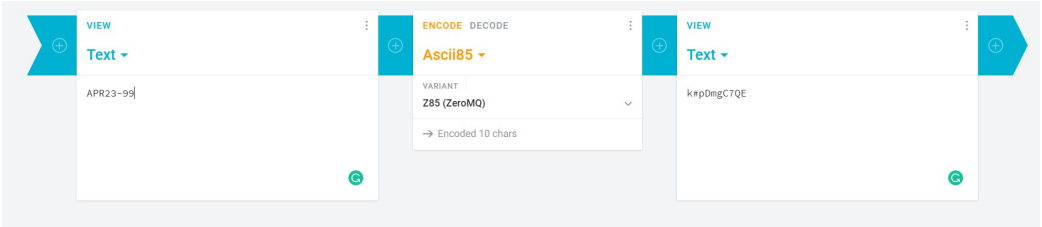
(b) packages.json.bak

Figure 17: Backup files found

As we can see in figure 17a, the coupons have some similarity and are probably encrypted. We also found in figure 17b that the website is using z85 [8] as encoding, so we searched for a online decoder [2] and decoded one of the coupons (figure 18a).



(a) Coupon Encoder



(b) Coupon Dencoder

Figure 18: The encoding and decoding process of a coupon

This way, we were able to guess the meaning of the decoded version (*month year - discount*). Although we could not forge a coupon, at least we cracked the encryption.

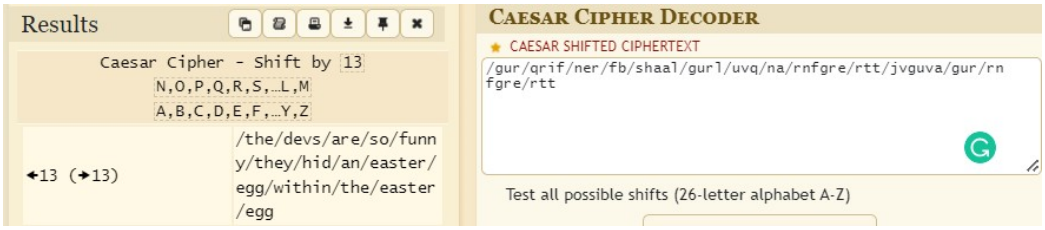
One more form of weak Cryptography can be found in the easter egg, mentioned in section 3.7. As we previously said, we were able to access the easter egg file, whose content is presented in figure 19.

[2] Z85 Decoder: `https://cryptii.com/pipes/z85-encoder`

Figure 19: Easter Egg file

As we can see, the file mentions line 12 as the real easter egg. After some research, we found that this is an encrypted message and, after deconding with base64 [3] [1] and decrypt with caesar cipher [4] [3], figure 20, we found out the decoded version */the/devs/are/so/funny/they/hid/an/easter/egg/within/the/easter/egg* which lead us to `http://localhost:3000/the/devs/are/so/funny/they/hid/an/easter/egg/within/the/easter/egg`, the real easter egg, visible in figure 21.

(a) Base64 Decoder



(b) Caesar Cipher Decryptor

Figure 20: The decoding process of eastere.gg message

[3] Base64 Decoder: `https://en.wikipedia.org/wiki/Base64`
[4] Caesar Cipher Decoder: `https://en.wikipedia.org/wiki/Caesar_cipher`

