Task 1: Exploiting OWASP Juice Shop

2. Methodology

2.1 Part 1: Environment Setup

Virtual machine: Kali Linux

Step 1: Install Node.js

Node.js is a JavaScript runtime that executes server-side JavaScript code. Since OWASP Juice Shop is built with Node.js, it is required for running the application. Without Node.js, Kali Linux will not be able to interpret or execute the backend code.

Figure 1: Installation of nodejss

Step 2: install npm

npm (Node Package Manager) is used to install the dependencies required by Juice Shop, such as Express.js, Angular, and SQLite. These are defined in the package.json file. When executed, npm downloads all the necessary packages into a node_modules folder.

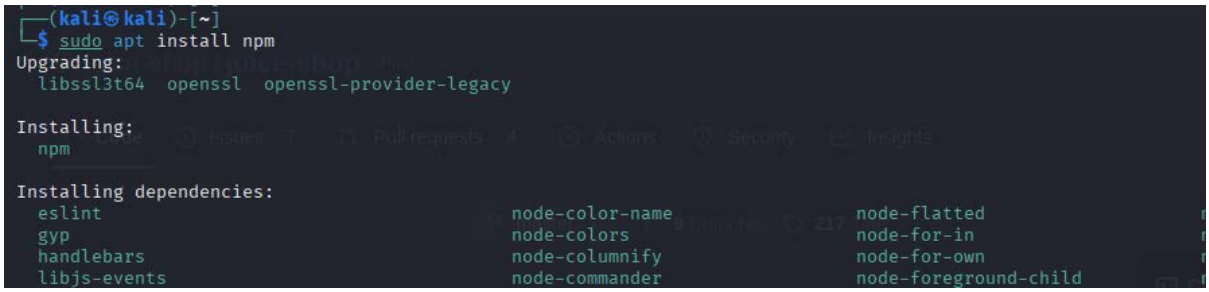


Figure 2: Installation of npm

Step 3: Install OWASP Juice Shop

OWASP Juice Shop can be installed using npm after cloning or downloading the project from its official GitHub repository.

Figure 3: Installing OWAS juice shop

Figure 4: Downloaded Directories

Step 4: Install Dependencies

After installation, the package.json file is read by npm to automatically download and install all required dependencies.

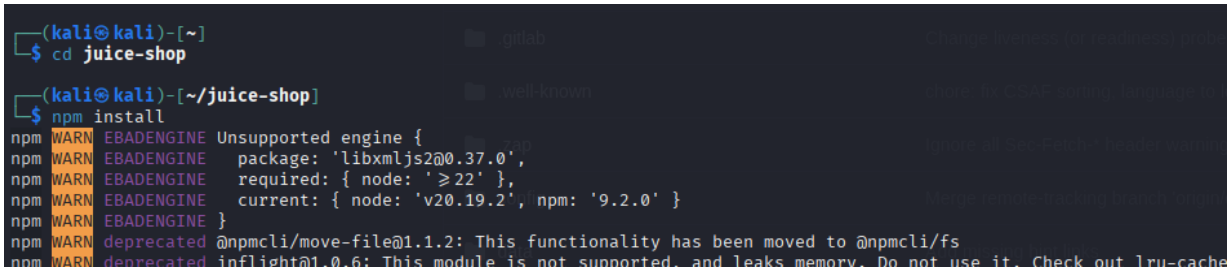


Figure 5: Installation of all the dependencies

Step 5: Start OWASP Juice Shop

To launch the application, the npm start command is executed. This starts the server and binds it to a local port, which in our case http://localhost:3000.

```
┌──(kali㉿kali)-[~/juice-shop]
└─$ npm start

> juice-shop@18.0.0 start
> node build/app

info: Detected Node.js version v20.19.2 (OK)
info: Detected OS linux (OK)
info: Detected CPU x64 (OK)
info: Configuration default validated (OK)
info: Entity models 19 of 19 are initialized (OK)
info: Required file server.js is present (OK)
info: Required file index.html is present (OK)
info: Required file styles.css is present (OK)
info: Required file main.js is present (OK)
info: Required file tutorial.js is present (OK)
info: Required file runtime.js is present (OK)
info: Required file vendor.js is present (OK)
info: Port 3000 is available (OK)
info: Domain https://www.alchemy.com/ is reachable (OK)
info: Chatbot training data botDefaultTrainingData.json validated (OK)
info: Server listening on port 3000
```

Figure 6: Running npm start and showing the local port

Step 6: Access OWASP Juice Shop via Browser

Once the server is running, the application is accessible through any web browser. The main product page displays various items and interactive elements.

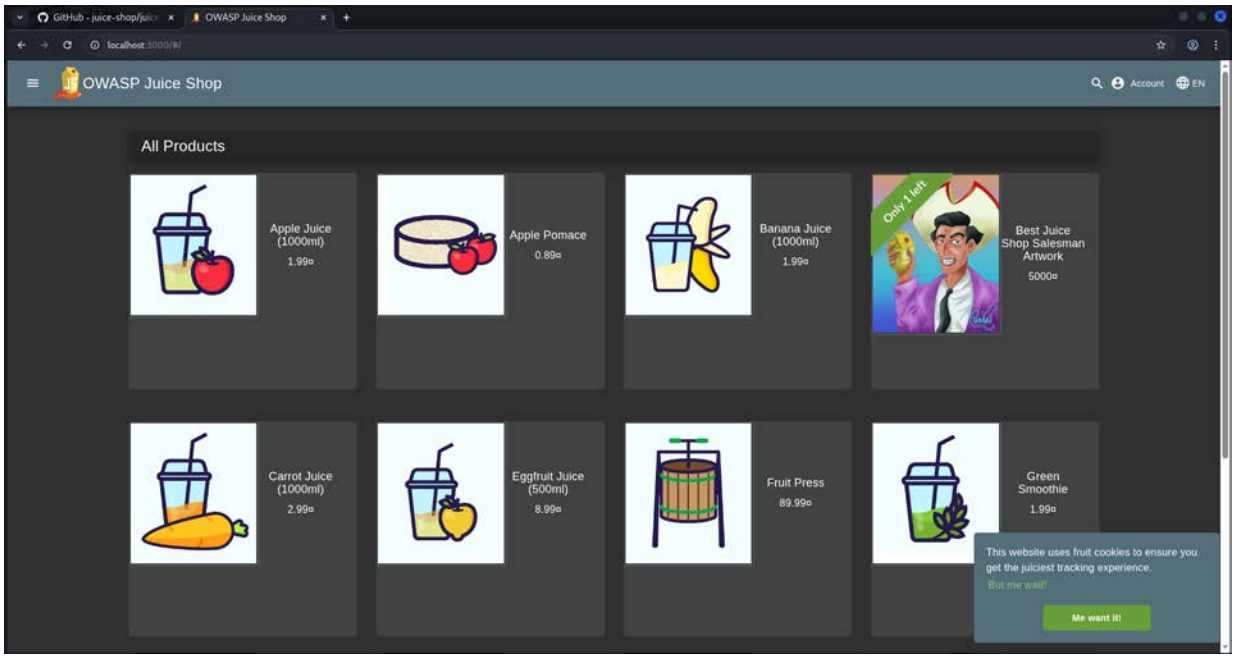


Figure 7: OWASP Juice Shop Product Page

Figure 7 displays the main product page of the OWASP Juice Shop web application. The screen shows a product catalog featuring various items for sale. This publicly accessible

storefront provides an entry point for security testing, offering multiple interactive features that could be explored for potential vulnerabilities such as improper input handling, insecure authentication, or client-side weaknesses.

2.2 Part 2: Information Gathering

2.2.1 Automated Scanning

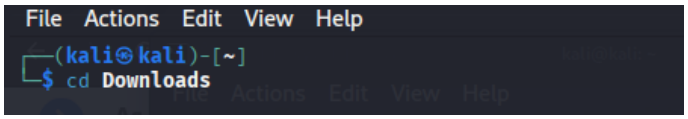
OWASP ZAP and Skipfish were used to perform automated scanning of the OWASP Juice Shop web application. These tools helped identify potential vulnerabilities and misconfigurations by crawling and analyzing the application's structure and behavior.

2.2.1.1 OWASP ZAP Scanning Steps

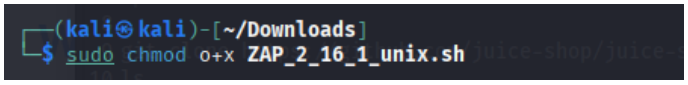
Step 1: Install OWASP ZAP

To install OWASP ZAP, first we downloaded zap linux installer on https://www.zaproxy.org, then located the download zap installer and change directory which in our case the Downloads folder. The following steps were taken:

1. Changed directory:



2. Modified file permission to make it executable:



   What this command does: It adds execute permission for others (not the owner or group) on the installer file, allowing them to run the script. We do this because by default, a downloaded .sh file (shell script) may not have execute permissions, especially if downloaded from the internet or another machine. So when you try to run it with without giving permission you might get a "Permission denied" error. Adding execute permission with chmod ensures the system allows the script to be executed.

3. Ran the installer:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ sudo ./ZAP_2_16_1_unix.sh
Starting Installer ...
^[[A^[[A^[[A
```

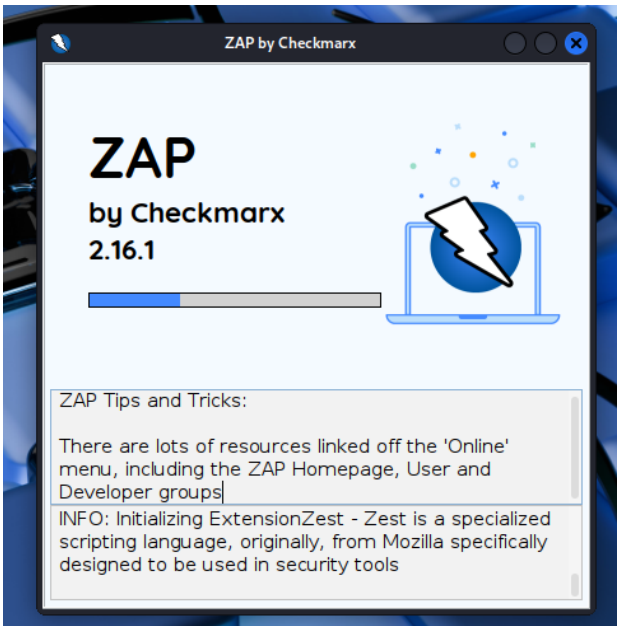


Figure 8: ZAP installation GUI

Step 2: Launch OWASP ZAP

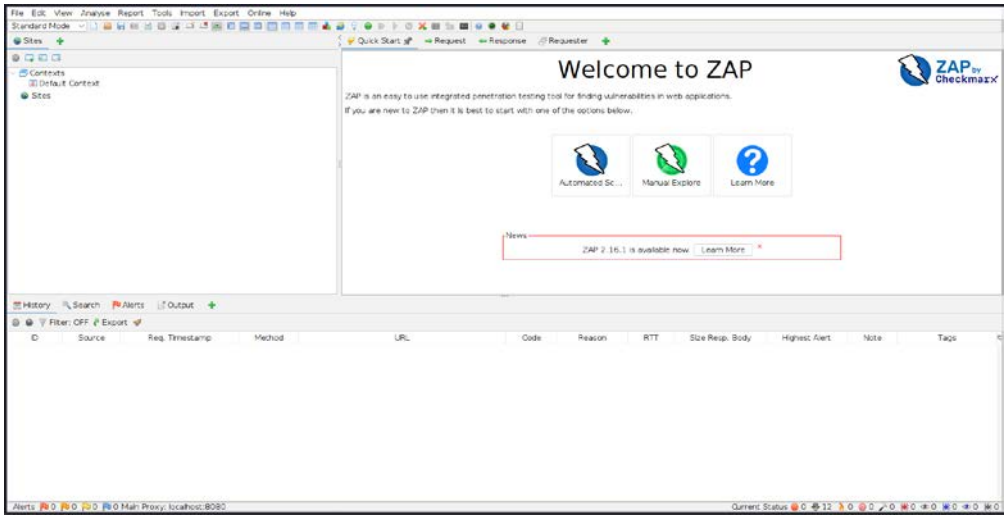


Figure 9: OWASP ZAP's main window

Step 3: Start Automated Scanning

The target URL for the OWASP Juice Shop was inserted into ZAP, and a scan was initiated. ZAP scanned the web app and identified some vulnerabilities such as:

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

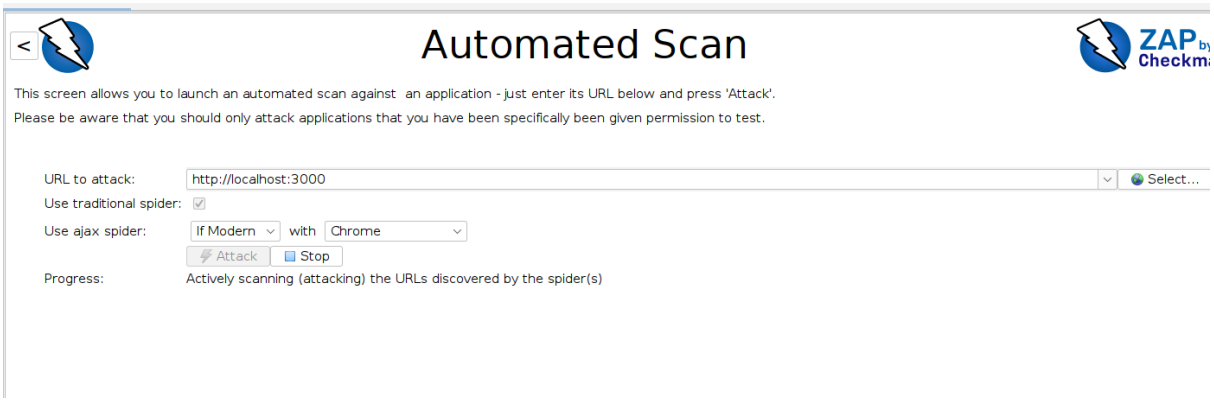


Figure 10: Juice Shop URL being inserted into ZAP for scanning.

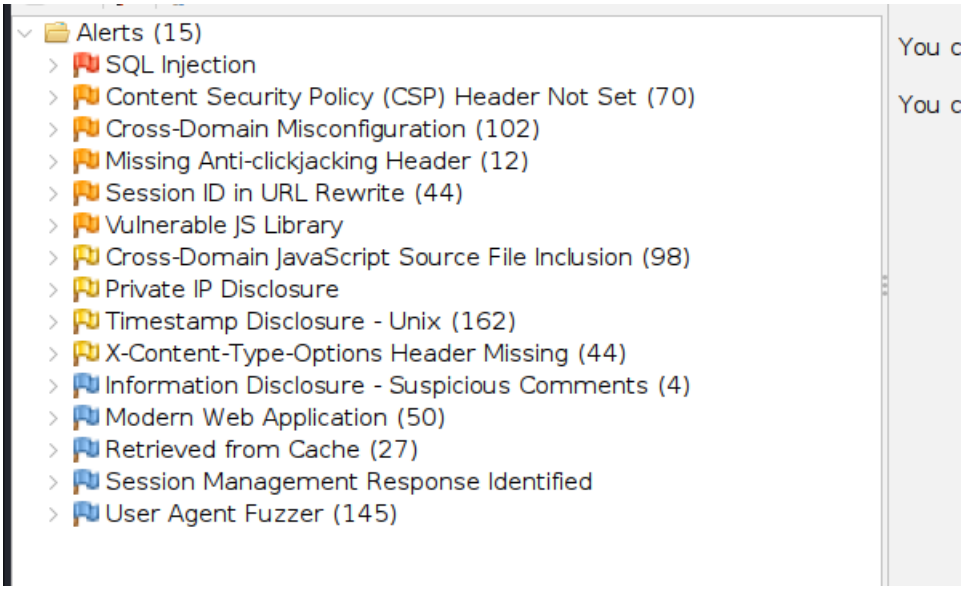


Figure 11: OWASP juice shop Zap scan results

2.2.1.2 Skipfish Scanning Steps

Step 1: Install Skipfish

Skipfish is a web application security scanner that performs recursive, crawl-based vulnerability assessments. It is typically pre-installed in Kali Linux. To confirm its presence, the following command was executed:

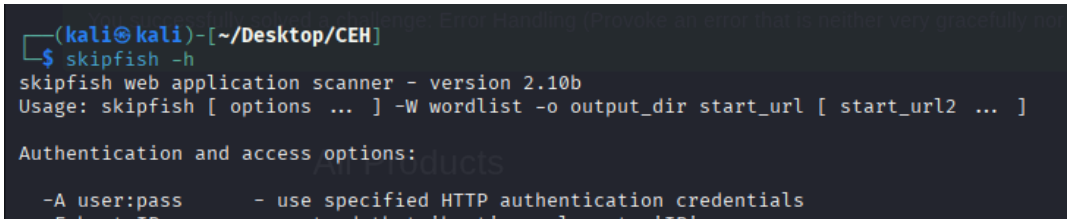


Figure 12: Command used to check if skipfish is installed

Step 5: Run Skipfish

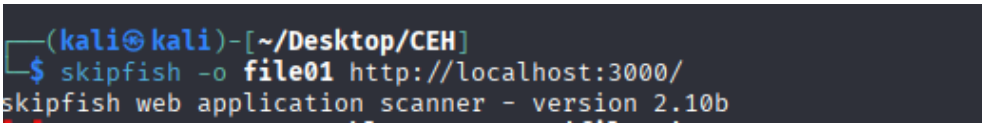


Figure 13: Command to run skipfish

Figure 14: Automated Scanning using Skipfish in terminal

After the scan is executed against the Juice Shop, the results of found vulnerabilities will be saved in an HTML report.

After the Skipfish scan was completed, we navigated to the directory we created during the scanning process, as shown in *Figure 15*.

Figure 15: Directory Created by Skipfish (file01)

Within this directory, we located the index.html file, which contains the report generated by Skipfish, as highlighted in *Figure 16*.

Figure 16: Location of the index.html Report File in file01

By opening the index.html file, we were able to view the automated vulnerability scanning results for OWASP Juice Shop. These results are illustrated in *Figure 17.*

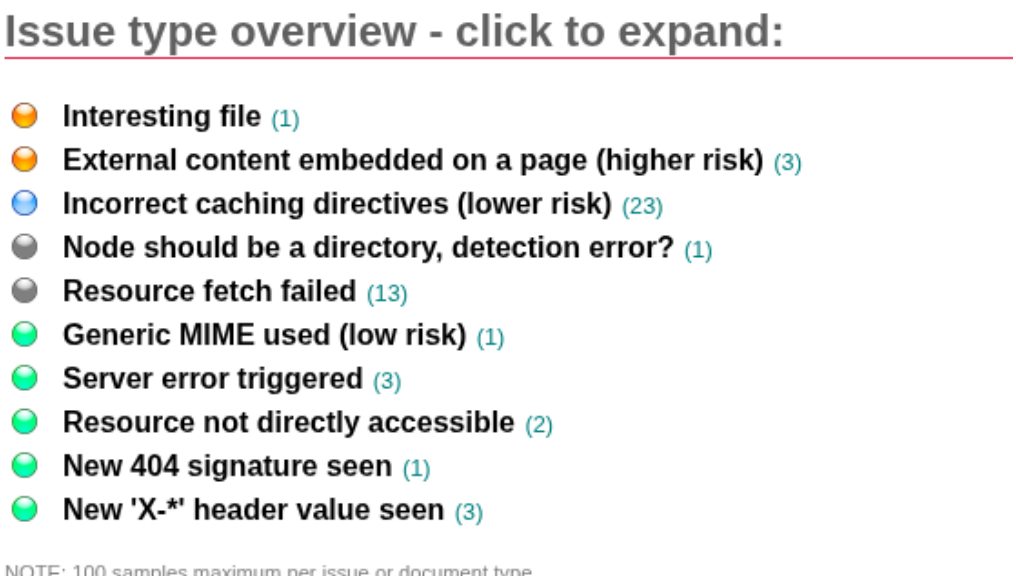


Figure 17: OWASP Juice shop automated Scanning result using Skipfish

2.2.2 Manual Exploration

Manual exploration of the OWASP Juice Shop application was conducted using browser developer tools and OWASP ZAP. The assessment began with exploring the web interface, during which the admin email address was identified within one of the product review comments. URL manipulation techniques were applied to discover hidden paths such as the score-board (as shown in Figure 18) and administration page (as shown in Figure 28), revealing non-linked functionality and admin-only areas that could be exploited in the absence of proper access controls.

Injection flaws were identified using both manual and automated techniques. SQL injection vulnerabilities were tested with OWASP ZAP's fuzzing feature on the login endpoint, successfully bypassing authentication by injecting payloads like 1' OR 1=1 --. Cross-Site Scripting (XSS) vulnerabilities were also discovered, including:

- Stored XSS, where JavaScript payloads were saved in the server (e.g., within comments),
- Reflected XSS, where the payload was reflected in the response (e.g., through search queries)
- DOM-based XSS, where input was injected into the search box (e.g., <iframe src="javascript:alert('xss')">) and executed by client-side JavaScript.

Brute-force attacks were performed on the login page and password reset functionality using OWASP ZAP's forced browsing and fuzzing tools, targeting the known user admin@juice-sh.op. By analyzing server responses such as HTTP status codes and response bodies, valid credentials were eventually discovered. The application lacked key protections like CAPTCHA, rate limiting, or account lockout mechanisms, which allowed repeated login attempts.

Command injection was also tested in areas like the feedback and file upload functionality. The application accepted files without proper extension filtering, allowing upload of files without .pdf or any extension, and potentially leading to code execution. Authentication bypass attempts were also conducted, validating weak session handling and error-based feedback during login and password recovery processes.

Overall, this manual exploration revealed critical vulnerabilities in Juice Shop, including injection flaws, broken authentication, lack of brute-force protection, XSS variants, file upload issues, and exposed admin areas. These findings reflect common real-world weaknesses and highlight the importance of secure input handling, access control, and layered security in web applications.

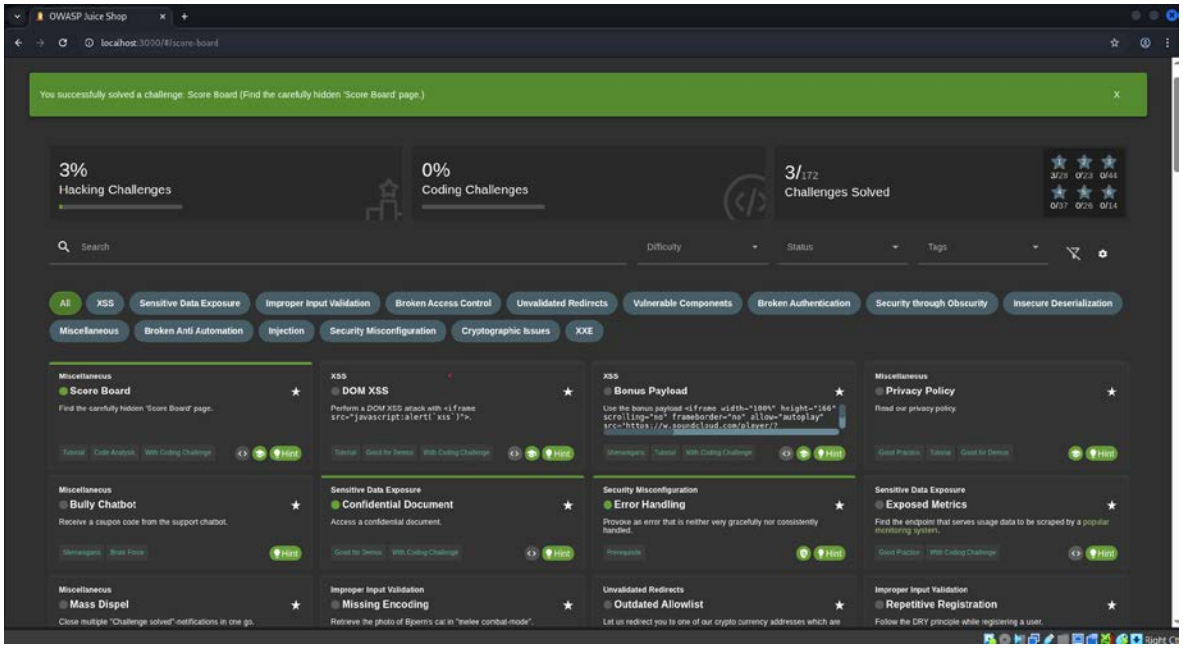


Figure 18: OWASP Juice shop Score-board

2.3 Part 3: Vulnerability Discovery and Exploitation

This section presents the key vulnerabilities identified during the security assessment of the OWASP Juice Shop application, along with their associated CVSS (Common Vulnerability Scoring System) Base Scores calculated using version 3.1. Each vulnerability was discovered through a combination of manual exploration and automated testing using OWASP ZAP and browser developer tools. Exploitation techniques such as input manipulation, forced browsing, fuzzing, and payload injection were used to demonstrate the real-world impact of each flaw. For each identified issue, appropriate remediation strategies are proposed to harden the system against future attacks.

No	Vulnerability	CVSS Base Score
1	SQL Injection	8.4 (High)
2	Cross-Site Scripting (XSS)	5.1 (Medium)
3	Authentication Bypass	8.4 (High)
4	File Type Upload	8.4 (High)
5	Command Injection	8.4 (High)
6	Cross-Site Request Forgery (CSRF)	5.3 (Medium)
7	Forged Review	6.5 (Medium)

Table 1: OWASP Juice Shop Vulnerabilities and CVSS Base Score (3.1)

Severity	Severity Score Range
None	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

Table 2: CVSS v3.x Rating

2.3.1 SQL Injection

Severity Level: CVSS 8.4 (High)

SQL Injection is a critical web application vulnerability that occurs when an attacker can manipulate SQL queries by injecting malicious input through user-controlled fields. In this exploitation, Sql injection was first performed by accessing the login page and used OWASP ZAP to capture and manipulate the login request. A fuzzing attack was conducted on the /rest/user/login endpoint by injecting a payload like "1' OR 1=1 --" into the email field. This payload altered the backend SQL logic, causing the condition to always evaluate as true, thereby bypassing authentication without needing valid credentials.

Further exploitation involved tampering with the HTTP POST request body to modify the email and password values, enabling to log in as another user or even escalate privileges within the application.

Exploitation Goals:

- Bypass authentication by injecting SQL payloads into login fields.
- Trick the SQL engine into treating the condition as always true.