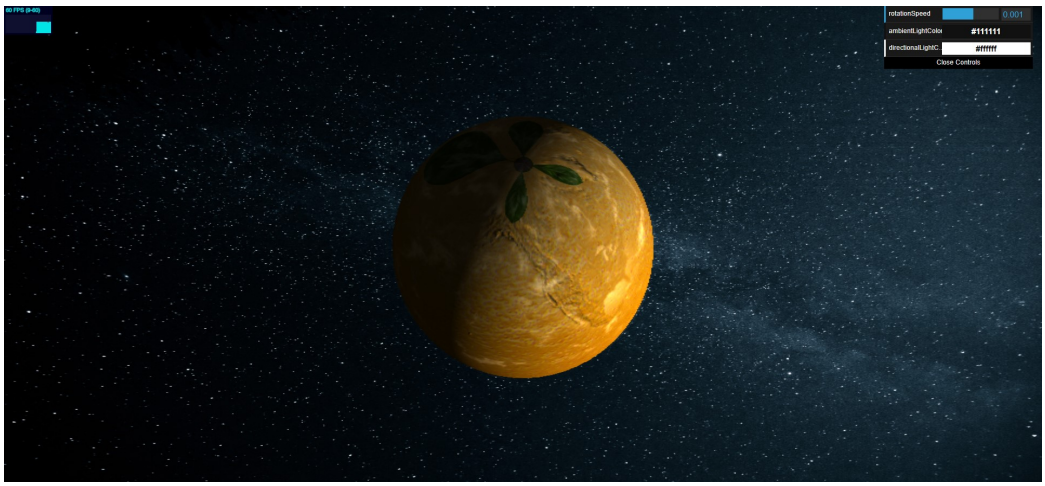


Figure 21: Easter Egg

3.10 Client Side Testing

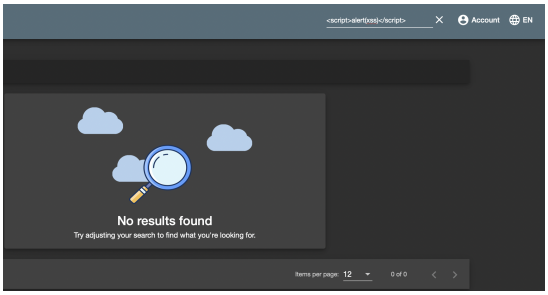
On the search bar, we noticed it is possible to inject *HTML* or scripts. The first try was to put a <*script*> tag in the search bar:

```
<script> alert('xss injection') </script>
```

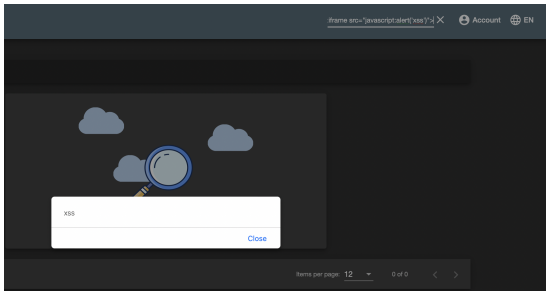
Which does not work, as shown in figure 22a. The second attempt was with an <*iframe*> tag:

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

Which results in an alert on the website, as we can see in figure 22b.



(a) failed



(b) passed

Figure 22: Two types of XSS injection attacks

4 Web application security firewall

4.1 Information Gathering

For this section, the WAF does not hide the information previously obtained. Therefore, reproducing the same procedures in section 3.1, we get the same information.

4.2 Configuration and Deployment Management Testing

In this part, the WAF forbade us to make any review, inclusive of the admin account. The request always gives us the error 403 (forbidden). The first intuition was to reduce the *PARANOIA* level to one, but it did not change anything.

4.3 Identity Management Testing

As shown in figure 7 in section 3.3, the login form presents an error, indicating the existence of an account with this email. Now with the WAF, the error is not shown and is only saying in the browser's inspector console that there is a bad request error (figure 23).

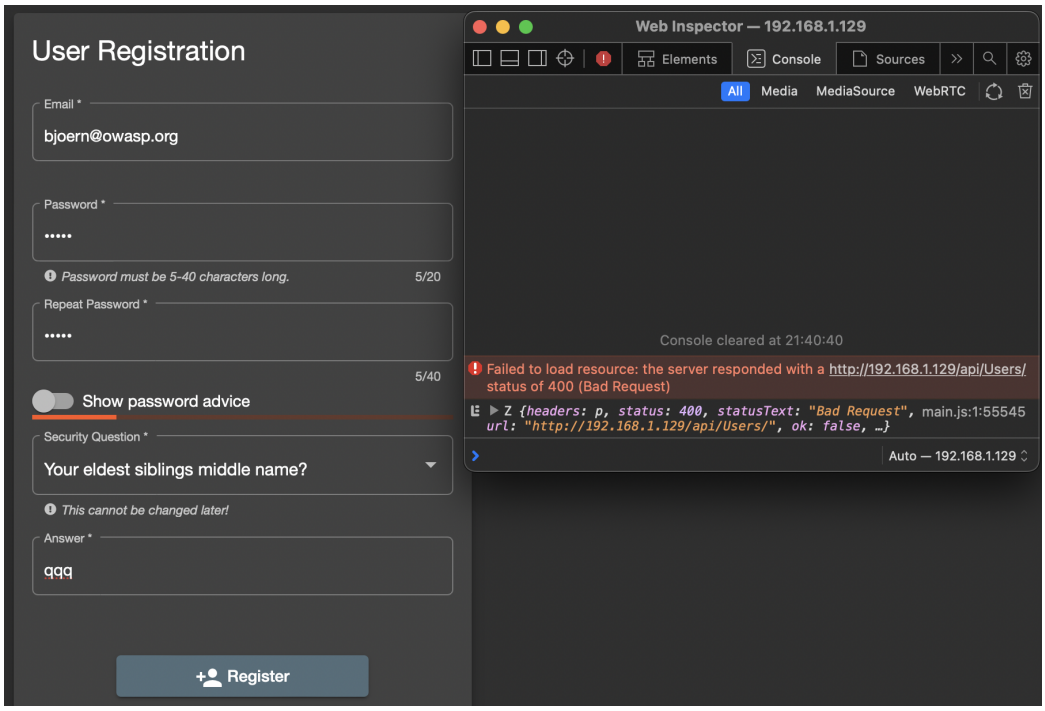


Figure 23: Error message not displayed

4.4 Authentication Testing

The WAF continues authorizing the weak passwords tried in section 3.4. The firewall cannot do anything for the default password because it is on the World Wide Web.