- Gain unauthorized access to the application without valid credentials.
- Modify POST request data to impersonate other users or escalate privileges.

2.3.1.1 Steps to Reproduce and Exploitation Process

Step 1: Lunch Juice shop from chrome browser or preferred web application on Zap.

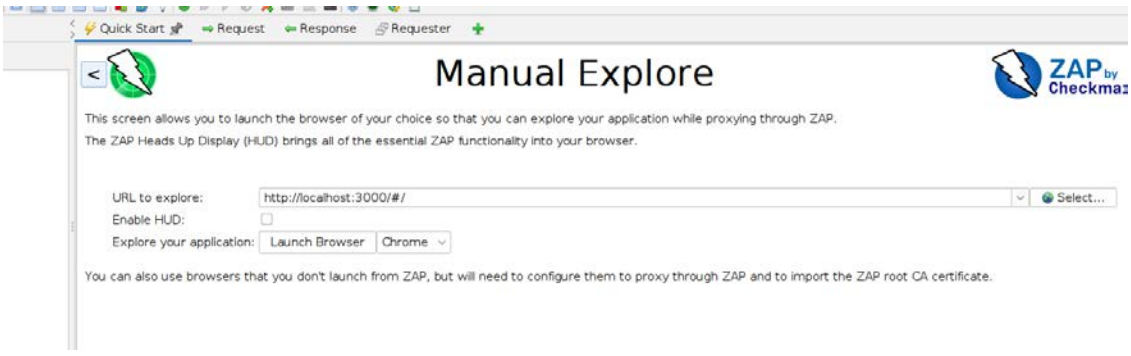


Figure 19: Initial Setup for SQL Injection Testing Using ZAP

Figure 19 shows the "Manual Explore" interface in OWASP ZAP, where we will launch the Juice Shop application in a Chrome browser through ZAP. The URL **http://localhost:3000/#/** is entered. This setup allows ZAP to intercept and analyze traffic between the browser and the Juice Shop application, marking the first step in testing for SQL injection vulnerabilities.

Step 2: Access the login page through Account and try to login to juice shop

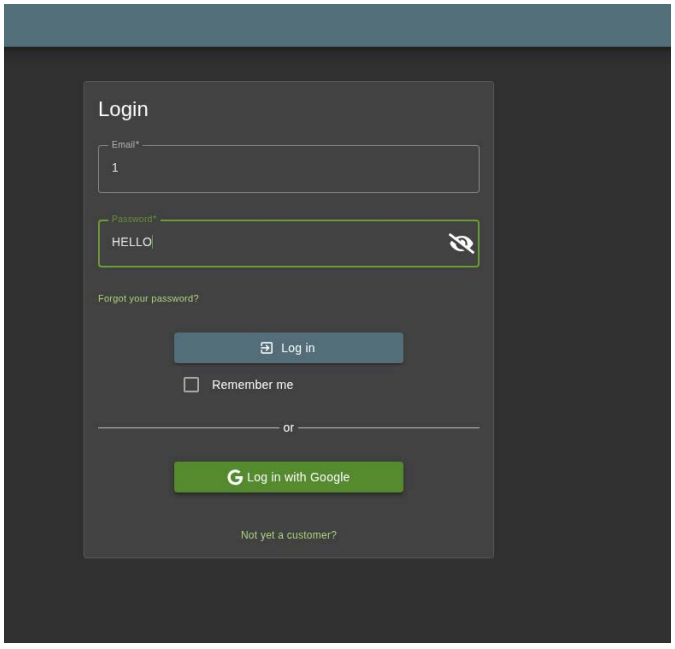


Figure 20: Login Page of OWASP Juice Shop

Step 3: Go back to the Zap application and on the History panel locate the login API endpoint used to process your username and password. “http://localhost:300/rest/user/login”

Figure 21: ZAP history tab

Figure 21 shows the History tab in OWASP ZAP, highlighting a POST request to http://localhost:3000/rest/user/login. This request attempts to log in to the Juice Shop application and is important for testing SQL injection. The POST method is used because login credentials (email and password) are sent in the request body, not in the URL. This is common for secure form submissions. Targeting the POST request allows us to inject SQL payloads into the login fields to bypass authentication and access the admin section or other restricted parts of the application.

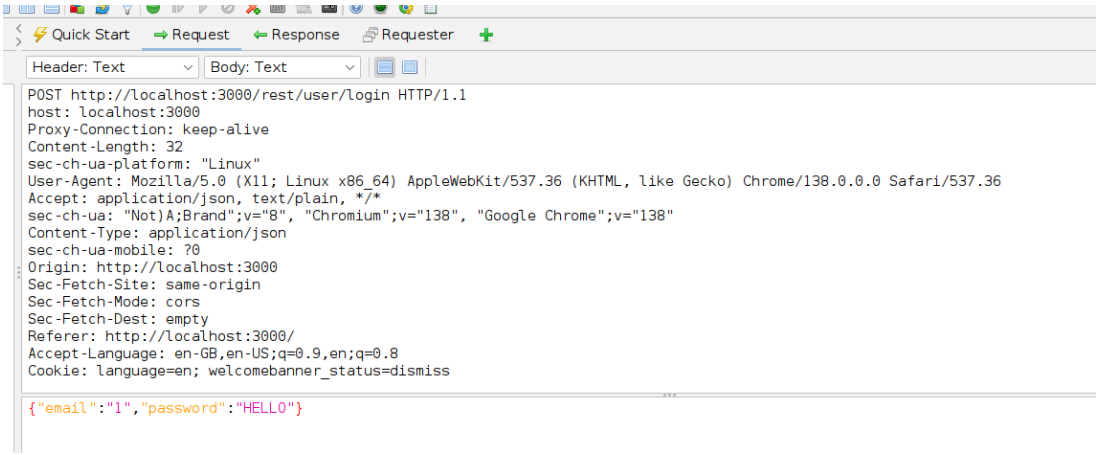


Figure 22: The Request Line, Header and Payload of the http request sent

Figure 22 shows the HTTP POST request to the /rest/user/login endpoint captured in OWASP ZAP after logging into Juice Shop. The request includes the user's email and password, along with headers such as Content-Type. This figure highlights how user credentials are transmitted to the server and can be analyzed for potential vulnerabilities like SQL injection, weak authentication, or insecure data handling.

Step 4: Lunch a Fuzz Attack

By launching a fuzzing attack on the login endpoint, we simulated how malicious input could be injected into the authentication process.

To begin the fuzzing process, right-click on the http://localhost:3000/rest/user/login request in OWASP ZAP and selected Attack then select Fuzz as shown in ***figure 23***. This action

opened the Fuzzer configuration window, where it will allow us to select the login request body as the injection point.

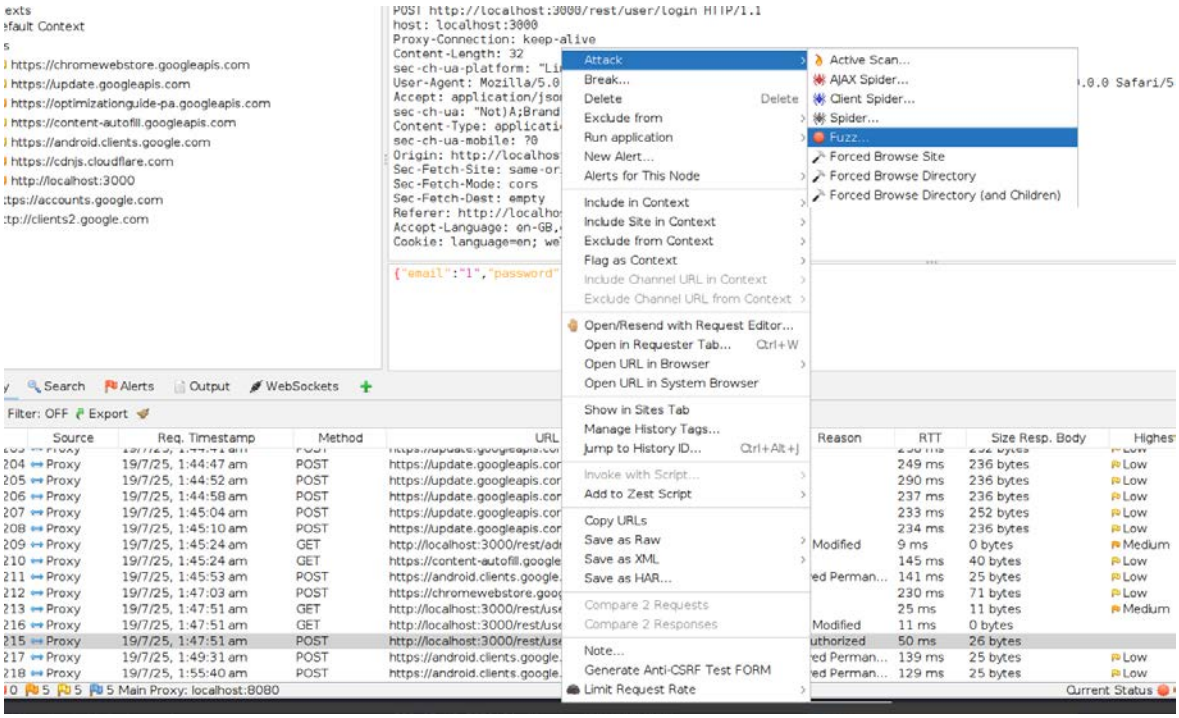


Figure 23: Launching a Fuzz Attack on the Login Endpoint

Next, add a SQL injection payload to the email field using the format: {"email":" ' or 1=1 --","password":"HELLO"}.

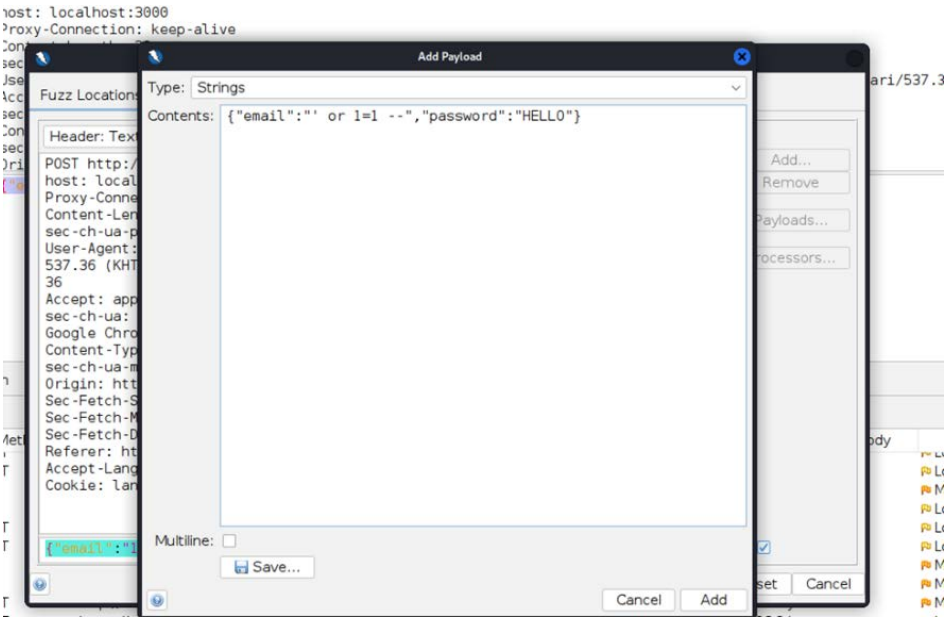


Figure 24: Adding a SQL injection payload

Figure 24 shows a fuzzing payload being added in OWASP ZAP targeting the Juice Shop login endpoint. This SQL injection attempts to bypass authentication by injecting the expression 1=1, which always evaluates as true, while the -- comment sequence causes the database to ignore the rest of the query.

Finally, start the fuzzing process.

Figure 25: Result of Fuzz attack

Figure 25 shows the results captured in the Fuzzer tab. While the original request received a 401 Unauthorized response, the fuzzed request with the SQL injection payload returned a 200 OK response. This indicates that the login was successfully bypassed, confirming the existence of a SQL injection vulnerability in the login form of OWASP Juice Shop.

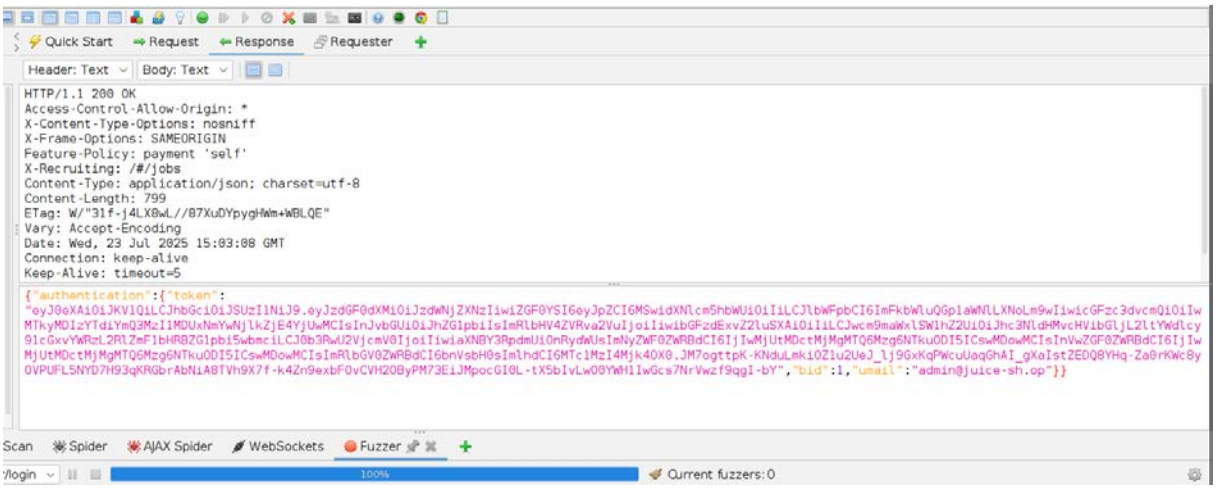


Figure 26: Successful Authentication Response after SQL Injection

Figure 26 shows the HTTP response captured in OWASP ZAP following the SQL Injection attack on the login page of OWASP Juice Shop, confirming successful authentication as the admin user (admin@juice-shop.shop) and elevated privileges. This proves the SQL injection was successful and allowed unauthorized access to an administrator account.

Step 5: Go to the login page of the OWASP Juice shop and log in as administrator

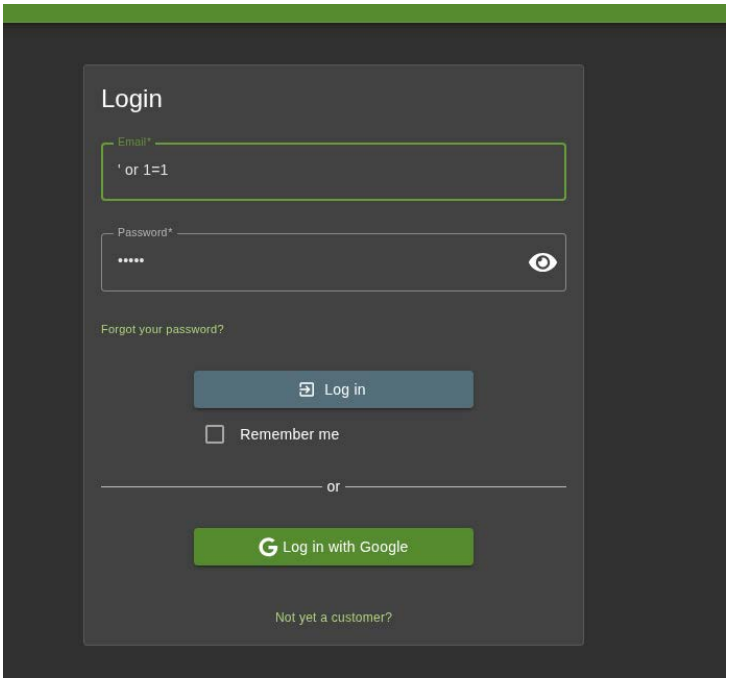


Figure 27: Login Attempt Using SQL Injection Payload

Step 6: Access Administration page with the url: http://localhost:3000/administration

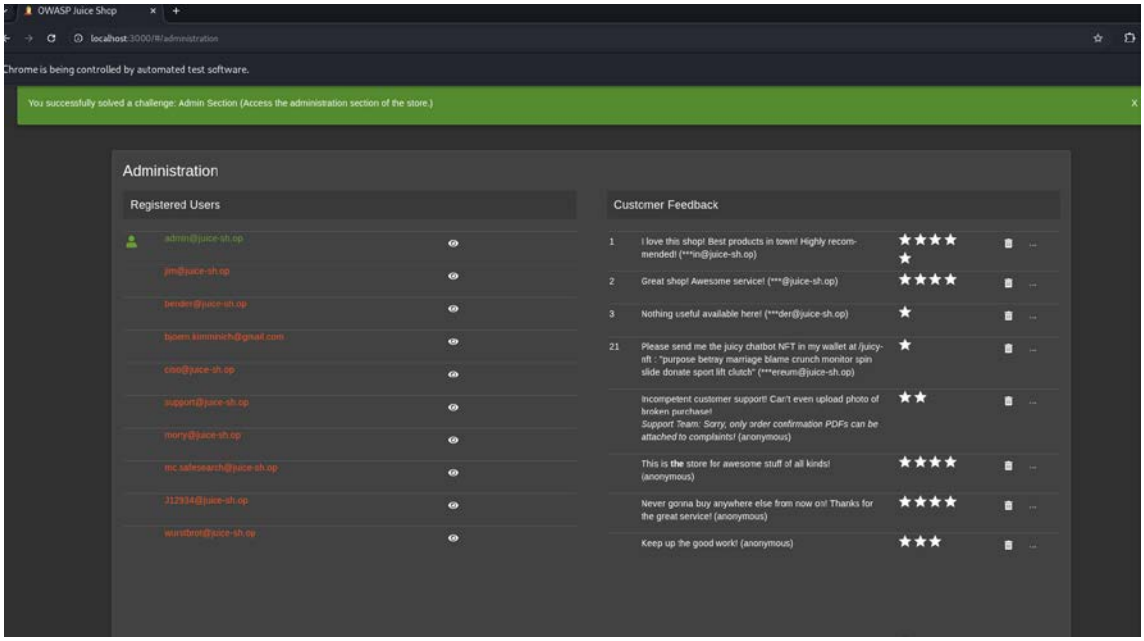


Figure 28: Accessing the Admin Section of OWASP Juice Shop

Figure 28 displays the "Administration" interface of the OWASP Juice Shop application, which became accessible after successfully exploiting a SQL injection vulnerability. This figure demonstrates the impact of unauthorized access due to weak input validation:

- Registered Users Panel (Left Side): Shows a list of user accounts, including the administrator. Each user entry includes an eye icon that allows viewing detailed user information.
- Customer Feedback Panel (Right Side): Displays submitted feedback from users, including comments and star ratings. Trash icons next to each entry enable the attacker to delete feedback records.

  This figure emphasizes the severity of SQL injection attacks, showing how an attacker can gain admin-level access and manipulate sensitive application data.

2.3.1.2 Suggested remediation

To mitigate SQL Injection vulnerabilities, the following remediation steps should be implemented:

1. Use parameterized queries (prepared statements) to prevent direct injection into SQL statements.
2. Sanitize and validate all user inputs both on the client and server side.
3. Apply the principle of least privilege to database accounts used by the application.
4. Monitor for abnormal login behaviors using intrusion detection systems (IDS).
5. Conduct frequent penetration testing and code reviews to detect vulnerabilities early.

By implementing these measures, developers can significantly reduce the risk of SQL injection and protect the application from unauthorized access and data breaches.

2.3.2 Cross-Site Scripting (XSS)

Severity Level: CVSS 5.1 (Medium)

Cross-Site Scripting (XSS) is a security vulnerability that allows attackers to inject and execute malicious scripts in a trusted website viewed by other users. It occurs when applications fail to properly validate or sanitize user inputs. In this task on OWASP Juice Shop, we exploited three types of XSS: Stored, Reflected, and DOM-based.

- Stored XSS involved injecting payloads that are saved on the server and executed when other users access the affected page.
- Reflected XSS used malicious code in a URL parameter to trigger an alert when visiting a crafted link.
- DOM-based XSS exploited client-side JavaScript by injecting scripts that the browser directly executed from the DOM.

These attacks demonstrate how unfiltered input and insecure rendering of content can expose both data and users to serious threats.

Exploitation Goals:

- Demonstrate Stored XSS by extracting a JWT token and sending it in a malicious curl request.

- Perform Reflected XSS by injecting a script into the URL (id parameter) to trigger a JavaScript alert.
- Exploit DOM-based XSS by injecting JavaScript into the search box that is rendered directly by the browser.
- Highlight the lack of input sanitization and output encoding that makes these attacks possible.

2.3.2.1 Steps to Reproduce and Exploitation Process

2.3.2.1.1 Stored XSS: Payload saved to the server

Step 1: Inspect the webpage and find the JSON Web Token (JWT) under Storage > Local Storage.

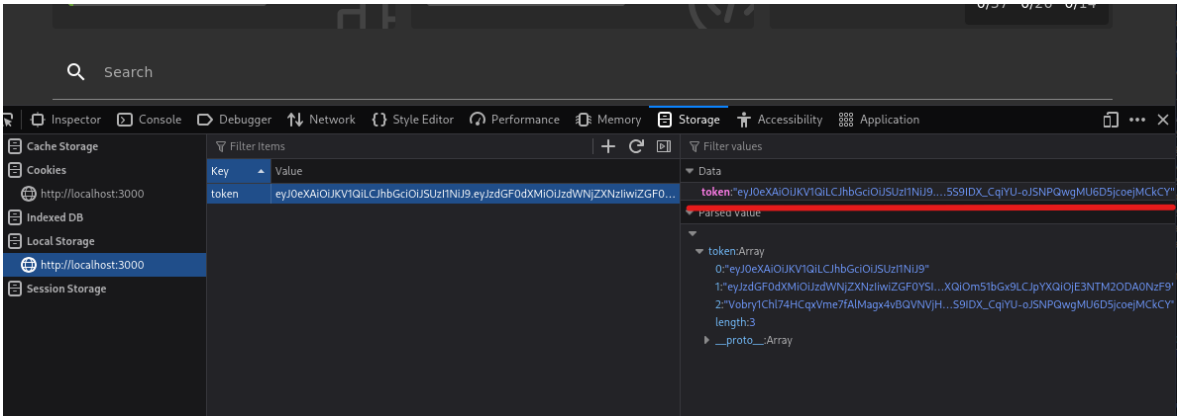


Figure 29: Screenshot of Juice shop web browser's developer tools storage

Figure 29 shows the browser's developer tools open to the Local Storage section under the Storage tab. The screenshot captures a JWT (JSON Web Token) stored under the key named "token" for the domain http://localhost:3000 (owasp juice shop). The token is displayed as a long base64-encoded string starting with eyJ..., which represents its three parts: the header, payload, and signature. This step is part of analyzing stored XSS by locating and examining authentication tokens saved on the client side.

Step 2: After obtaining the token, generate the curl POST command and send it in Linux bash.

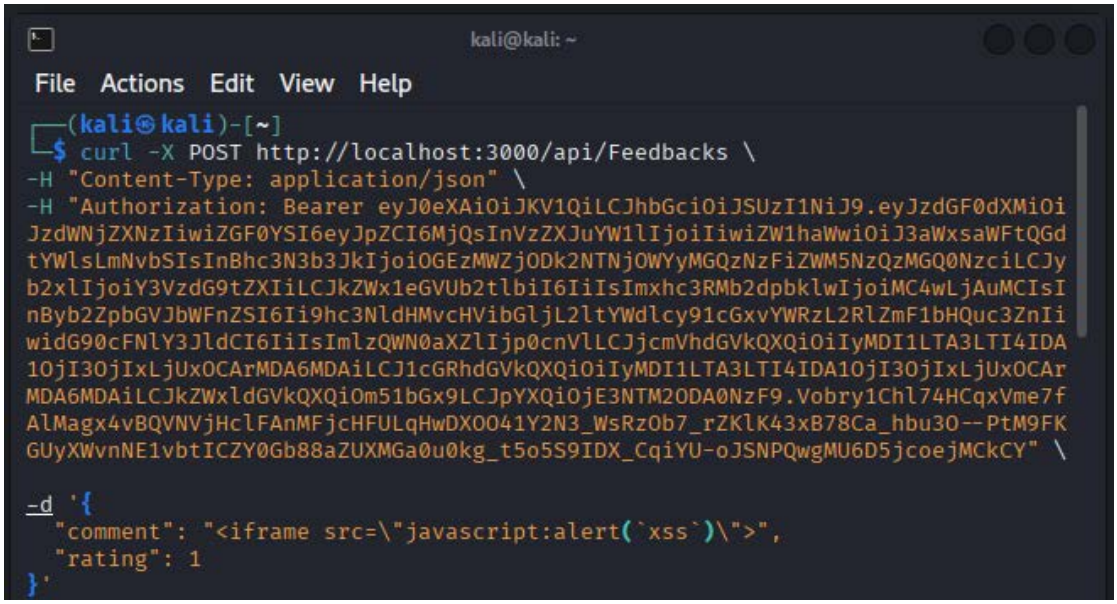


Figure 30: Executing a curl POST Command with a JWT in Linux Bash

Figure 30 shows the terminal where a curl command is used to send a POST request to owasp juice shop. The command used includes a JSON payload and an Authorization header containing a Bearer token specifically the JWT, which was obtained earlier in step 1 (refer to Figure 29). This demonstrates how the token is used to authenticate the request.

2.3.2.1.2 Reflected XSS: Payload in the URL

In this hack, we'll perform a basic reflected XSS attack on the Juice Shop app by modifying a parameter value in the URL.

Step 1: Locate Order History page

- Start by clicking on Account in the top-right corner,
- Go to Orders and Payment,
- Select Order History.

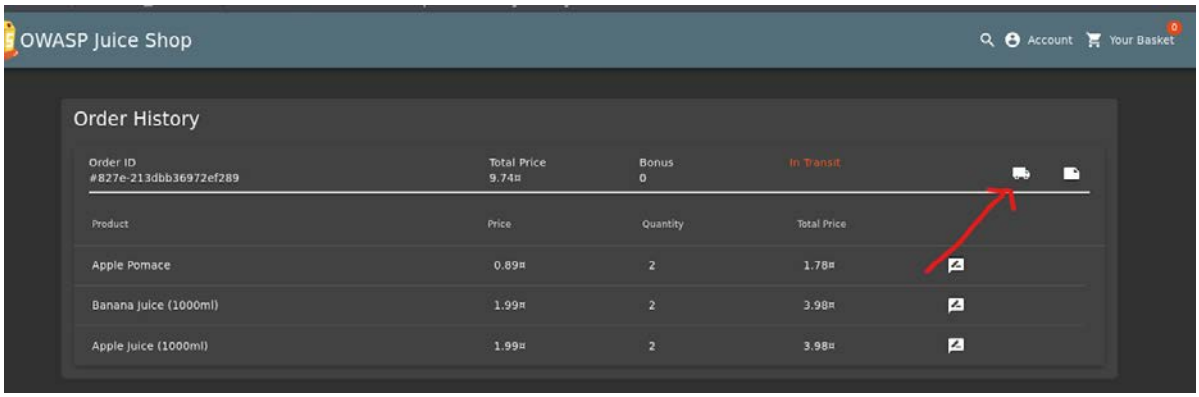


Figure 31: Screenshot of Order History Page

Step 2: Access the expected delivery page

- Click on the truck icon as show in ***figure 31.***

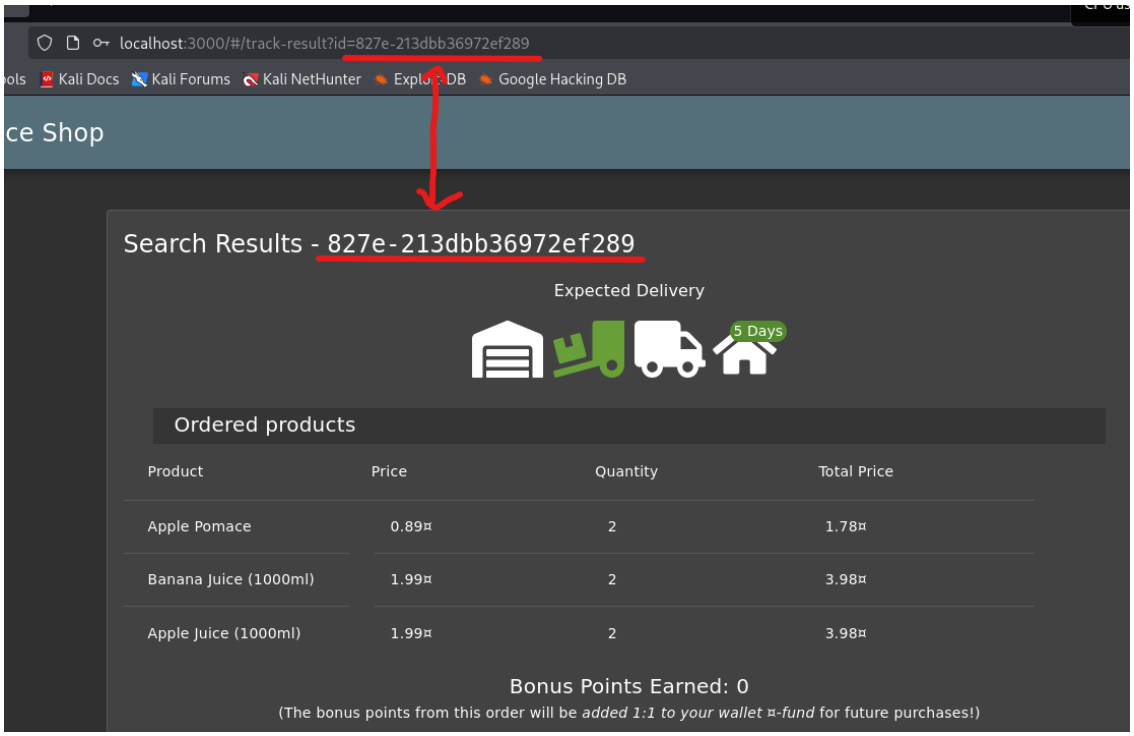


Figure 32: Screenshot of the expected delivery page

Figure 32 shows the "Expected Delivery" page, which shows the search results. Look closely at the URL and notice that the parameter value is not encoded or passed through a trusted HTML link, making it vulnerable.

Step 3: Reflected XSS attack

- Paste the following URL: http://localhost:3000/#/track-result?id=<iframe src="javascript:alert(`xss`)"> .

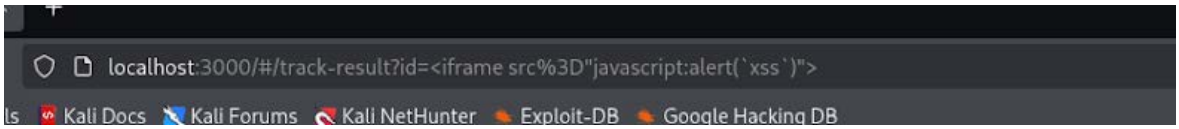


Figure 33: Screenshot of the modify url

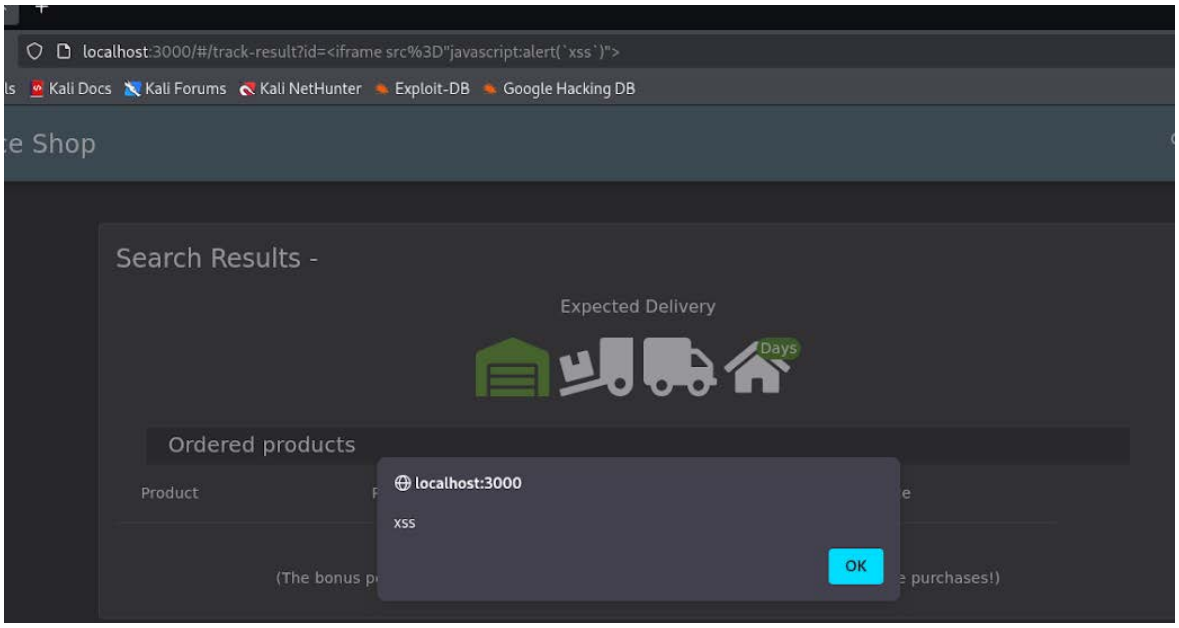


Figure 34: Successful Reflected XSS Attack

Figure 34 shows the result of the reflected Cross-Site Scripting (XSS) attack. The screenshot captures a JavaScript alert box displaying the message "xss", which appears after injecting a malicious payload into the id parameter of the URL. This confirms that the input was reflected and executed without proper sanitization, demonstrating a successful exploitation of a reflected XSS vulnerability.

2.3.2.1.3 DOM-based XSS: Exploits client-side JavaScript directly.

Step 1: inject javascript into search box (e.g. <iframe src="javascript:alert(`xss`)">)

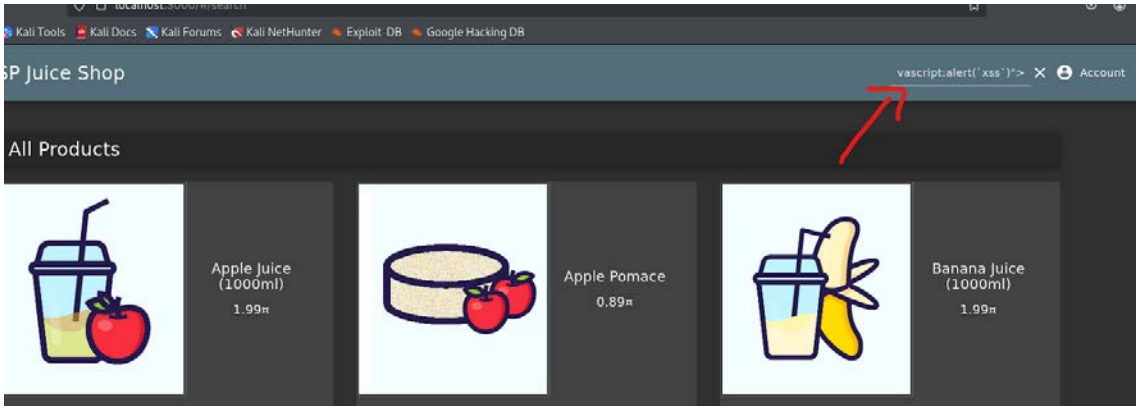


Figure 35: screenshot of javascript in the search bar

Step 2: Observe the Result

- After executing the search, the application will render the injected script in the page's DOM (Document Object Model).
- A JavaScript alert box will appear on the screen as seen in ***Figure 36***

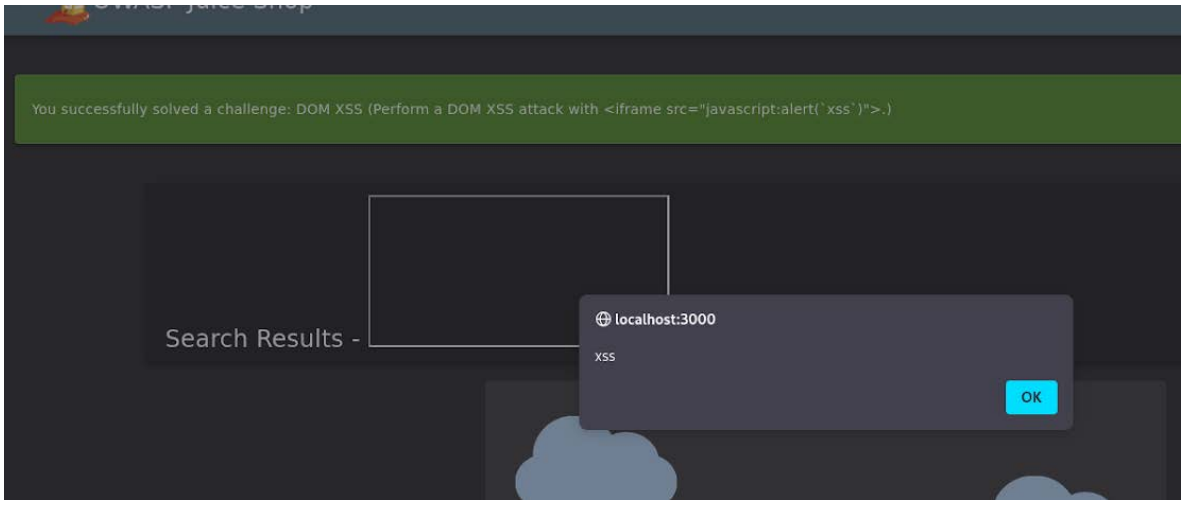


Figure 36: Screenshot of DOM-Based XSS alert box triggered

Figure 36 displays the result of executing a DOM-based XSS attack. The screenshot captures a browser alert box with the message "xss", confirming that the injected JavaScript was successfully processed and executed by the browser. This indicates that

OWASP Juice Shop failed to properly sanitize input reflected in the DOM, making it vulnerable to client-side script injection.

2.3.2.2 Suggested remediation

1. Input Validation: Ensure all user inputs are validated on both the client and server side.
2. Output Encoding: Properly encode outputs when displaying user data in HTML, JavaScript, or URLs.
3. Content Security Policy (CSP): Implement a strong CSP to restrict execution of untrusted scripts.
4. Use Trusted Libraries: Use frameworks that automatically handle XSS protection, like React or Angular.
5. Avoid Dangerous APIs: Refrain from using innerHTML, document.write, or similar APIs that directly inject user input into the DOM.
6. Sanitize Inputs: Use libraries such as DOMPurify to clean HTML input before rendering.

2.3.3 Authentication Bypass

Severity Level: CVSS 8.4 (High)

Authentication bypass is a type of security vulnerability that allows an attacker to gain unauthorized access to a system by circumventing the normal authentication mechanisms. In this task, we aim to exploit an authentication bypass vulnerability in OWASP Juice Shop by performing a union-based SQL injection to extract sensitive data such as the totpSecret (used for 2FA), and then use it to impersonate another user (wurstbrot). This attack demonstrates how weak input validation and improper handling of authentication tokens can be abused to gain unauthorized access.

Exploitation Goals:

- Inject SQL payload to extract the totpSecret from the database.
- Retrieve the target user's email (wurstbrot).
- Bypass standard login and generate a valid 2FA code using the extracted secret.
- Successfully log in as the target user without knowing their password.

2.3.3.1 Steps to Reproduce and Exploitation Process

Step 1: In the OWASP Juice Shop product search bar, type apple to generate a product related request.

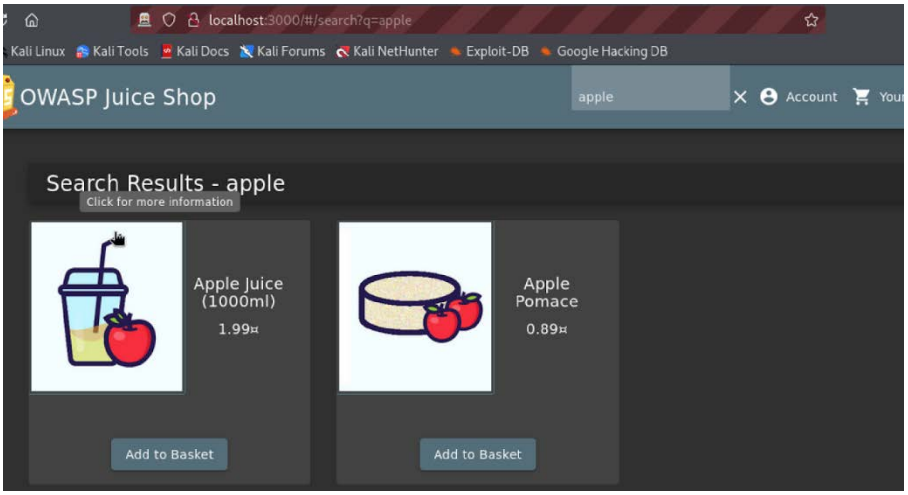


Figure 37: Displayed results for 'apple' search.

Step 3: Open OWASP ZAP, and locate the captured request:

- GET /rest/products/search?q=apple.
- Right-click the request and choose Open in Requester Tab

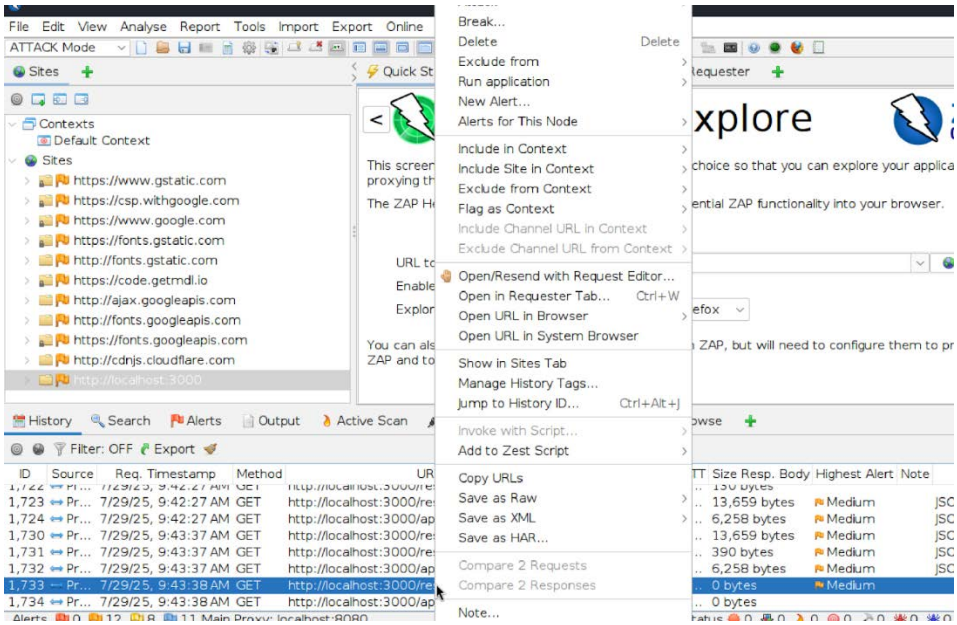


Figure 38: Captured GET request for the product search in ZAP.

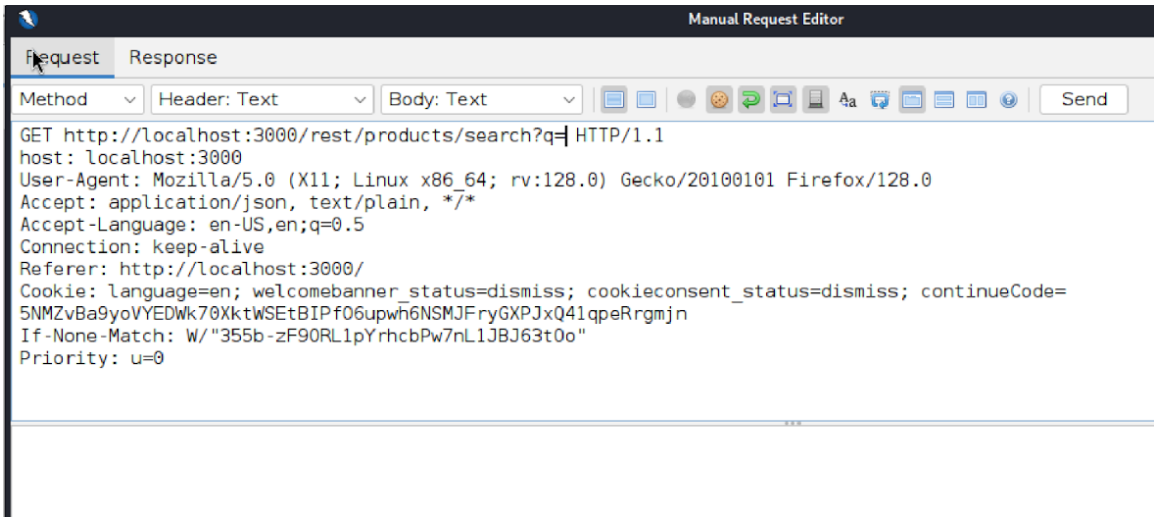


Figure 39: Request tab for the product search in ZAP.

Step 4: Paste the SQL command

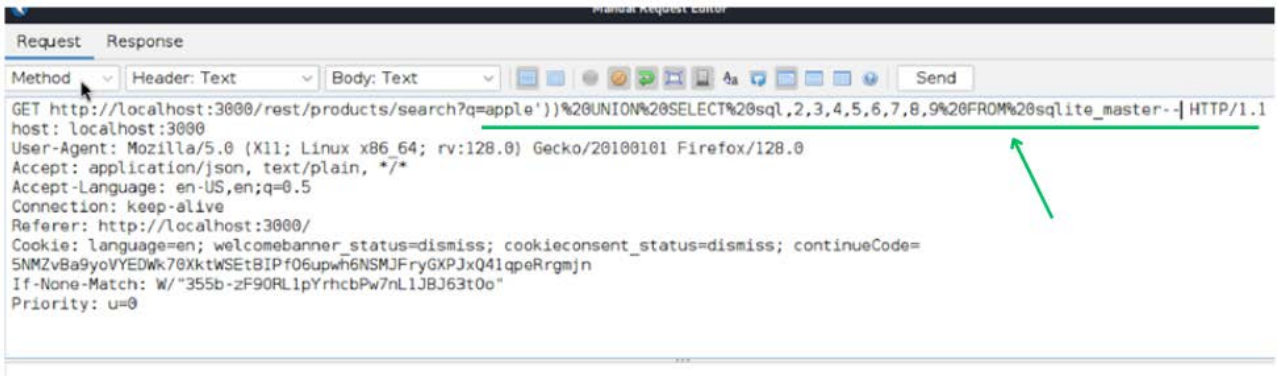


Figure 40: Injecting SQL command in the request header.

Figure 40 illustrates a manual SQL Injection attack performed by injecting a malicious SQL query into the q parameter:

- apple'))%20UNION%20SELECT%20sql,2,3,4,5,6,7,8,9%20FROM%20sqlite_master--.