4.5 Authorization Testing

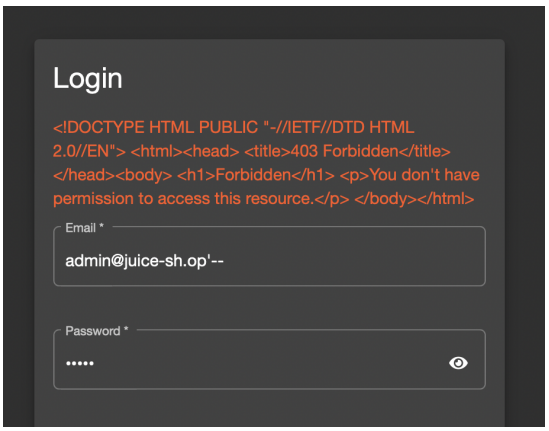
Regarding the confidential file, the firewall could do nothing because it is something that needs to be protected and fixed on the server-side.

4.6 Session Management Testing

This time, the WAF does not work for our session management test cases. For the basket test, it is possible to put another value of the bid variable and see another user's basket. We also get a different user id in the administration panel for the feedback.

4.7 Input Validation Testing

In this section, we tried the SQL and null byte injection from section 3.7. Unfortunately, in this case, they do not work as shown in figure 24.



(a) SQL injection (b) Null byte injection

Figure 24: Injection attacks do not work with WAF

4.8 Testing for Error Handling

The WAF does not handle the error of section 3.8. However, curiously, the HTML page error is outputted in the error label above the email, as we can see in figure 25.

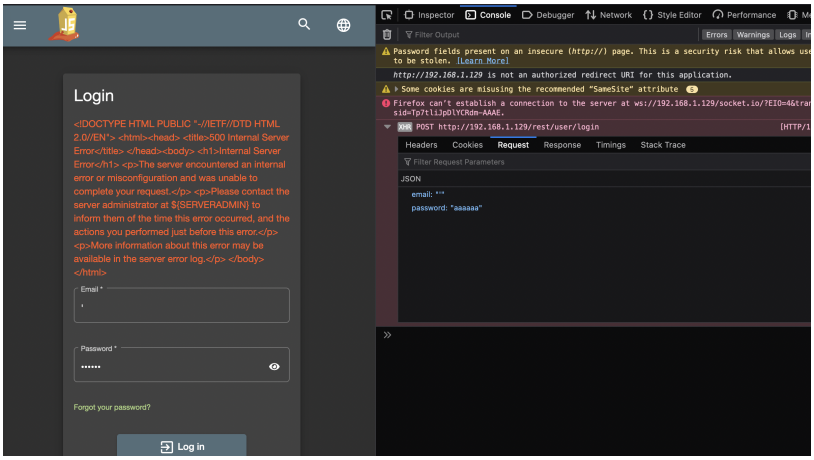


Figure 25: Another type of error

4.9 Client Side Testing

The search bar's <iframe> injection does not work. This is because the WAF does not handle it.

5 Conclusion

We found this project interesting, mainly in the first phase, where we learned various web application penetrations techniques. We also learned how to mitigate a significant part of those problems using a firewall. However, the WAF does not respond to all needs, and there are some cases where the use of additional security is indispensable.

References

[1] *Base64 decode and encode - online.* URL: `https://www.base64decode.org/`.

[2] *BURP suite - application security testing software.* URL: https://portswigger.net/burp.

[3] dCode. *Caesar cipher (shift) - online decoder, encoder, solver, translator.* URL: https://www.dcode.fr/caesar-cipher.

[4] Juice-Shop. *Juice-shop/juice-shop: Owasp Juice Shop: Probably the most modern and sophisticated insecure web application.* URL: https://github.com/juice-shop/juice-shop.

[5] Ferruh Mavituna. *SQL Injection Cheat Sheet.* May 2022. URL: https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/.

[6] *Null byte injection.* URL: https://www.whitehatsec.com/glossary/content/null-byte-injection.

[7] Owasp. *Owasp/WSTG: The Web Security Testing Guide is a comprehensive open source guide to testing the security of web applications and web services.* URL: https://github.com/OWASP/wstg.

[8] *Z85.* URL: https://www.npmjs.com/package/z85.