This union-based SQL injection attempts to extract schema information from the sqlite_master table, which stores metadata about the SQLite database. The crafted payload manipulates the query structure and uses the UNION SELECT statement to retrieve internal SQL data, helping the attacker uncover database details such as table and column names as seen in ***Figure 41***.

Step 5: After sending the response, find for the ‘totpSecret’ in the response tab

Figure 41: Extracted totpSecret from the response.

Step 6: Change the header again with now the ‘totpSecret’

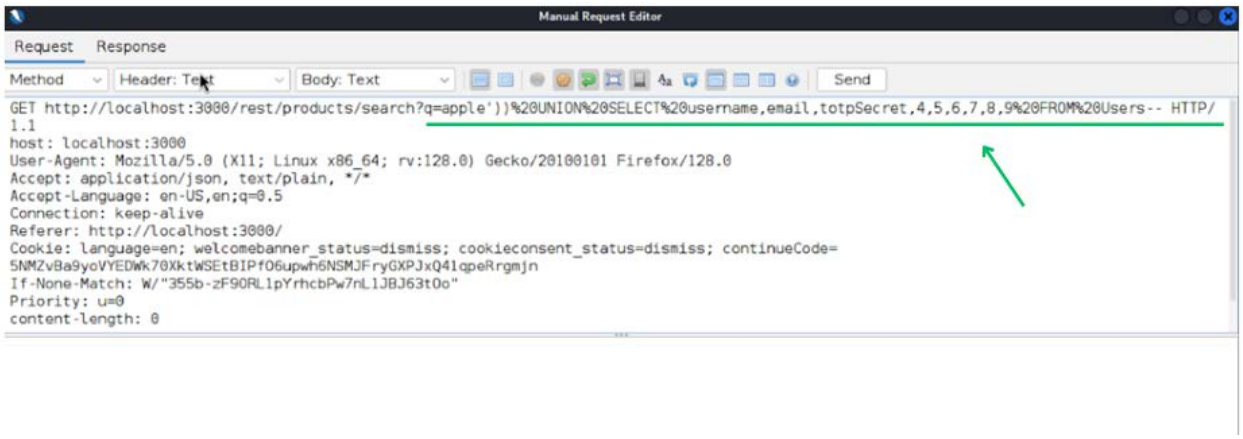


Figure 42: Updated request with totpSecret.

Step 7: find for the ‘wurstbrot’ in response tab and get the wurstbrot email address

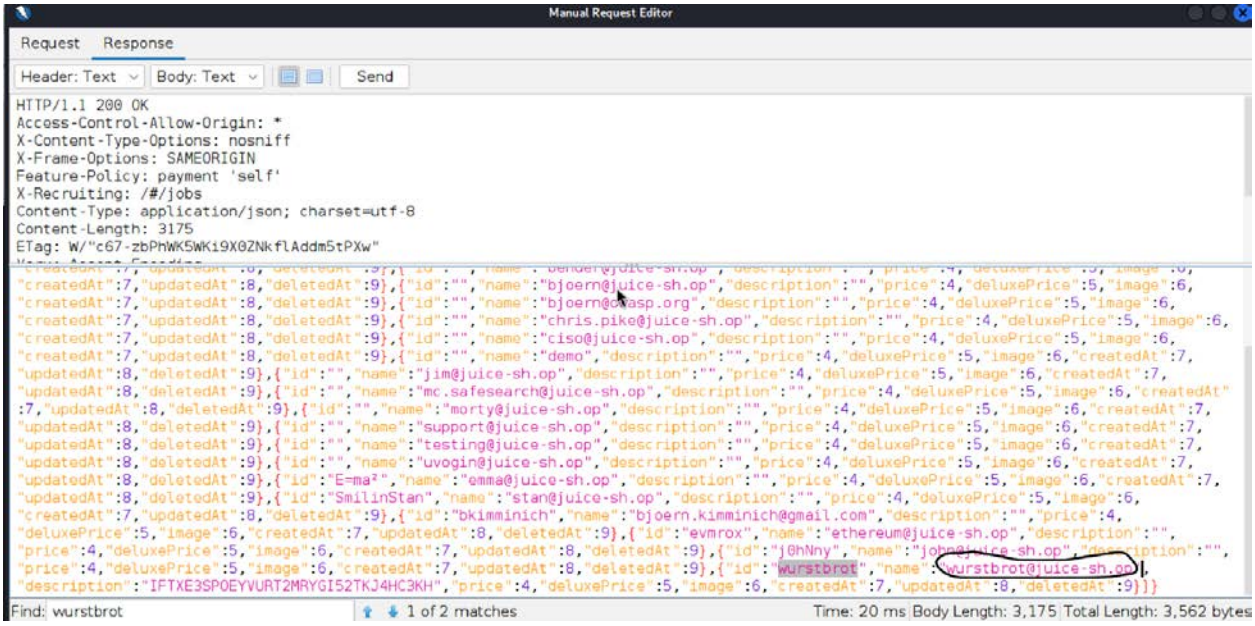


Figure 43: Response reveals wurstbrot's email address

Step 8: Go back to OWASP Juice shop and attempt login with incorrect details

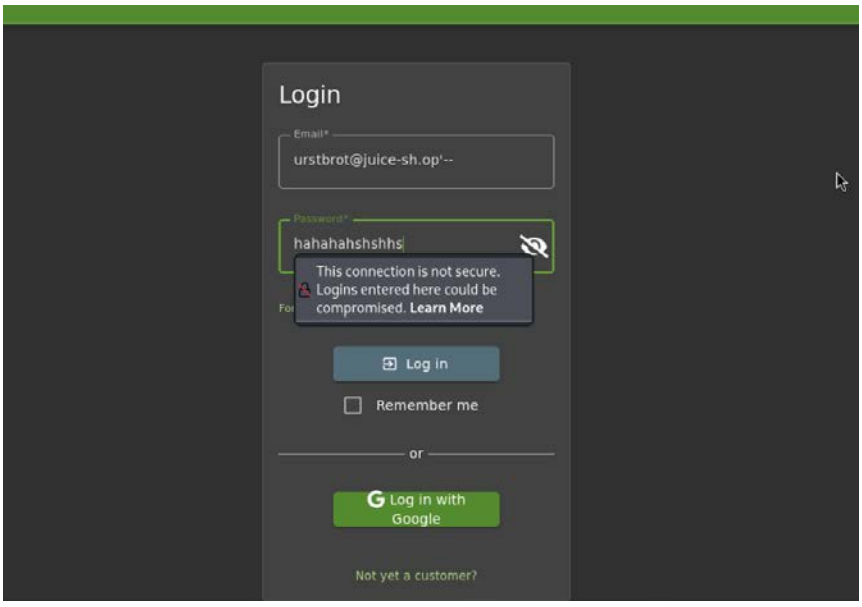


Figure 44: Attempted login with incorrect credentials.

Step 9: Now log in using the correct wurstbrot email address.

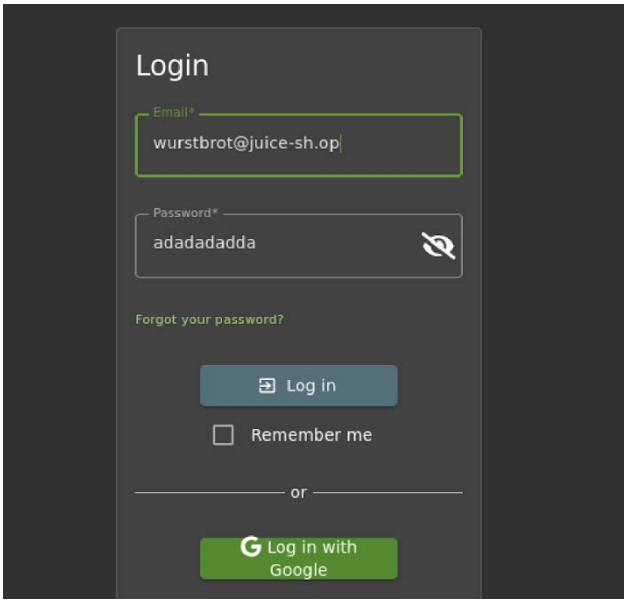


Figure 45: Attempt login with correct wursbrot email address

Step 10: Return to ZAP and locate the previous login response that includes the 2FA totpSecret.

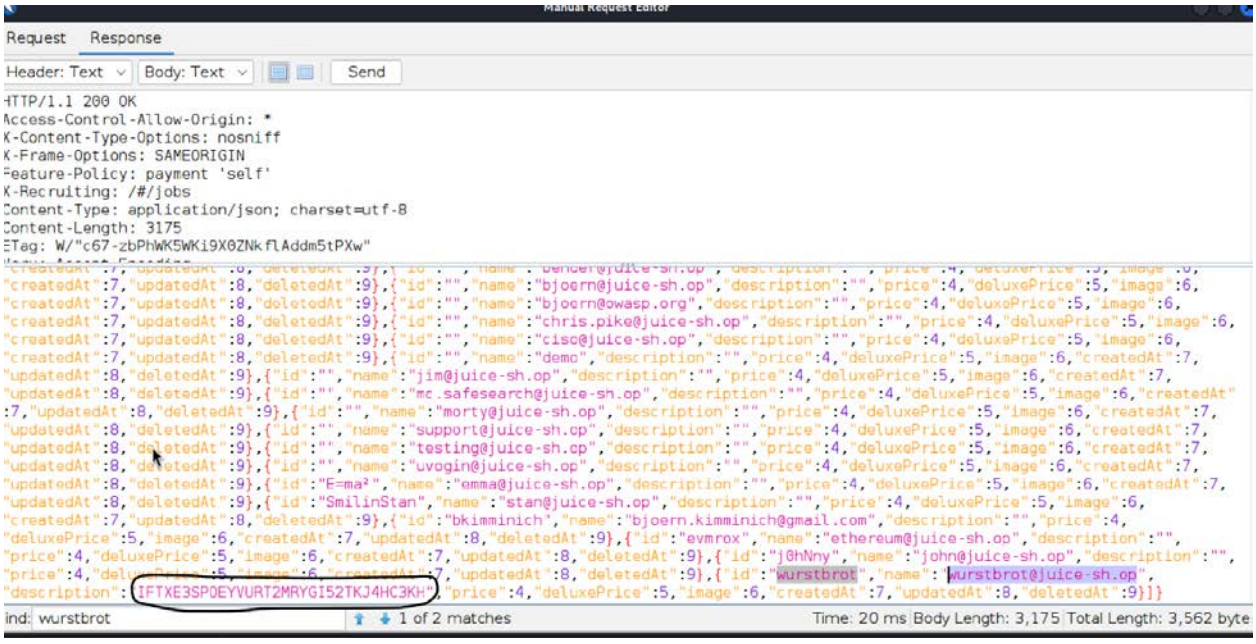


Figure 46: Response reveals wurstbrot's 2FA

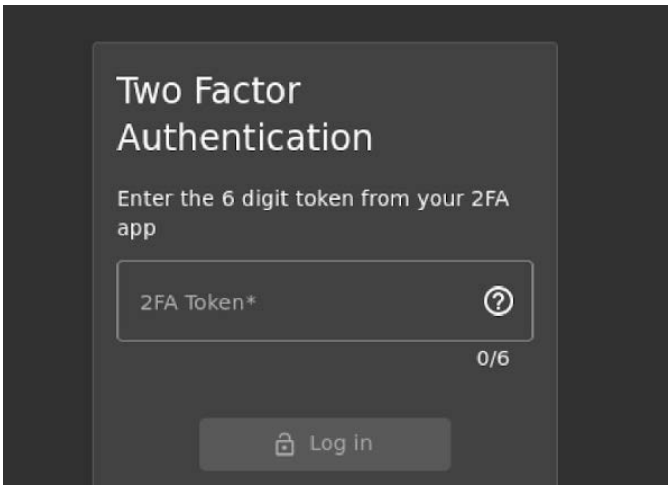


Figure 47: 2FA code will be asked

Step 11: Use google authenticator to generate the code for the 2FA authentication

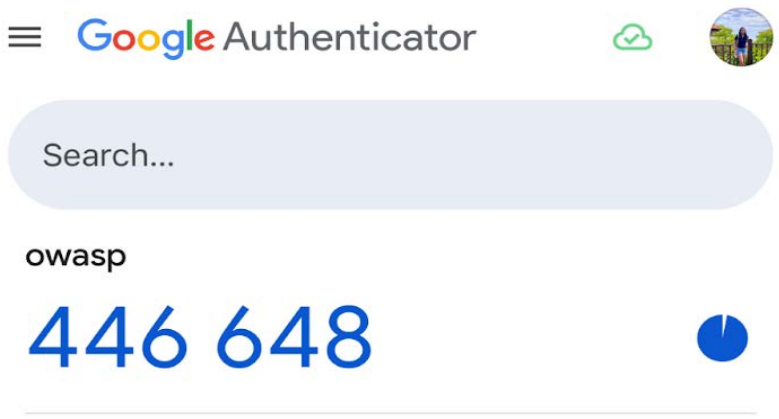


Figure 48: Generated 2FA code in google authenticator app.

Step 12: Enter the generated code in the OWASP juice-shop

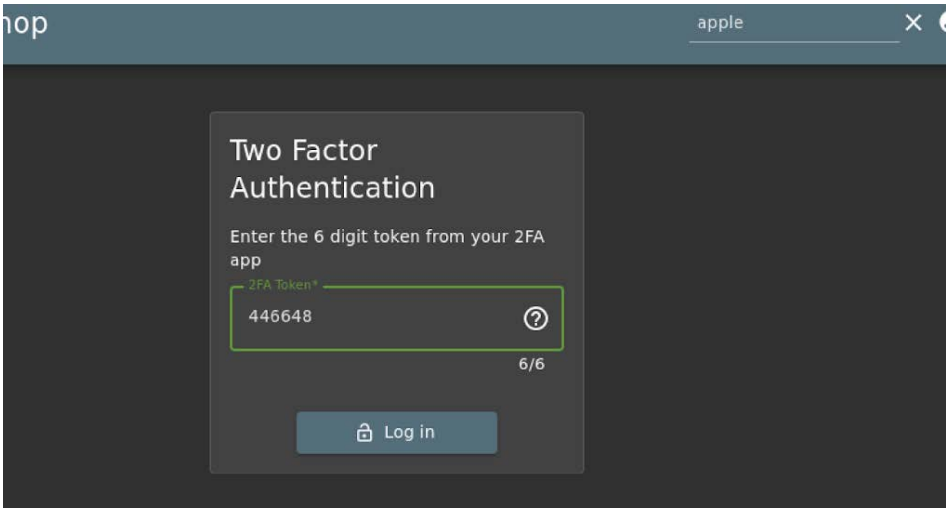


Figure 49: Submitting 2FA code for wurstbrot login

Step 13: 2FA authentication has been successfully solved

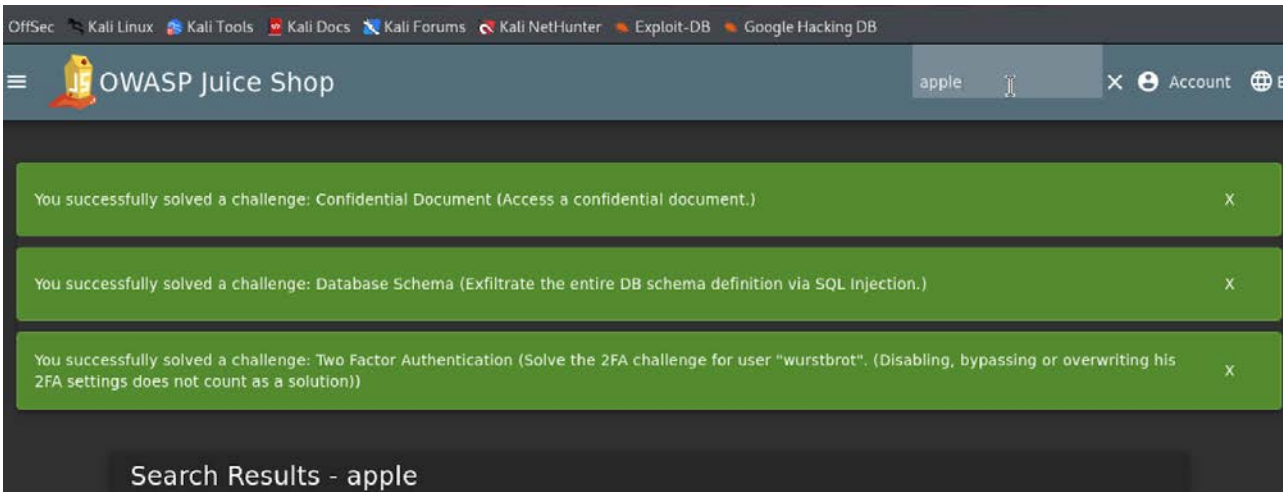


Figure 50: 2FA authentication solved and user logged in.

2.3.3.2 Suggested remediation

- Implement strict input validation and parameterized queries to prevent SQL injection.
- Ensure sensitive data like totpSecret is never exposed in client responses.
- Use rate-limiting and logging for login attempts to detect suspicious behavior.

- Enable multi-layer authentication and ensure secrets are securely stored and encrypted.
- Conduct regular security assessments and use automated tools to scan for vulnerabilities.

2.3.4 File Type Upload

Severity Level: CVSS 8.4 (High)

In OWASP Juice Shop, file uploads are restricted to certain file types like PDF and ZIP. The application performs client-side validation to ensure that uploaded files have the correct extension and type. However, during our test, we discovered that these restrictions can be bypassed using OWASP ZAP.

File upload vulnerability was exploited by first submitting a valid PDF file through the complaint form, then intercepting the request in OWASP ZAP. Using the manual request editor, the .pdf extension from the filename was removed before resending the request. The backend failed to properly validate the modified filename and still accepted the file, demonstrating that file validation was only happening on the client side. This bypass confirms a server-side weakness, as the application accepted a file that didn't meet its declared file type policy.

Exploitation Goals:

- Identify and bypass file type validation by editing the file name.
- Demonstrate that the backend accepts non-PDF files if client-side validation is bypassed.
- Successfully upload a manipulated file and complete the vulnerability challenge.

2.3.4.1 Steps to Reproduce and Exploitation Process

Step 1: Launch Zap to open the OWASP juice shop browser via Firefox

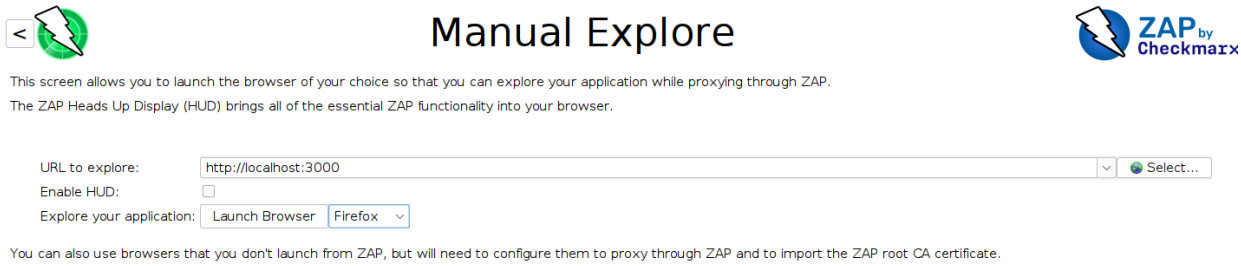


Figure 51: Initial Setup for command Injection testing using ZAP

Step 2: Log in with your account credentials.

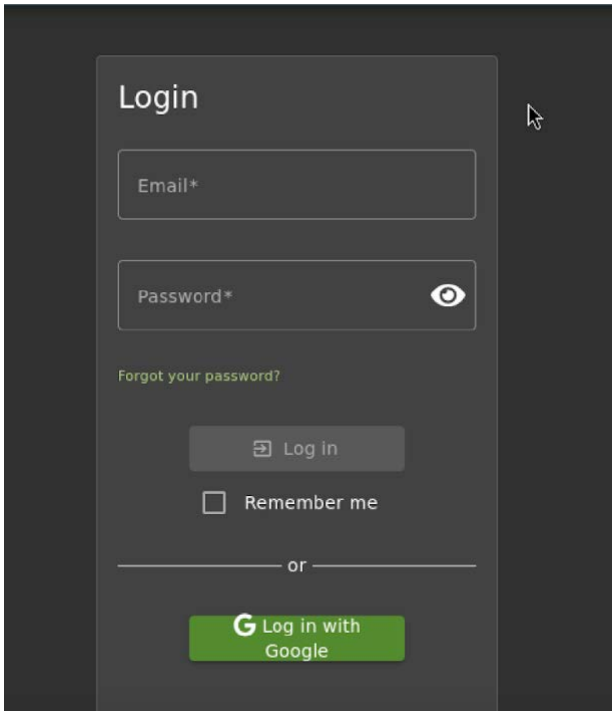


Figure 52: OWASP Juice Shop Login page

Step 3: Locate the complaint option from the side menu

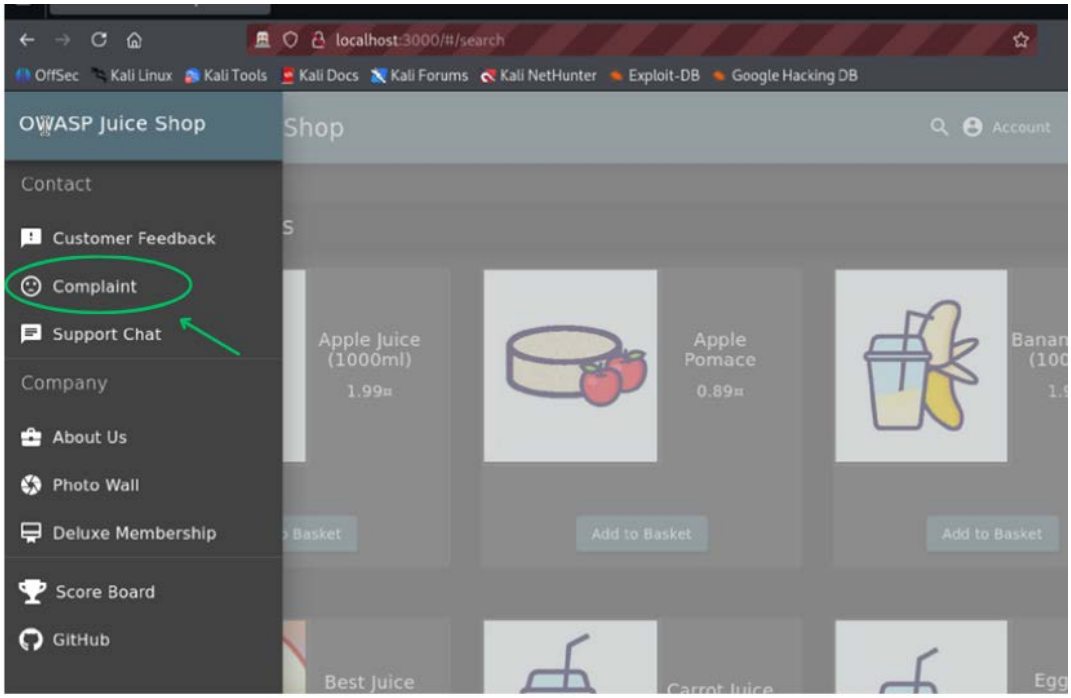


Figure 53: Side bar menu

Step 4: Write a complaint and upload a pdf document

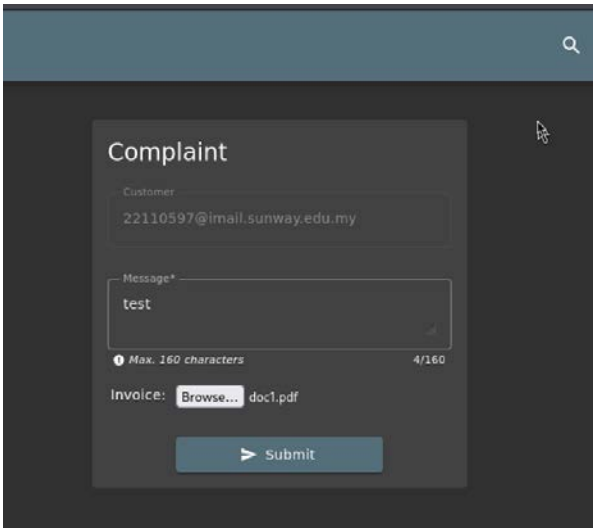


Figure 54: Complain tab

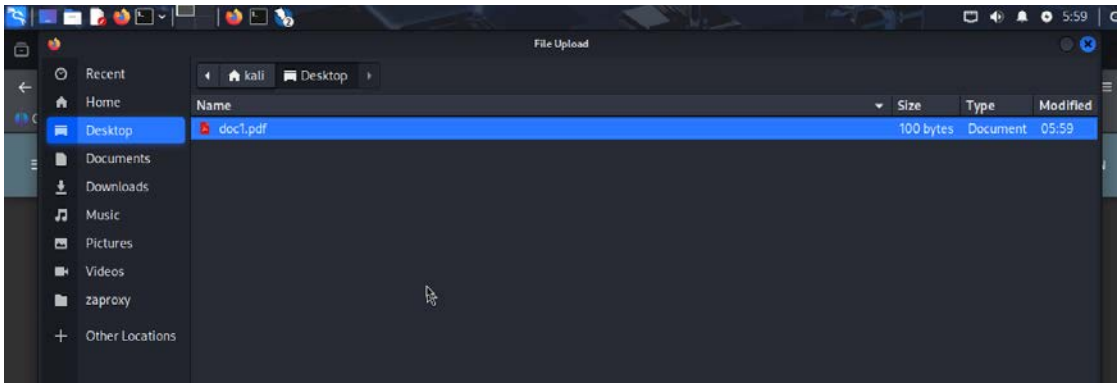


Figure 55: Uploading pdf file

Step 5: Submit the complaint and an alert will show that the complaint has been successfully sent.

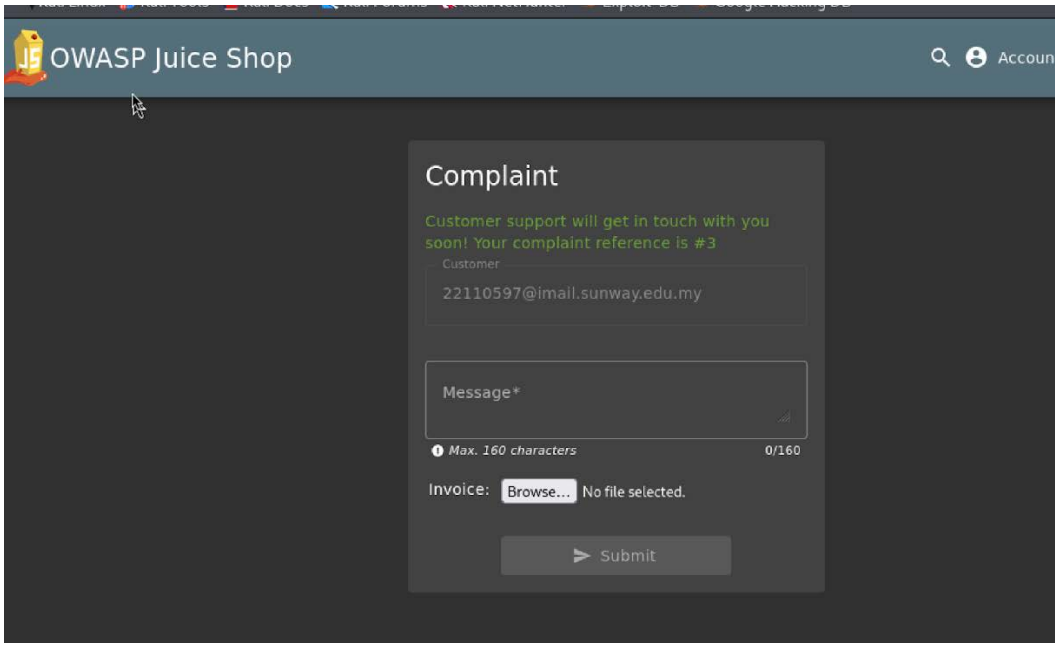


Figure 56: Successfully submitted the complaint

Step 6: In ZAP, locate the POST /file-upload request under Sites > http://localhost:3000

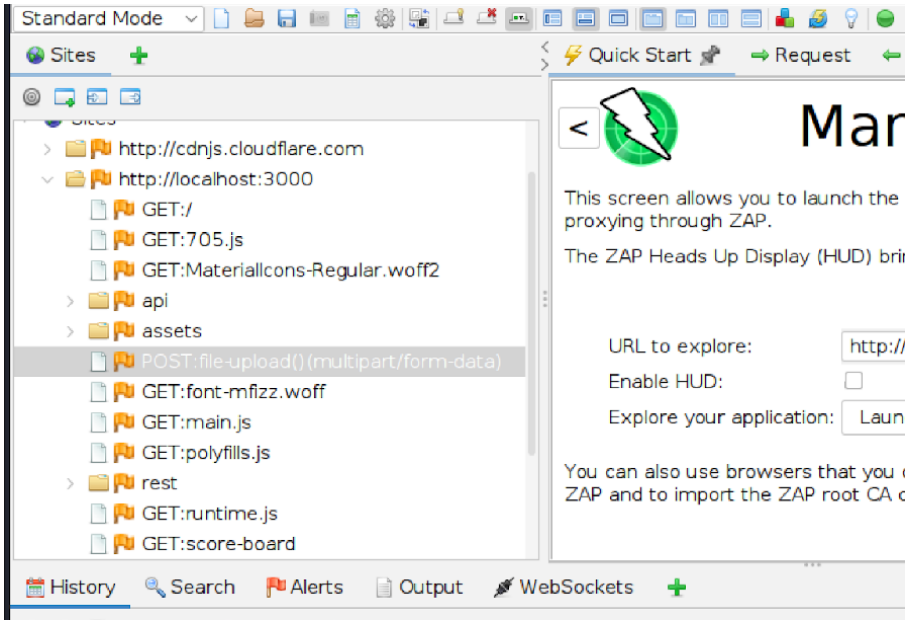


Figure 57: POST /file-upload request

Step 7: right click on the POST-file upload and select Open/Resend with Request Editor

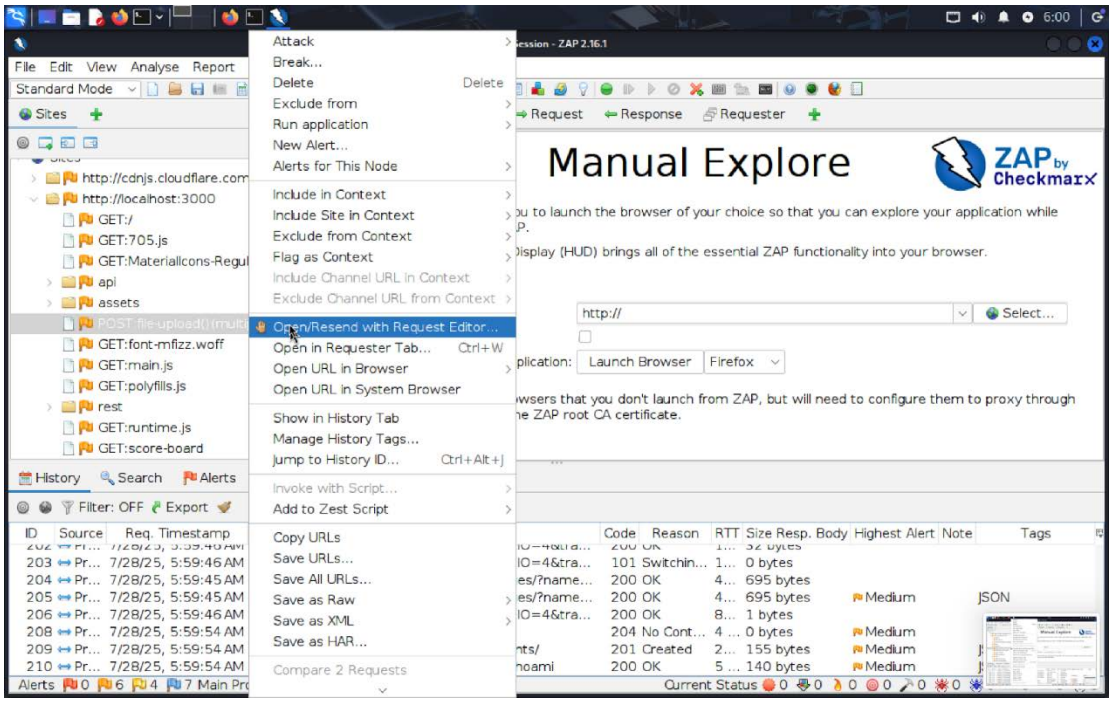


Figure 58: Selecting Open/Resend with Request Editor

After selecting Open/Resend with Request Editor, a manual request editor will be opened showing the file details

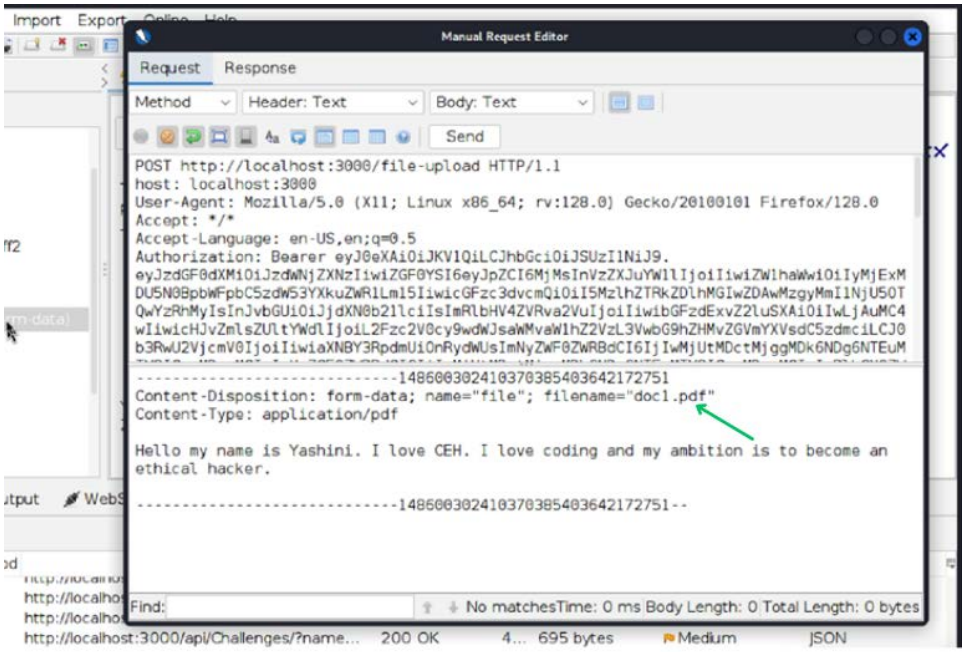


Figure 59: manual request editor tab

Step 8: Remove the pdf from the filename that has been uploaded

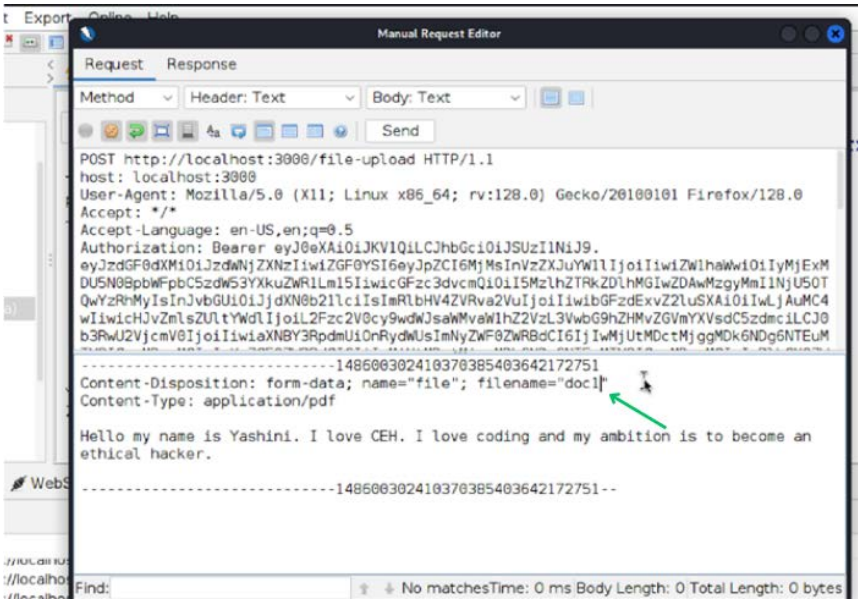


Figure 60: Removing pdf from the file name

Step 9: After editing, click Send to submit the modified request.

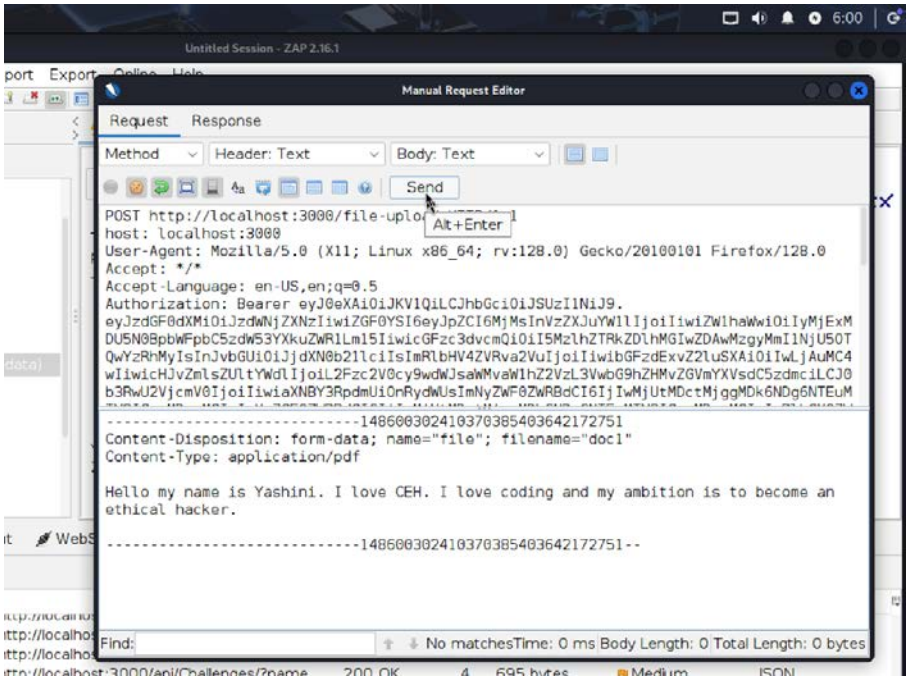


Figure 61: Submitting the modified request

Step 10: Once the request is sent successfully, it will show 200 OK

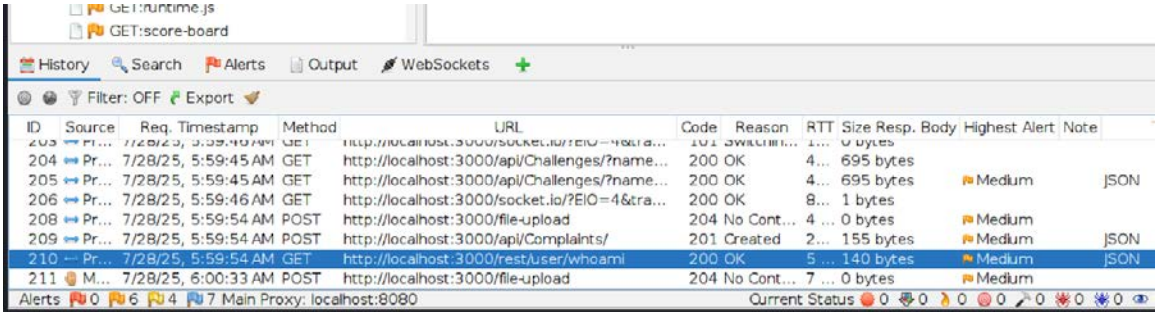


Figure 63: The response returns 200 OK

The file upload type vulnerability has been successfully solved after uploading a file without the pdf.

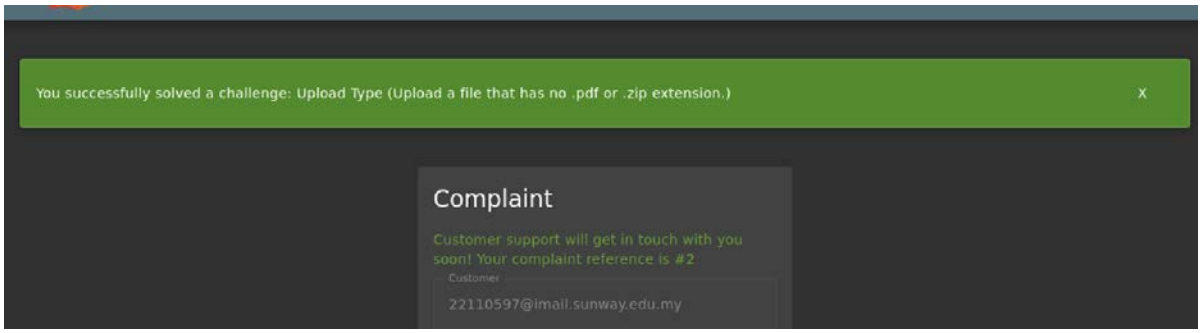


Figure 63: Successfully conducted file upload

Step 11: The terminal logs also confirm successful completion of the vulnerability in around 6 minutes.

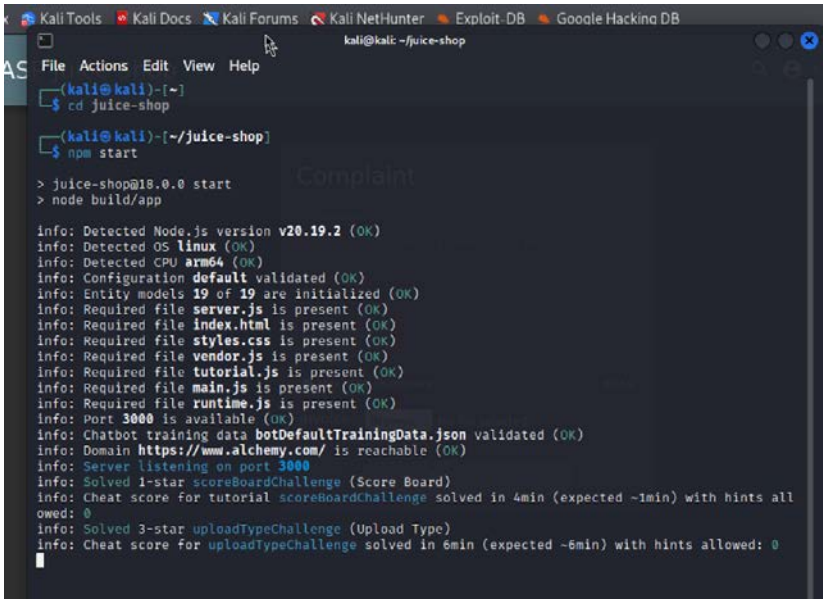


Figure 64: terminal logs

2.3.4.2 Suggested remediation

1. Enforce server-side validation of file types and MIME headers, not just extensions.
2. Check file content signatures (magic bytes) to verify the actual file type.
3. Restrict uploads to safe, expected formats (PDF/ZIP only) and limit file size.
4. Store uploaded files outside the web root and rename them using secure, random names.
5. Use content scanning tools to detect embedded threats (e.g., scripts in PDFs).

6. Provide feedback to users for rejected files and log all uploads for auditing.

2.3.5 Command Injection

Severity Level: CVSS 8.4 (High)

Command Injection is a high-severity vulnerability that occurs when an application insecurely passes user input directly to the system shell or command-line interface. This can allow attackers to execute arbitrary system commands on the server, potentially gaining access to sensitive data or compromising the entire system.

In OWASP Juice Shop, command injection will be performed in the Deluxe Membership feature. By intercepting the POST /deluxe-membership request using OWASP ZAP, manually injected the whoami command into the request payload. The backend executed this command, allowing to bypass the payment requirement and gain deluxe membership without actually paying.

Exploitation Goals:

- Bypass the payment check for deluxe membership.
- Inject and execute a system command (whoami) through the vulnerable endpoint.
- Gain elevated access (deluxe membership) without authorization.
- Demonstrate lack of input sanitization in server-side command handling.

2.3.5.1 Steps to Reproduce and Exploitation Process

Step 1: Launch OWASP juice shop via ZAP

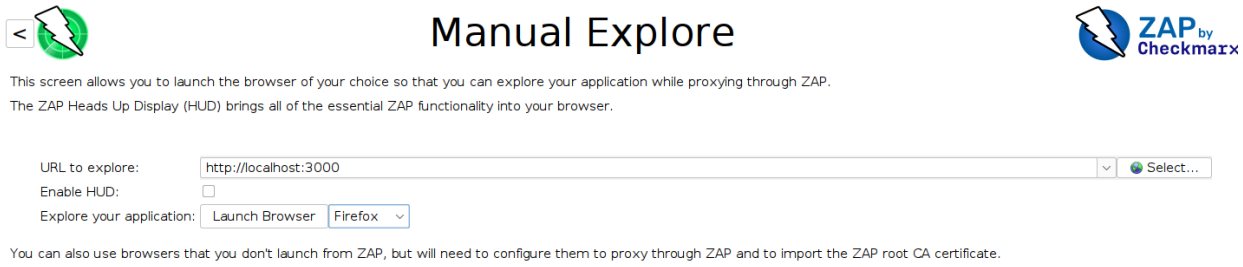


Figure 65: Initial Setup for command Injection testing using ZAP

Step 2: Access the Deluxe Membership page

- Head to the sidebar and click on deluxe membership

Figure 66: Screenshot of sidebar

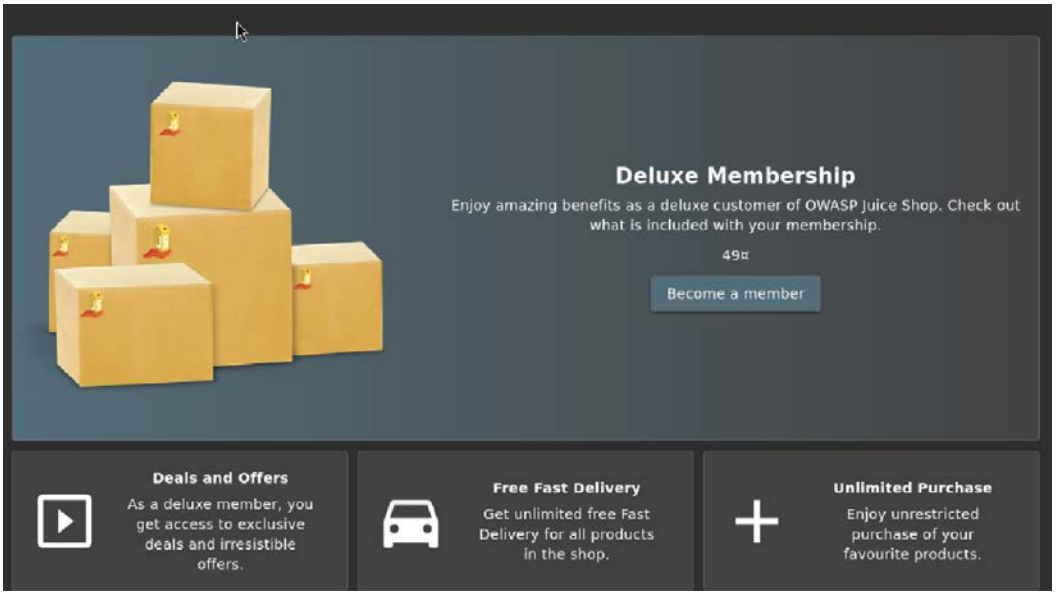


Figure 67: Screenshot of Deluxe Membership page

Step 3: Go to ZAP, under History tab find the POST deluxe-membership and right click to access the manual request editor.

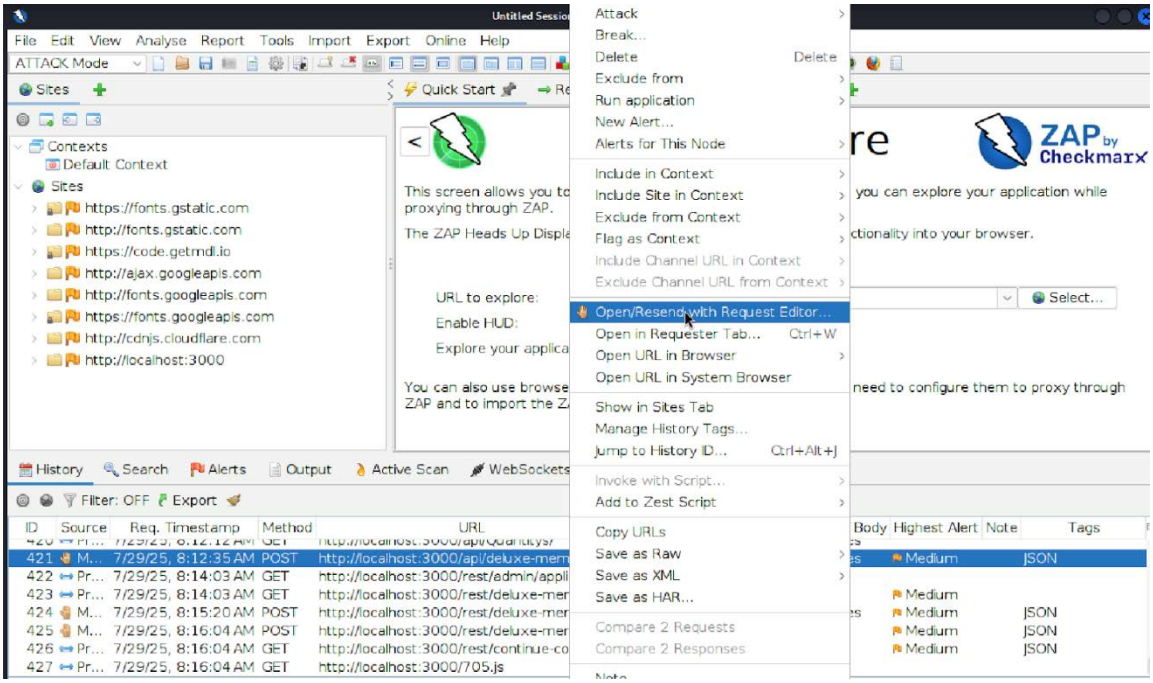


Figure 68: Selecting Open/Resend with Request Editor

Step 4: Add in the whoami command in the paylaod and send to the Requester.

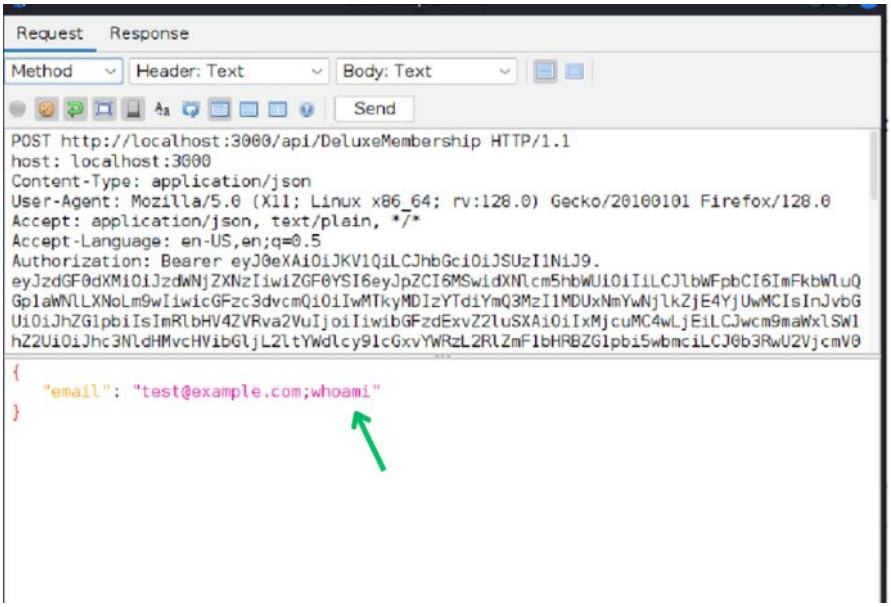


Figure 69: Injecting whoami in the payload in the request tab

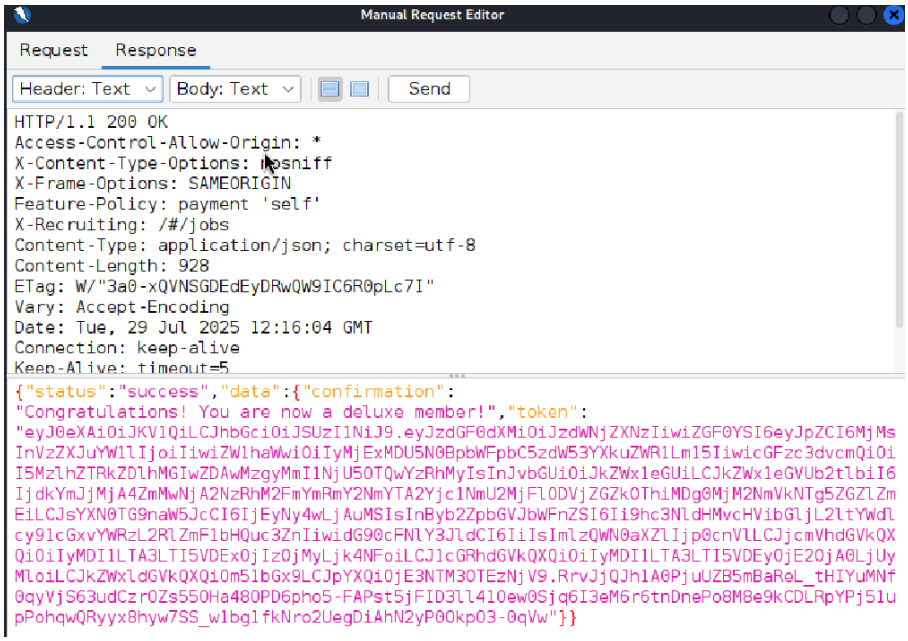


Figure 70: The response tab for showing confirmation of success.

Step 5: Check the OWASP juice-shop page to confirm if the command injection vulnerabilities has been completed.

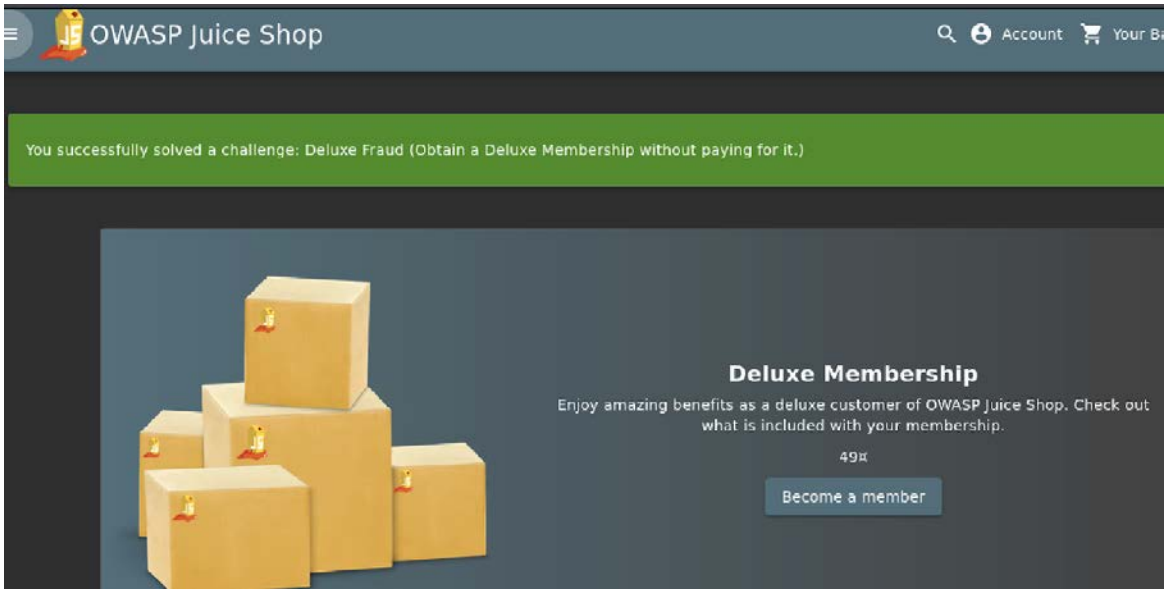


Figure 71: Screenshot of successfully conduct Command injection

2.3.5.2 Suggested remediation

1. Never pass user input directly to shell commands.
2. Use secure coding practices such as parameterized system calls (execFile in Node.js instead of exec).
3. Sanitize and validate all user inputs rigorously and reject any unexpected characters like ;, &, |, etc.
4. Run backend services with least privilege to minimize damage from command execution.
5. Monitor logs for suspicious command patterns and implement intrusion detection tools.
6. Perform regular security audits and use automated tools to scan for injection vulnerabilities.

2.3.6 Cross-Site Request Forgery (CSRF)

Severity Level: CVSS 5.3 (Medium)

Cross-Site Request Forgery (CSRF) is an attack that tricks an authenticated user into unknowingly submitting malicious requests to a web application. In OWASP Juice Shop, this vulnerability was exploited by hosting a malicious HTML form (csrf.html) on an external server. When a logged-in user accessed this page, it sent an unauthorized POST request to /profile, changing the user's username to "hacked" without their consent.

Initial attempts to launch this attack using online HTML editors like htmledit.squarefree.com failed due to browser security mechanisms like Same-Origin Policy and SameSite cookie restrictions. However, by locally hosting the HTML file using Python's http.server, we successfully bypassed these protections and demonstrated the CSRF vulnerability.

Exploitation Goals:

- Trick a logged-in user into unknowingly submitting a request.
- Bypass the UI and directly modify the user's profile information (username).
- Demonstrate that the /profile endpoint lacks adequate CSRF protection.
- Exploit trust in the authenticated session cookie to perform unauthorized actions.

2.3.6.1 Steps to Reproduce and Exploitation Process

Step 1: Craft a Local CSRF Exploit Page

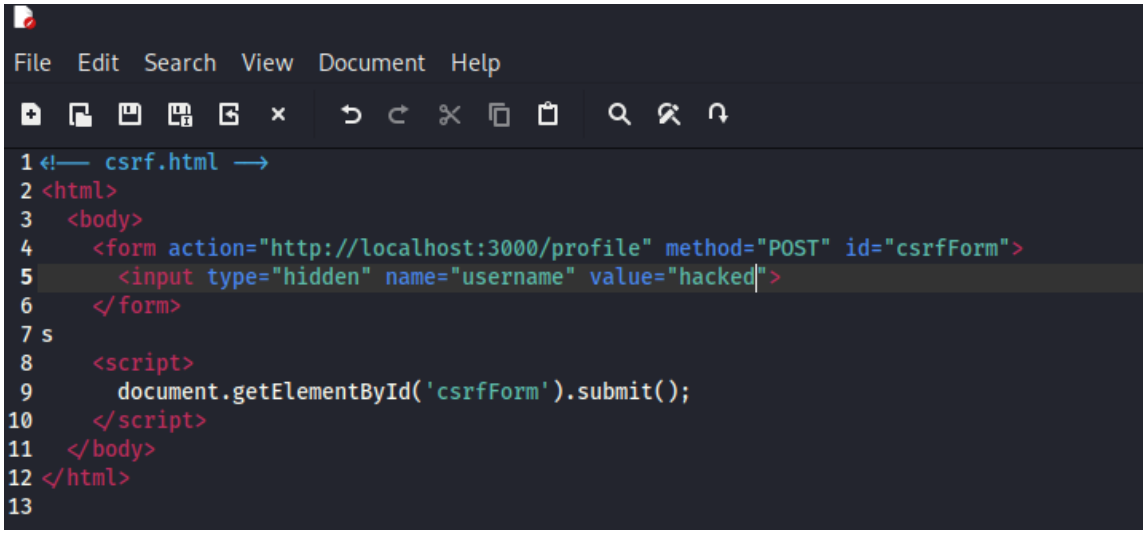


Figure 72: CSRF Exploit Code in Local HTML File

Step 2: Host the CSRF File on a Local Server

- Use Python to host the file:

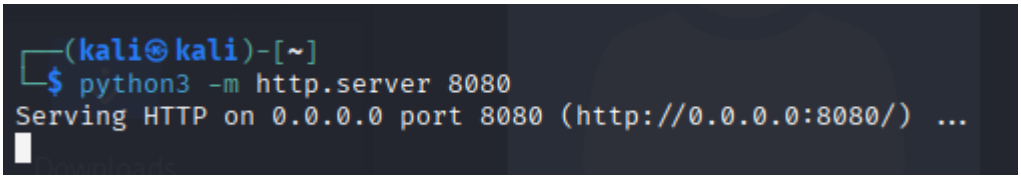


Figure 73: Hosting CSRF File with Python HTTP Server

Figure 73 demonstrates how the attacker serves the malicious file on port 8080.

Step 3: Trigger the CSRF Attack

- While logged into Juice Shop in one tab, open another tab and visit: http://localhost:8080/csrf.html
- The script silently sends the request to /profile.
- Go back to Juice Shop, navigate to the profile, and verify the username has been changed to "hacked".

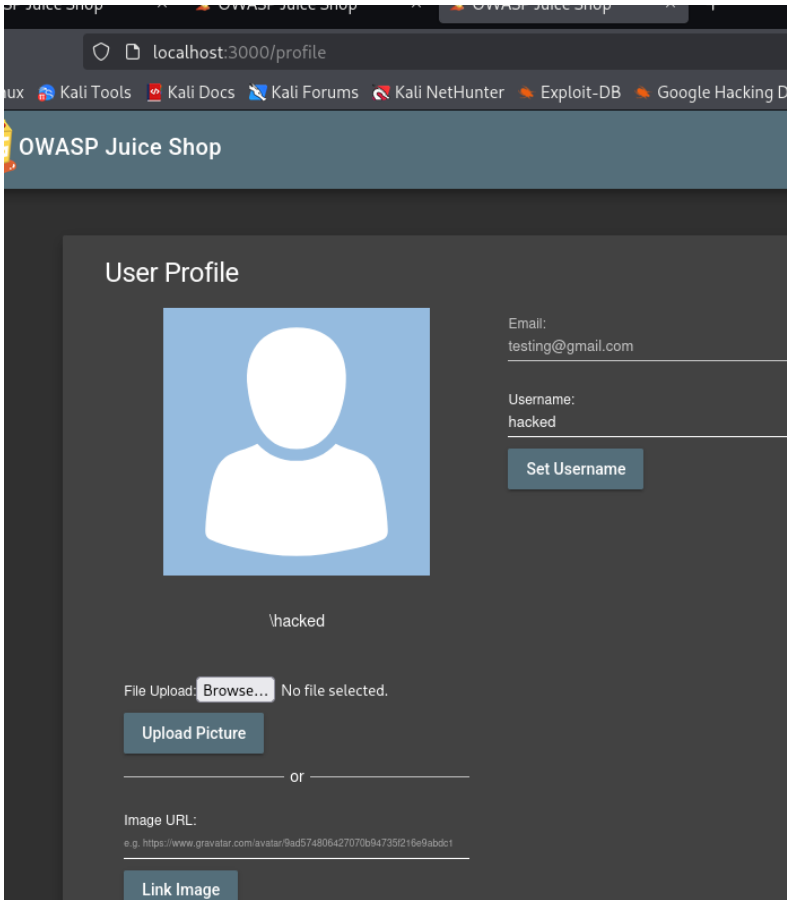


Figure 74: Username Changed Without User Consent

Confirms that the CSRF attack succeeded in altering user data.

2.3.6.2 Suggested remediation

To defend against CSRF attacks, implement the following controls:

1. CSRF Tokens:
    - Use unique, random CSRF tokens for each user session and validate them on all state-changing requests.
    - Ensure the token is embedded in forms and headers (e.g., using the X-CSRF-Token header).

2. SameSite Cookies:
    - Set the SameSite attribute on authentication cookies to Strict or Lax

3. Referer/Origin Header Checks:
    - Validate that requests originate from the same origin by checking the Referer or Origin headers.
4. X-Frame-Options Header:
    - This helps prevent clickjacking but is not a full defense against CSRF.

5. Disable CORS or Whitelist Trusted Origins Only:
    - Avoid Access-Control-Allow-Origin: * unless absolutely necessary.

6. Educate Users and Developers:
    - Conduct regular secure coding training and security awareness sessions.

2.3.7 Forged Review

Severity Level: CVSS 6.5 (Medium)

A forged review is when someone pretends to be another user and posts a fake review on their behalf, without their permission. This is a type of identity spoofing and is considered a security vulnerability because it breaks the integrity of user-generated content.

Exploitation Goal:

- The main goal of this exploitation is to forge a product review on behalf of another user, without their authorization. This undermines the trust and authenticity of user-generated content and could be used to damage reputations or manipulate public perception of products.

2.3.7.1 Steps to Reproduce and Exploitation Process

Step 1: Launch http://localhost:3000 (OWASP juice shop) via zap

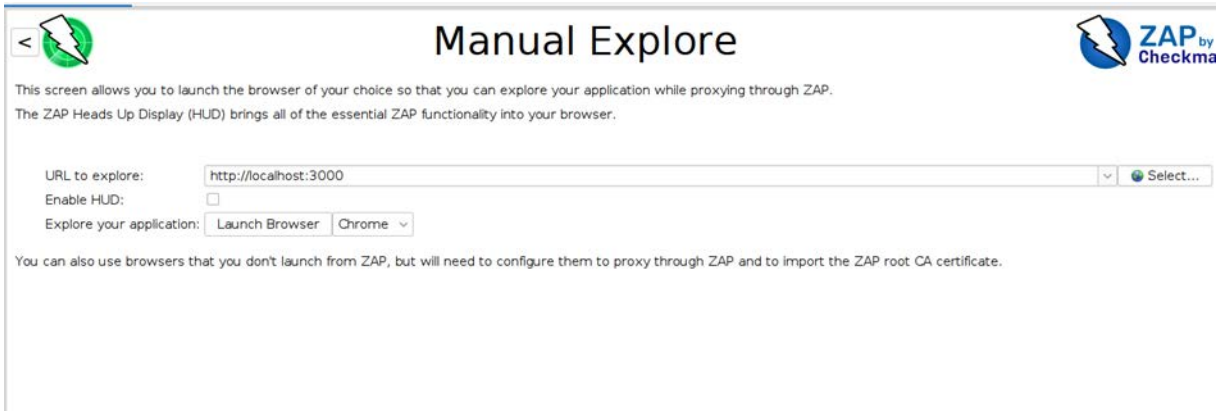


Figure 75: Launch juice shop browser via zap

Step 2: Choose a product, write a review and submit

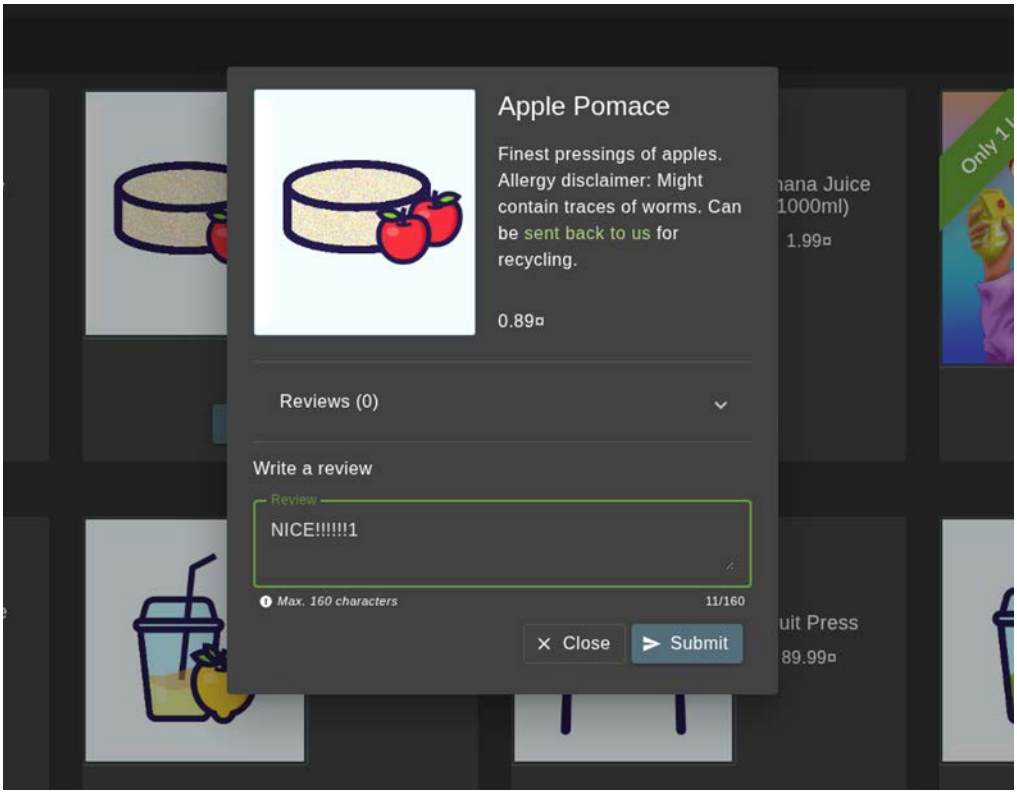


Figure 76: OWASP juice shop (product) review writing

Step 3: Return to ZAP and open the history on the PUT request to http://localhost:3000/rest/product/reviews.

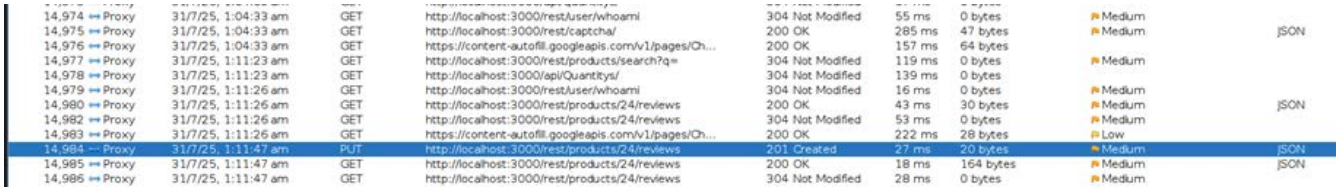


Figure 78: Zap history tab

Step 4: To open a PUT request, right-click on the request and open a request tab

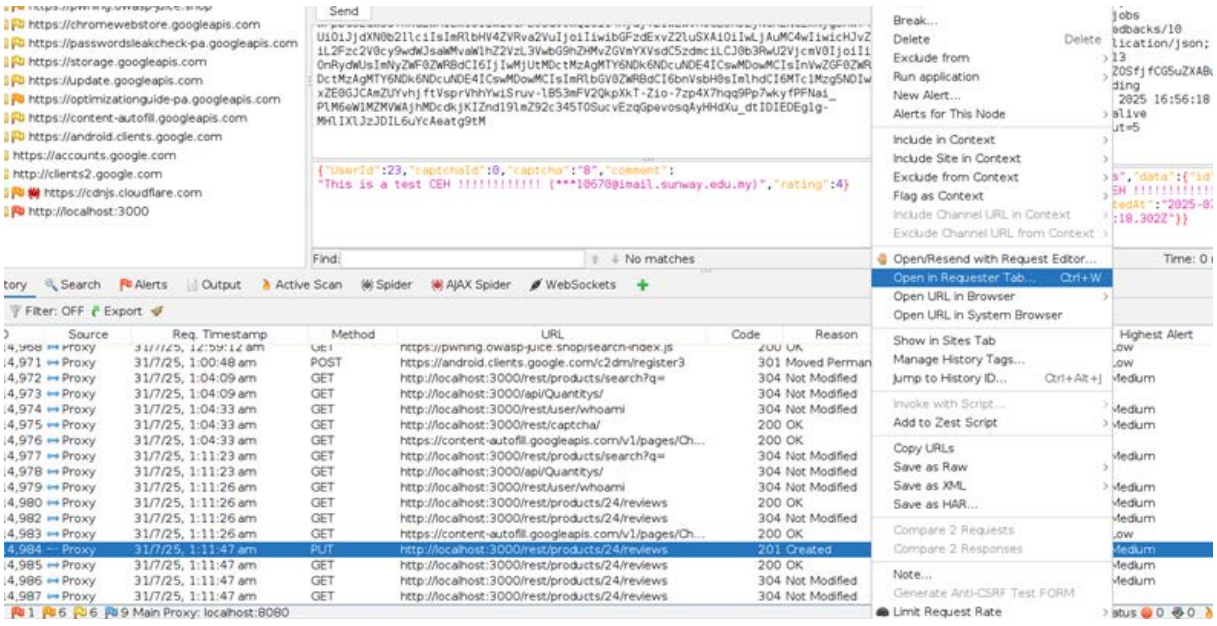


Figure 79: Opening request tab

Step 5: Alter the feedback and email to that of a different user that already exists like admin@juice-sh.op, and send.

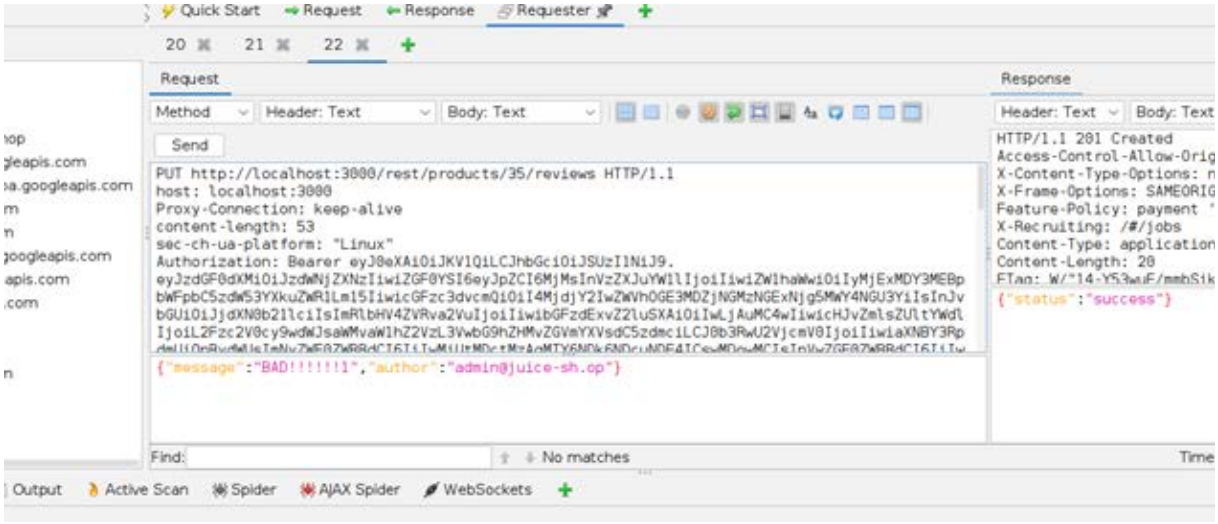


Figure 80: Request tab changing message and review author

Step 6: Challenge Forged Review has been solved

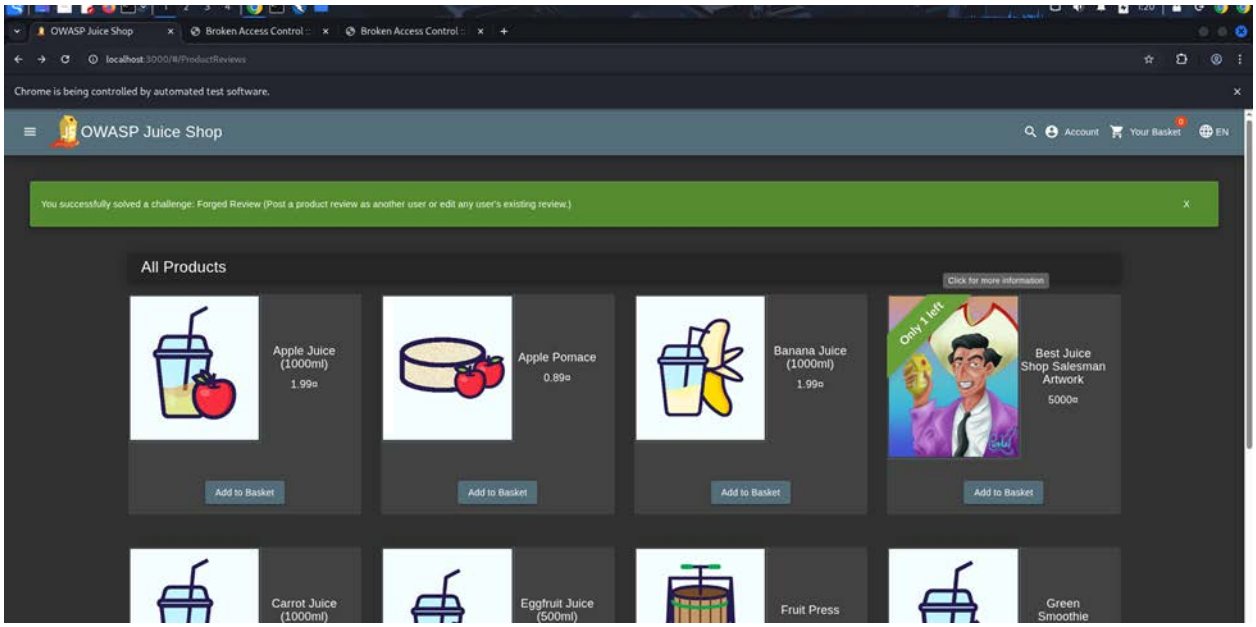


Figure 81: OWASP juice shop challenge completed

2.3.7.2 Suggested remediation

To prevent forged reviews:

1. Enforce strict server-side validation to match the logged-in user's identity with the review author field.
2. Use token-based authentication (JWT) to verify the request’s origin.
3. Remove client-controllable fields like author from the request payload.
4. Implement role-based access controls to prevent unauthorized modifications of user data.
5. Audit log activities to monitor suspicious review edits and user impersonation attempts.

2.4 Post-Exploitation

Post-Exploitation is the phase that follows an initial compromise, focusing on gaining deeper access and extracting valuable data from the system. In this step, we will perform privilege escalation by conducting a brute-force attack to obtain the credentials of a high-privileged (admin) user, allowing us to access sensitive administrative functionalities of the application. Once elevated access is achieved, we will proceed to extract sensitive information such as user credentials, personal data, and configuration files by exploring the admin panel, inspecting API responses, and exploiting vulnerabilities like directory traversal. These activities help assess the potential damage an attacker could cause after breaching the system.

2.4.1 Privilege Escalation

2.4.1.1 Brute-Force Attack

2.4.1.1.1 Steps to Reproduce and Exploitation Process

Step 1: Locate Admin Email

- Lunch OWASP Juice Shop using Zap.
- Go to the Product Review.
- Look for a review or comment that reveals the email address: admin@juice-sh.op
- Copy the admin email address.

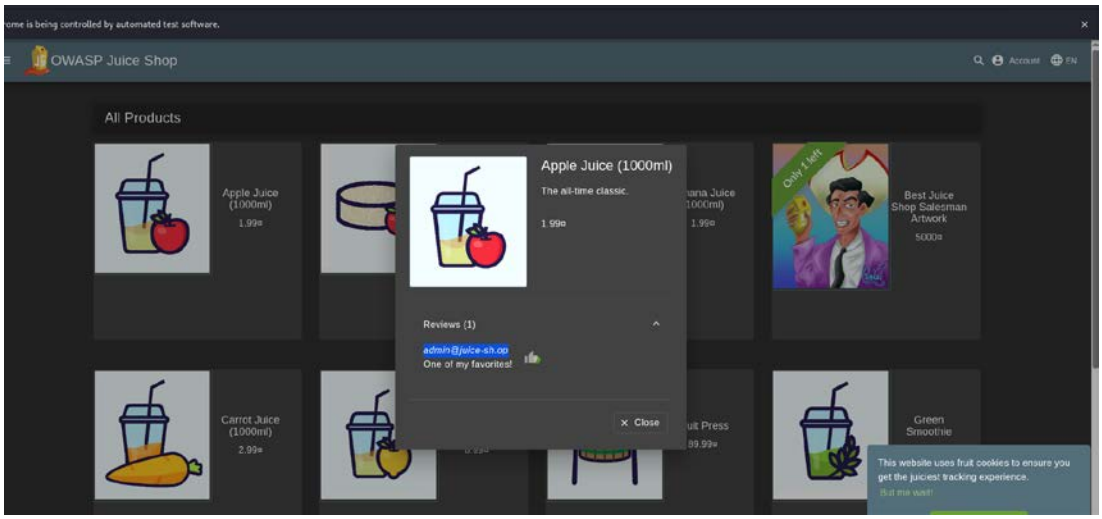


Figure 82: Admin email under product review

Step 2: Attempt to Log in with a dummy password

- Go to the Login Page.
- Enter email: admin@juice-sh.op and a dummy password: test.
- Press Login.

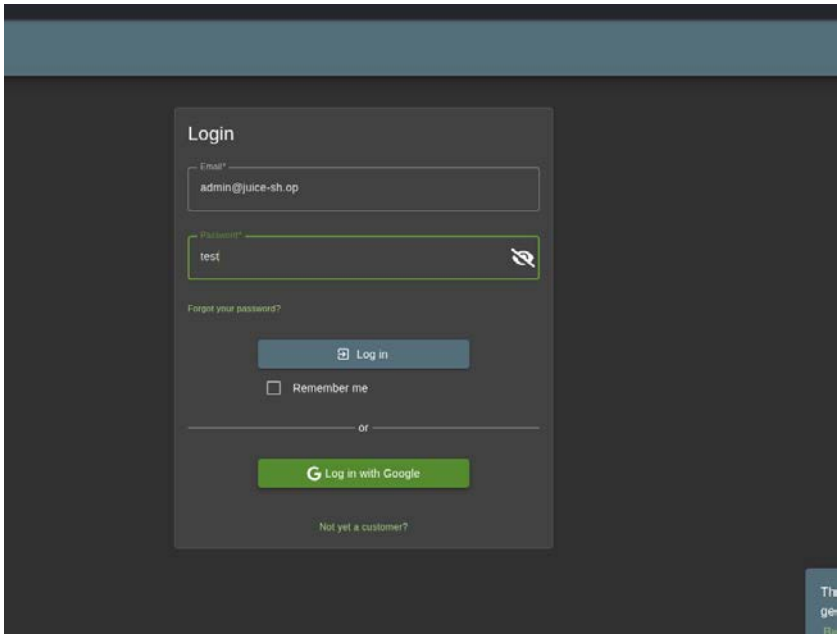


Figure 83: Attempt login with admin email

This will generate a POST request that will be captured in ZAP.

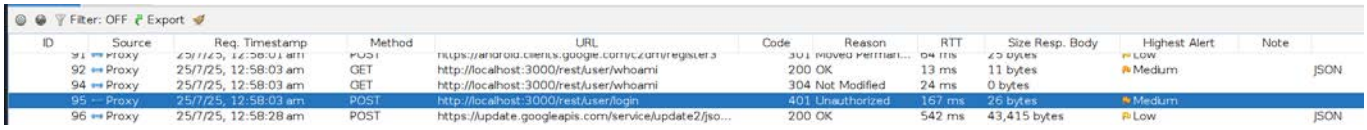


Figure 84: ZAP History – attempt login

Step 3: Set Up Fuzzing in ZAP

- Right click on the POST http://localhost:3000/rest/user/login
- Select Attack
- Then select Fuzz

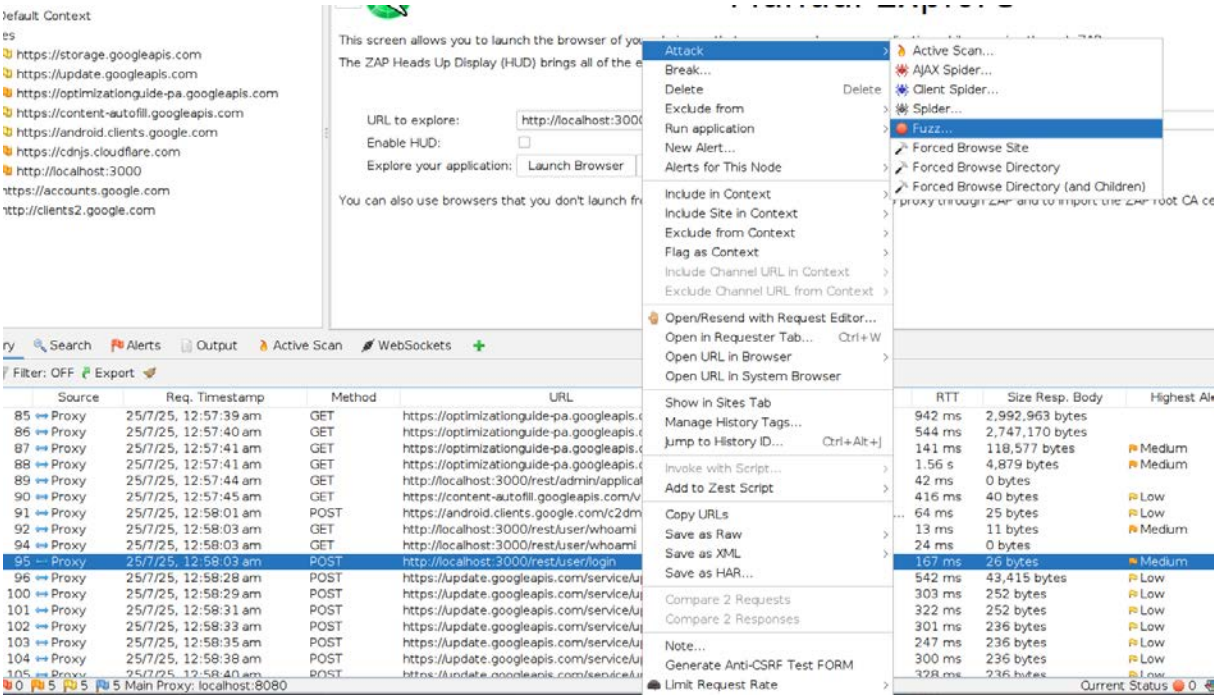


Figure 85: Launching a Fuzz Attack

Step 4: Select a Fuzzing Wordlist

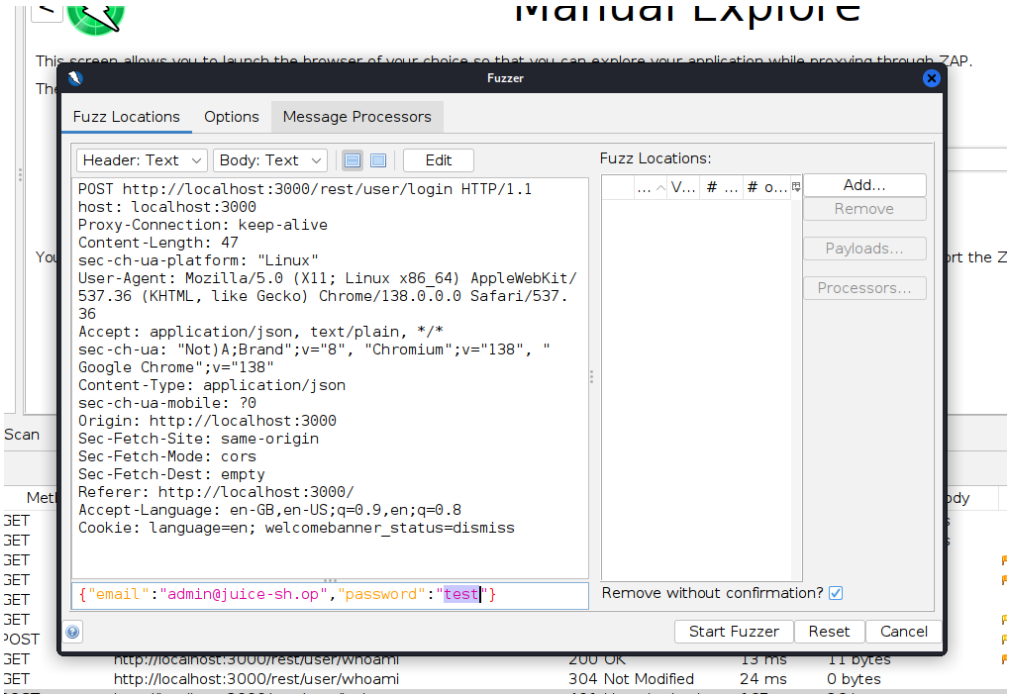


Figure 86: ZAP Fuzzer Configuration tab highlighting only the password

Figure 86 shows we have only highlighted the password parameter in an intercepted login request. This shows that the brute force attack will be only attempted to find the password of the admin@juice-sh.op

Step 5: Configured the Fuzzer to insert various password guesses from a list into that highlighted section.

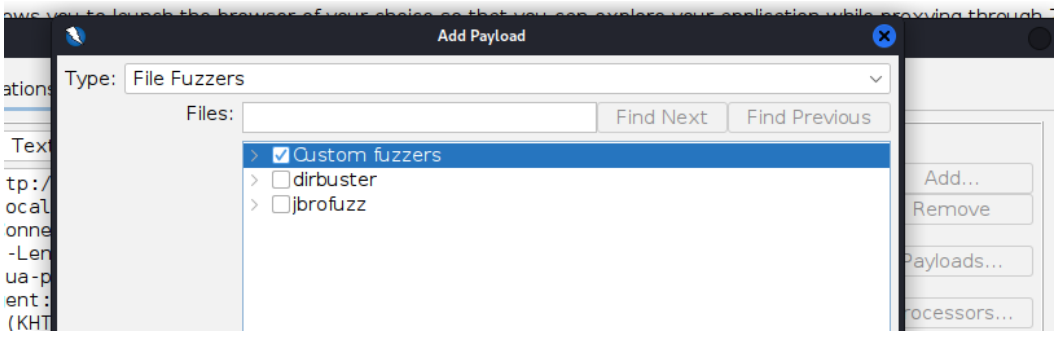


Figure 87: Adding Payloads for Fuzzing in OWASP ZAP

Figure 87 displays the "Add Payload" dialog box within OWASP ZAP's Fuzzer tool. This dialog is used to select the type of payloads that will be used during the brute-force attack. In this case "File Fuzzers" is selected as the payload type, and the option for file-based fuzzing are shown, "Custom fuzzers," is selected." This setting will allow us to configuring the fuzzer to use a list of pre-written passwords.

Step 6: Modify the payload and insert desired password lists

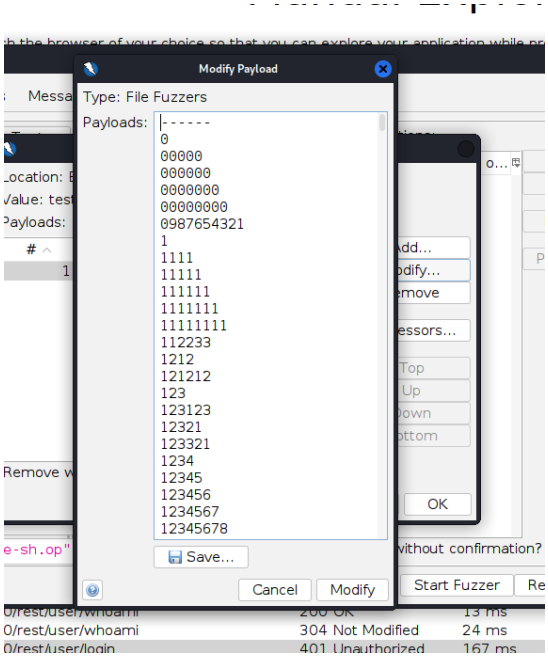
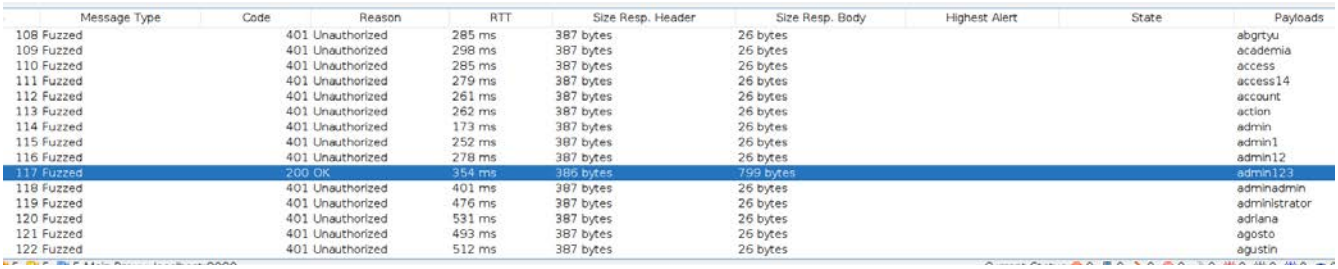


Figure 88: OWASP ZAP Modify Payload Dialog

Step 7: Lunch Fuzz attack

The Fuzzer tab then displays the results of these attempted requests as show in ***figure 77***



Message Type	Code	Reason	RTT	Size Resp. Header	Size Resp. Body	Highest Alert	State	Payloads
108 Fuzzed	401	Unauthorized	285 ms	387 bytes	26 bytes			abgrtyu
109 Fuzzed	401	Unauthorized	298 ms	387 bytes	26 bytes			academia
110 Fuzzed	401	Unauthorized	285 ms	387 bytes	26 bytes			access
111 Fuzzed	401	Unauthorized	279 ms	387 bytes	26 bytes			access14
112 Fuzzed	401	Unauthorized	261 ms	387 bytes	26 bytes			account
113 Fuzzed	401	Unauthorized	262 ms	387 bytes	26 bytes			action
114 Fuzzed	401	Unauthorized	173 ms	387 bytes	26 bytes			admin
115 Fuzzed	401	Unauthorized	252 ms	387 bytes	26 bytes			admin1
116 Fuzzed	401	Unauthorized	278 ms	387 bytes	26 bytes			admin12
117 Fuzzed	200	OK	354 ms	386 bytes	799 bytes			admin123
118 Fuzzed	401	Unauthorized	401 ms	387 bytes	26 bytes			adminadmin
119 Fuzzed	401	Unauthorized	476 ms	387 bytes	26 bytes			administrator
120 Fuzzed	401	Unauthorized	531 ms	387 bytes	26 bytes			adriana
121 Fuzzed	401	Unauthorized	493 ms	387 bytes	26 bytes			agosto
122 Fuzzed	401	Unauthorized	512 ms	387 bytes	26 bytes			agustin

Figure 89: Successfully identified the admin password using Brute force attack

Step 8: Attempt login again as admin user

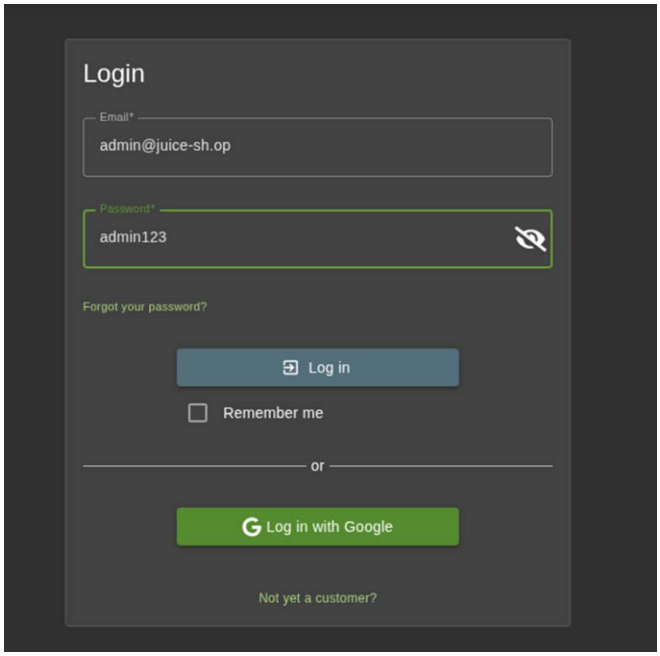


Figure 90: login page with admin password and email

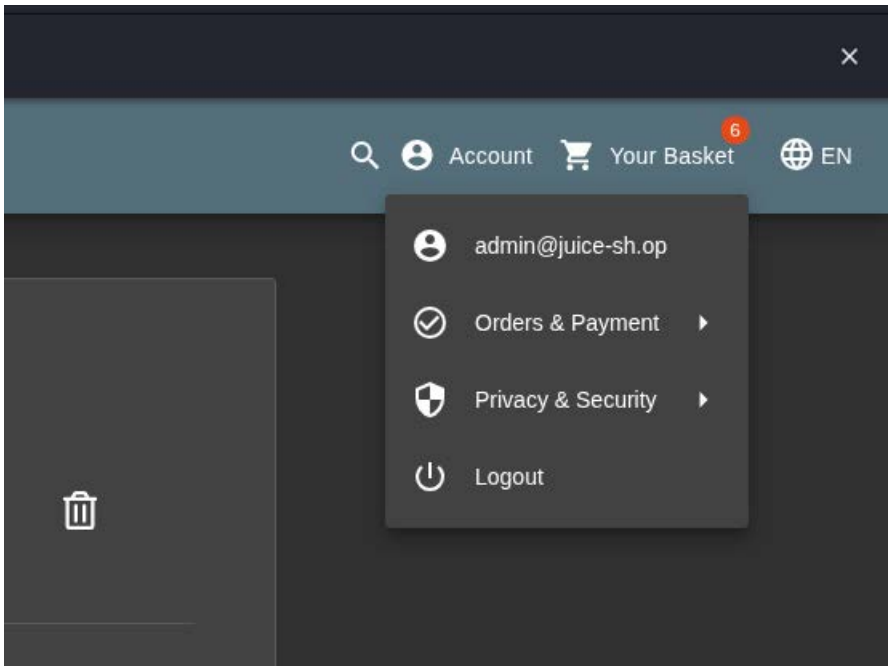


Figure 91: Successfully login as admin user

Step 9: Access Administration page with the url: http://localhost:3000/administration

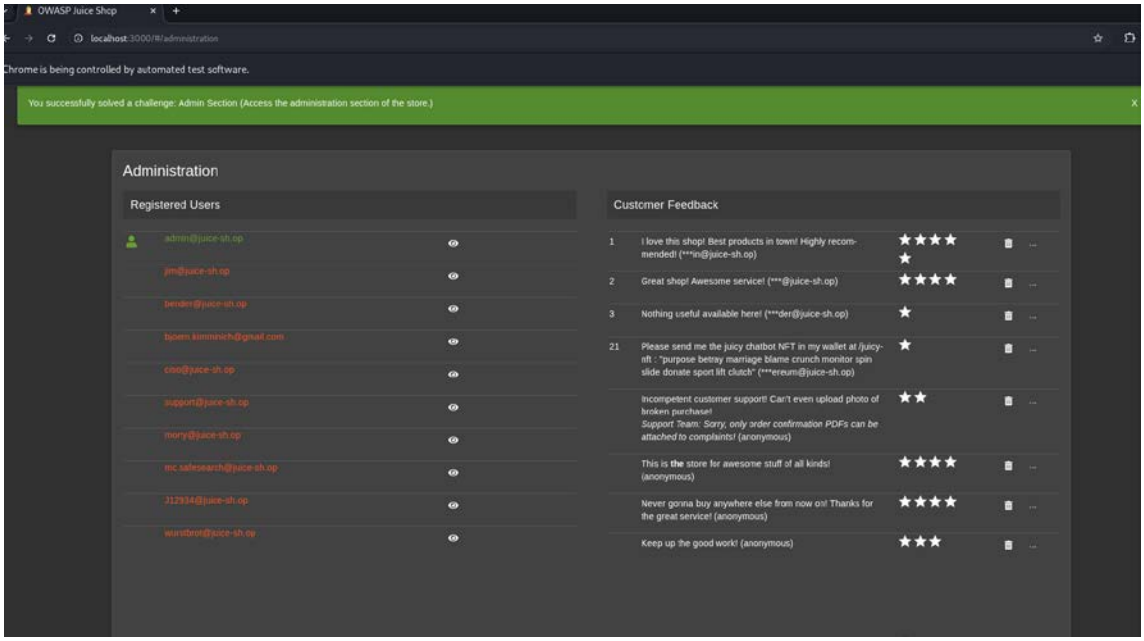


Figure 92: Access administration page

2.4.1.1.2 Suggested remediation

1. Use CAPTCHA
    - Add CAPTCHA or reCAPTCHA to login forms to block bots and automated tools.
2. Enforce Strong Password Policies
    - Require users to use complex passwords (minimum length, mix of characters).
    - Reject commonly used or breached passwords.
3. Enable Multi-Factor Authentication (MFA)
    - Add an extra layer of protection with OTPs, authenticator apps, or biometric verification.
4. Rate Limiting & Throttling
    - Limit the number of login attempts from a single IP or user in a given time frame.
    - Use tools like fail2ban to block abusive IPs.
5. Monitor and Alert Suspicious Login Attempts
    - Log and detect patterns like multiple failed logins from the same IP.
    - Send alerts for abnormal login behavior (login from a new device/location).
6. Use Account Verification for New Devices

- Require email or phone verification when logging in from unknown devices.

2.4.2 Extraction of Sensitive Information

This section focuses on identifying and extracting sensitive information from the OWASP Juice Shop application due to poor access control and input validation. Three key pieces of information were successfully extracted:

- Confidential Documents – Accessed through the publicly exposed /ftp directory, revealing files like acquisitions.md containing internal business plans.
- User Credentials – Retrieved by exploiting a SQL injection vulnerability in the product search bar, exposing user emails and hashed passwords.
- Access Log File – Discovered through directory fuzzing in the /support directory, revealing the access.log file which contains details about user activity and application structure.

These findings highlight serious security lapses that could lead to data breaches, reputational damage, and legal consequences if exploited in real-world scenarios.

2.4.2.1. Steps to Reproduce and Exploitation Process

2.4.2.1.1 Confidential Documents

- Goal: Access exposed internal business files stored under /ftp.

Step 1: Navigate to the About Us page of OWASP Juice Shop.

- Click on the redirect link that leads to: http://localhost:3000/ftp/legal.md

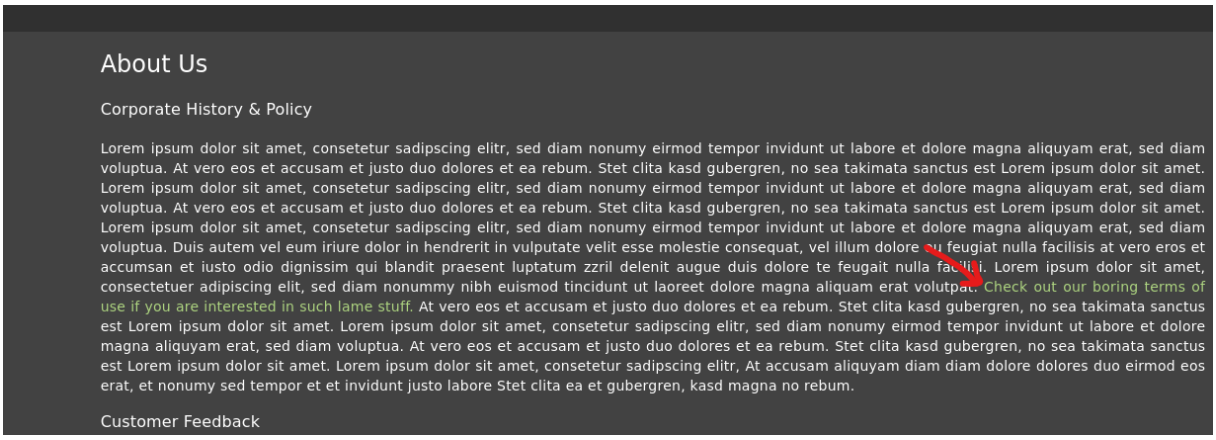


Figure 93: About Us page in OWASP Juice shop

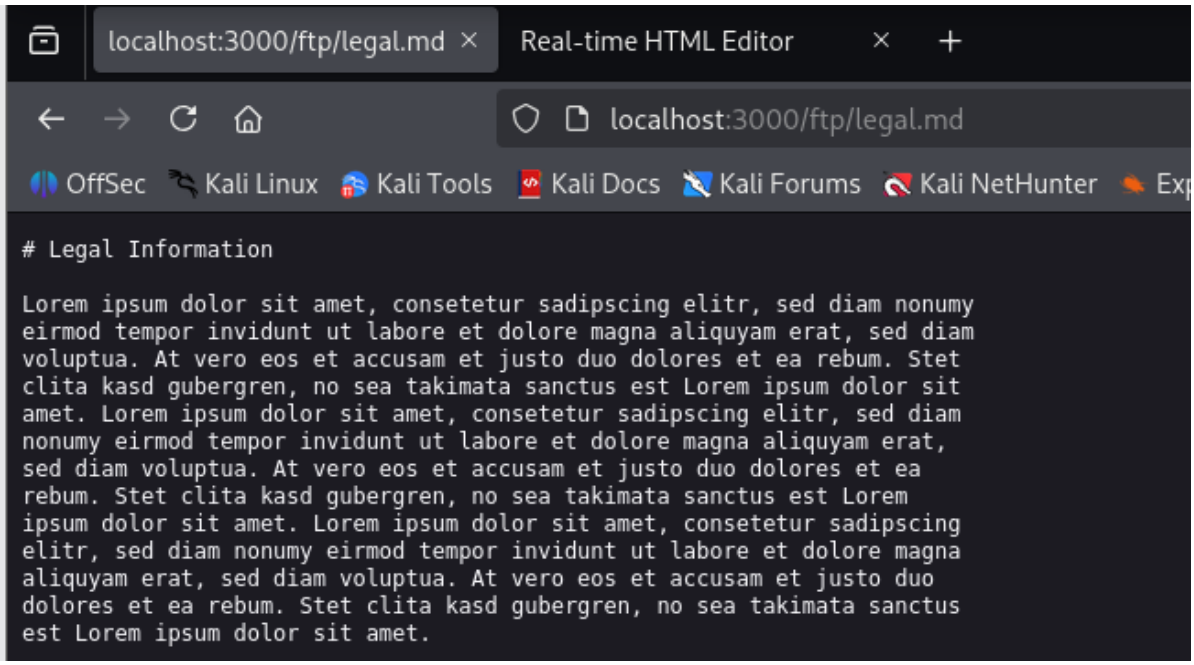


Figure 94: the /ftp/legal.md page

Step 2: Manually change the URL to: http://localhost:3000/ftp/

- The /ftp directory is openly accessible without any authentication or authorization checks. And we notice that there is a file named “acquisitions.md”.

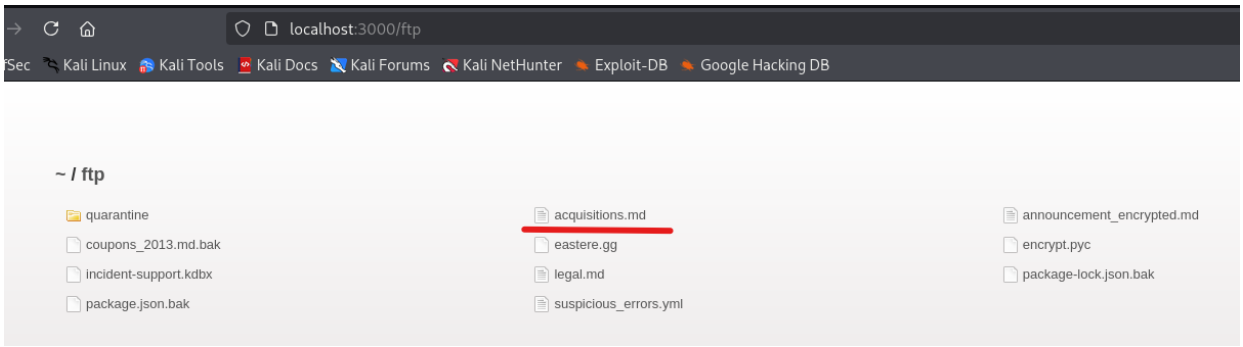


Figure 95: Inside the /ftp directory locating the acquisitions.md

Step 3: Click on acquisitions.md to read its contents, which include confidential business plans.

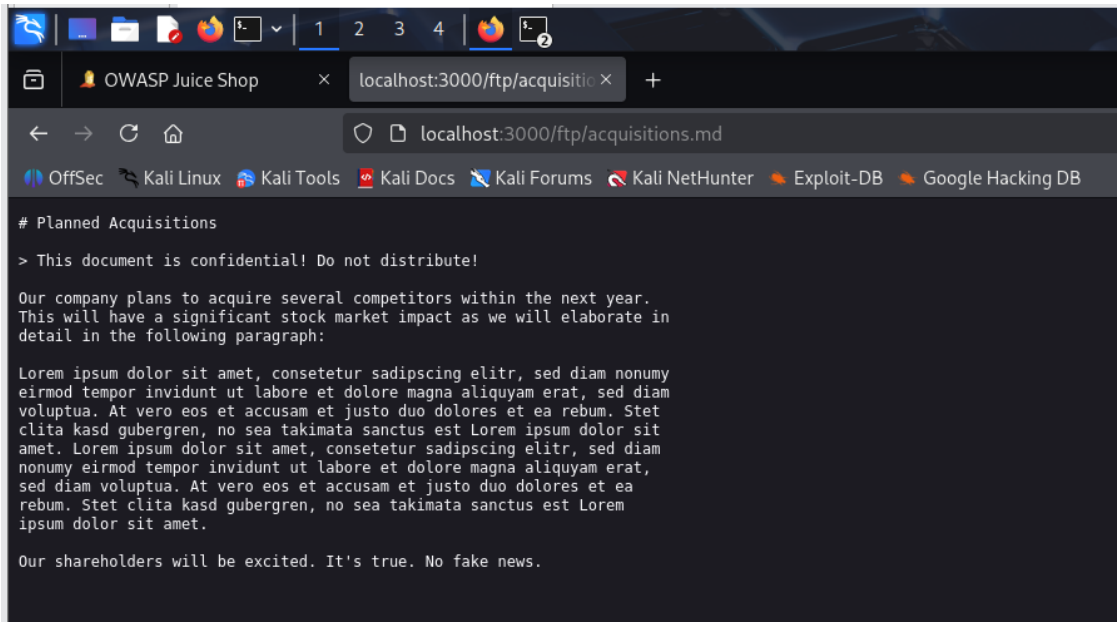


Figure 96: The acquisitions.md opened

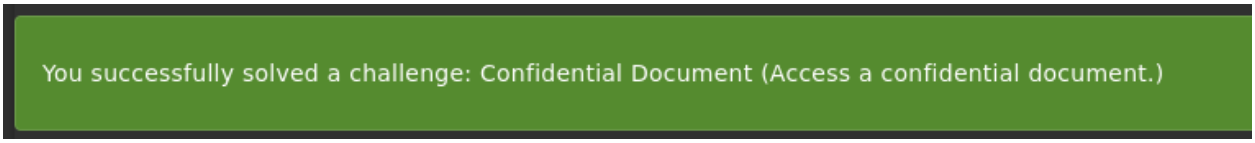


Figure 97: Successfully Completion message of accessing a confidential document

Impact: The public exposure of the "Planned Acquisitions" document, located in the /ftp directory of the OWASP Juice Shop instance, presents severe risks, including the potential for stock market manipulation and illegal insider trading due to pre-release knowledge of market-moving information. This significant security lapse could also lead to a substantial loss of competitive advantage if rival companies exploit the information, resulting in severe reputational damage and incur legal and regulatory penalties for the company's failure to protect confidential business intelligence.

2.4.2.1.2 User Credentials

- Goal: Exploit search bar to extract user emails and password hashes using SQL Injection.

Step 1: Open Juice Shop and locate the search bar.

- In the Search bar type in *"http://localhost:3000/rest/products/search?q=" GET request.*

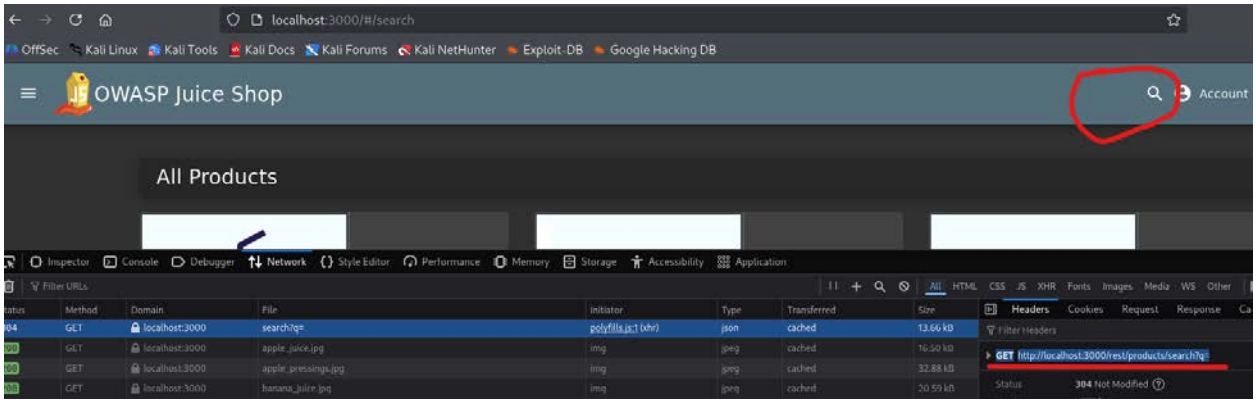


Figure 98: Browser Developer Tools

- After multiple try & errors we are able to find Email, Password of users using SQL injections, these are the results using "http://localhost:3000/rest/products/search?q=xxx%27))%20UNION%20SELECT%20email,%20password,%20%273%27,%20%274%27,%20%275%27,%20%276%27,%20%277%27,%20%278%27,%20%279%27%20FROM%20Users–"

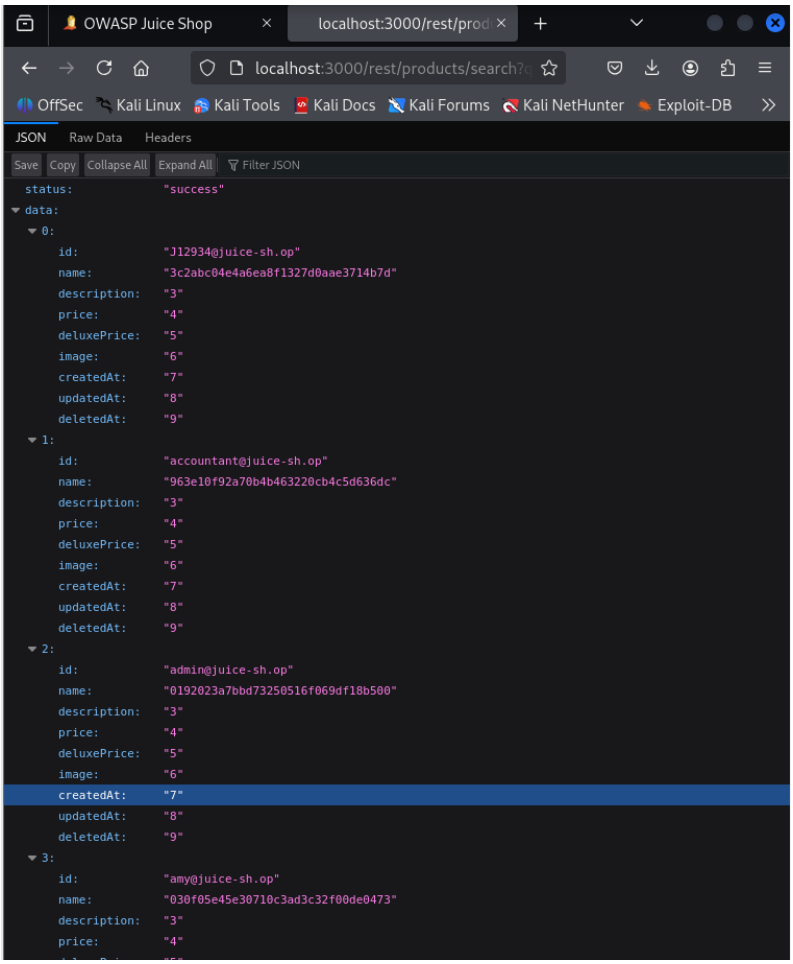


Figure 99: web browser displaying the raw JSON response

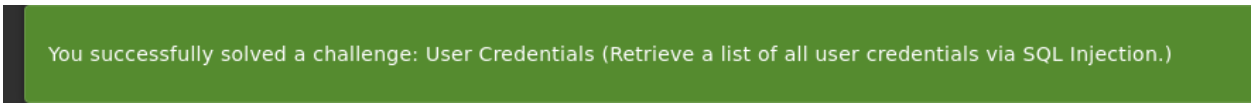


Figure 100: Completion of retrieving list of user credentials

Impact: The successful SQL injection attack, which exposed user emails and hashed passwords, which might cause severe risk to the organization and its users. This compromise of sensitive credentials could enable attackers to gain unauthorized access to user accounts within the application, potentially leading to further data breaches or malicious activities. Furthermore, if users have reused these passwords across other services, this vulnerability extends the risk beyond the immediate application, potentially

compromising other personal or financial accounts, ultimately leading to significant reputational damage and potential legal liabilities for the company.

2.4.2.1.3 Log file

- Goal: Discover hidden files like access logs using directory fuzzing.

Fuzzing wordlists, commonly used in Kali Linux, are collections of predefined inputs, such as common passwords, usernames, or known vulnerability patterns that are systematically fed into an application to discover weaknesses.

Step 1: Locate the /ftp directory with the url: http://localhost:3000/ftp/

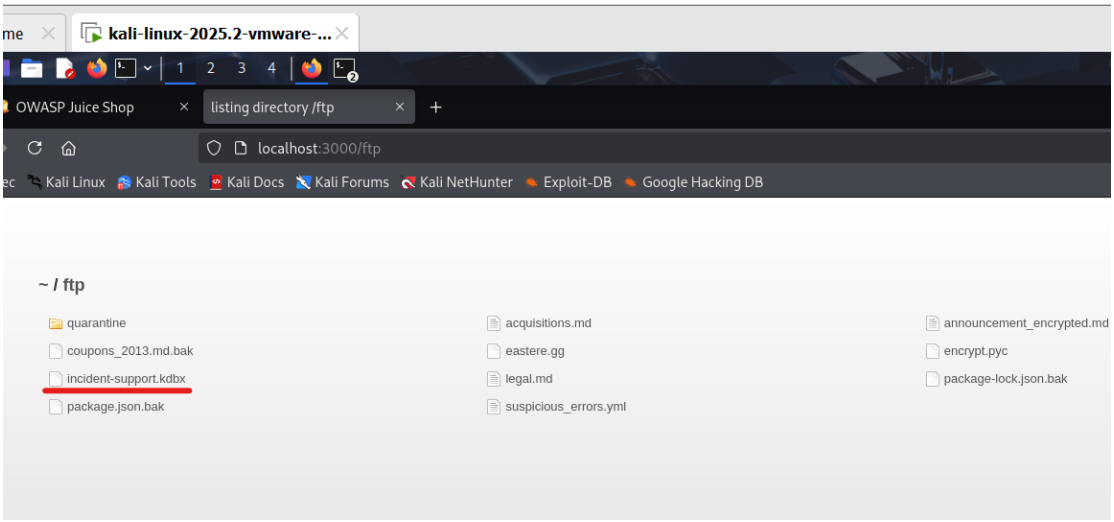


Figure 101: Inside the /ftp directory

We found out that there is a /ftp directory earlier, and inside there is a file named incident-support.kdbx as shown in ***Figure 101***. This filename, containing the word "support" which means a /support directory might exist.

Step 2: Run *ffuf -w /usr/share/wordlists/dirb/common.txt -u localhost:300/support/FUZZ* in the terminal

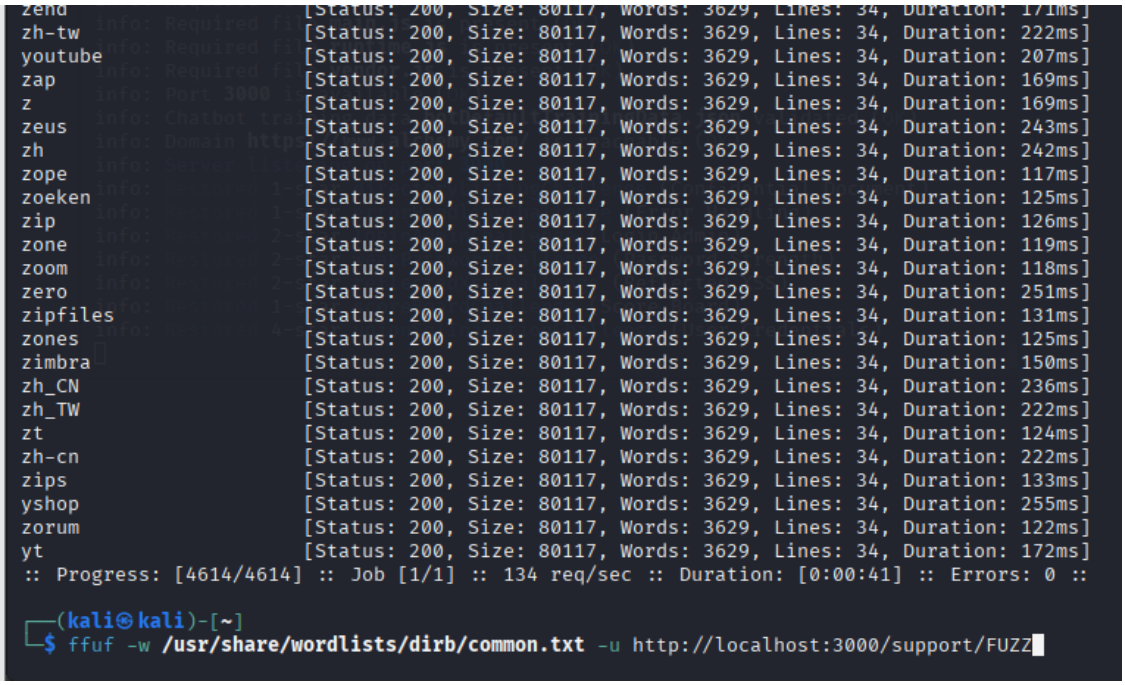


Figure 102: Run the command inside terminal

As shown in ***figure 102,*** It appears many files share the same size, this is because they likely trigger a default redirect or a "not found" page of a consistent size within the web application, leading to numerous false positive results during the fuzzing process.

Step 3: Filter out the false positive results with a new command: *ffuf -w /usr/share/wordlists/dirb/common.txt -u http://localhost:3000/support/FUZZ -fs 80117*

Figure 103: Refined ffuf scan using -fs to filter out.

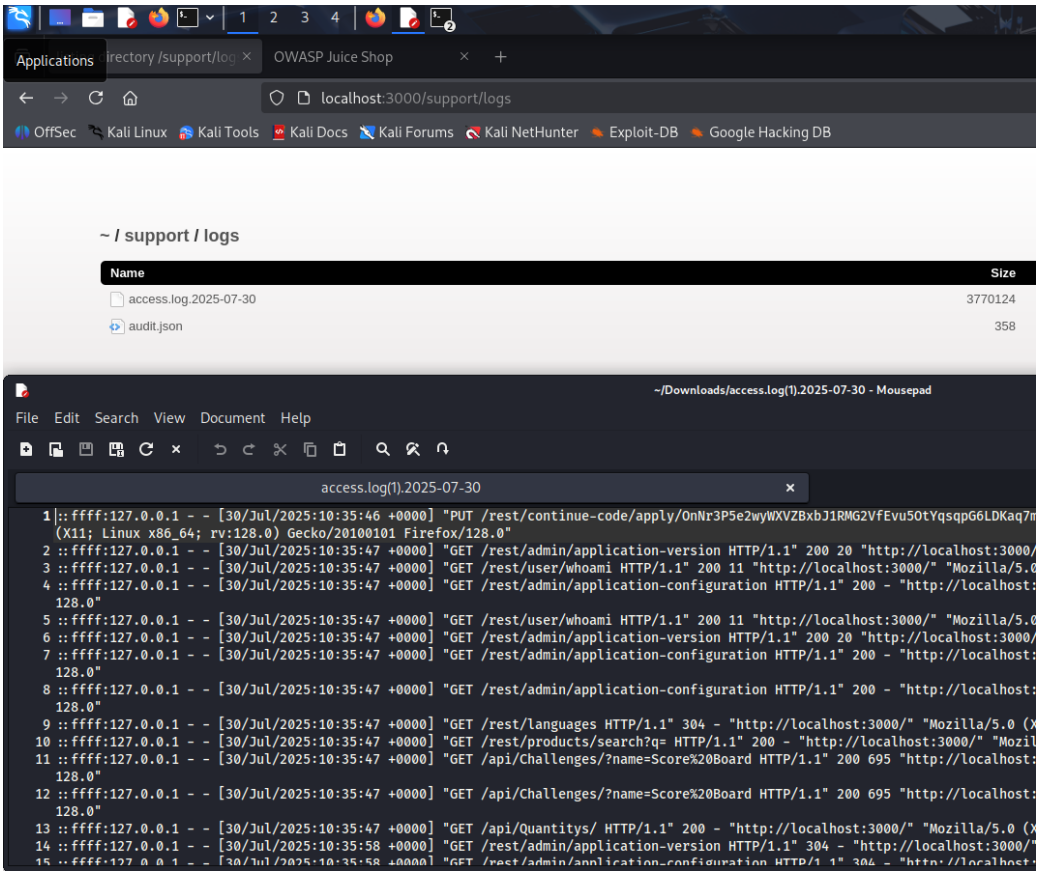


Figure 104: Accessing the discovered access.log

As shown in ***Figure 104***, we have successfully found the access.log file leaked by the support team. Its public availability could allow attackers to gather critical intelligence about the application's structure, user activity patterns, and potentially discover other hidden or vulnerable endpoints that users or automated systems have accessed. This information can then be leveraged to craft more targeted and effective attacks, compromising user privacy, system security, and ultimately leading to further exploitation of the application or its underlying infrastructure

2.4.2.2. Suggested remediation

1. **Restrict Access to Sensitive Directories**
    - Disable directory listing in the web server configuration.
    - Move sensitive directories like /ftp outside the web root.
    - Set correct file permissions to prevent unauthorized access.
2. **Protect Sensitive Files**

- Use authentication and authorization to restrict access to files.
- Store sensitive files in non-public locations.
- Avoid exposing backup or internal files on the server.

3. **Prevent SQL Injection**
    - Use parameterized queries or prepared statements.
    - Sanitize and validate all user inputs.
    - Employ input filtering techniques to block malicious payloads.
4. **Secure User Credentials**
    - Hash passwords using strong algorithms like **bcrypt**.
    - Do not store plain-text passwords in databases.
    - Implement rate-limiting and account lockout to prevent brute-force attacks.
5. **Protect Log Files**
    - Store logs in secured, non-public directories.
    - Regularly rotate and monitor logs.
    - Avoid logging sensitive information like passwords or tokens.
6. **Apply Least Privilege Principle**
    - Ensure files and directories have the minimum permissions required.
    - Regularly audit file and directory permissions on the server.
7. **Use a Web Application Firewall (WAF)**
    - Deploy a WAF to detect and block common attacks like SQL injection, directory traversal, and file access attempts.
8. **Perform Security Testing**
    - Conduct regular vulnerability assessments and penetration testing.
    - Patch and update components promptly to fix known issues.