(Web Application Security Assessment Report)

Project Title:

● Vulnerability Assessment of OWASP Juice Shop Using Industry-Standard Tools

Submitted By: Piyush Singh

Cybersecurity Intern
[*EMAIL*] [*LINKEDIN*] [*GITHUB*]

Internship Program:
Future Interns - Cybersecurity Track

Date of Submission:

Tools & Platforms Used:

● Kali Linux (Virtual Machine)
● OWASP Juice Shop (Docker-hosted)
● Burp Suite Community Edition
● OWASP ZAP
● Nikto
● SQLMap

Target Web Application:

● OWASP Juice Shop
● Deployed locally using Docker
● Accessible via *http://localhost:3000* (or container IP)

Internship Task:

Conduct vulnerability assessment of a web application and identify flaws such as:
● SQL Injection
● Cross-site Scripting (XSS)
● Cross-Site Request Forgery (CSRF)
● Authentication Misconfigurations
● Security Misconfigurations
● Mapped to OWASP Top 10 guidelines.

Project Type:

Cybersecurity Internship Task 1 – Web Application Security Testing
GitHub Repository: FUTURE_CS_01 Track (Code: CS)

Executive Summary

This report presents the results of a web application security assessment conducted as part of the Cybersecurity Internship Task 1 under the Future Interns Program. The assessment targeted the intentionally vulnerable application OWASP Juice Shop, hosted locally using Docker and tested within a Kali Linux virtual environment.

The goal of the project was to identify and analyze common web application vulnerabilities aligned with the OWASP Top 10 threat categories, using industry-standard ethical hacking tools. Throughout the assessment, tools like OWASP ZAP, Nikto, SQLMap, and Burp Suite were utilized to conduct active and passive scanning, test for injection flaws, misconfigurations, broken access control, and client-side vulnerabilities.

Key phases of the assessment included:

- Reconnaissance and scanning using OWASP ZAP and Nikto
- Automated exploitation of SQL Injection using SQLMap
- Manual testing of XSS and CSRF attacks via Burp Suite Repeater
- Validation and analysis of authentication, session handling, and data exposure issues

Key Vulnerabilities Identified:

- SQL Injection on product search parameter (q)
- Cross-Site Scripting (XSS) via user input on the search and feedback forms.
- Broken Authentication allowing login bypass techniques.
- Sensitive Data Exposure including emails and password hashes (from Users table)
- Broken Access Control through manipulation of user privileges and endpoints.
- Security Misconfigurations like missing CSP, clickjacking headers, and exposed JS libraries.

Each of these vulnerabilities was documented with:

- Impact level (High / Medium / Low)
- Screenshots of exploitation
- Proof of concept (PoC) payloads
- Mitigation strategies based on OWASP guidelines

The findings from this assessment can serve as a guide to secure the Juice Shop application, and by extension, provide real-world value to developers and organizations seeking to defend against modern web threats.

Table of Contents

(Tools and Environment Setup)

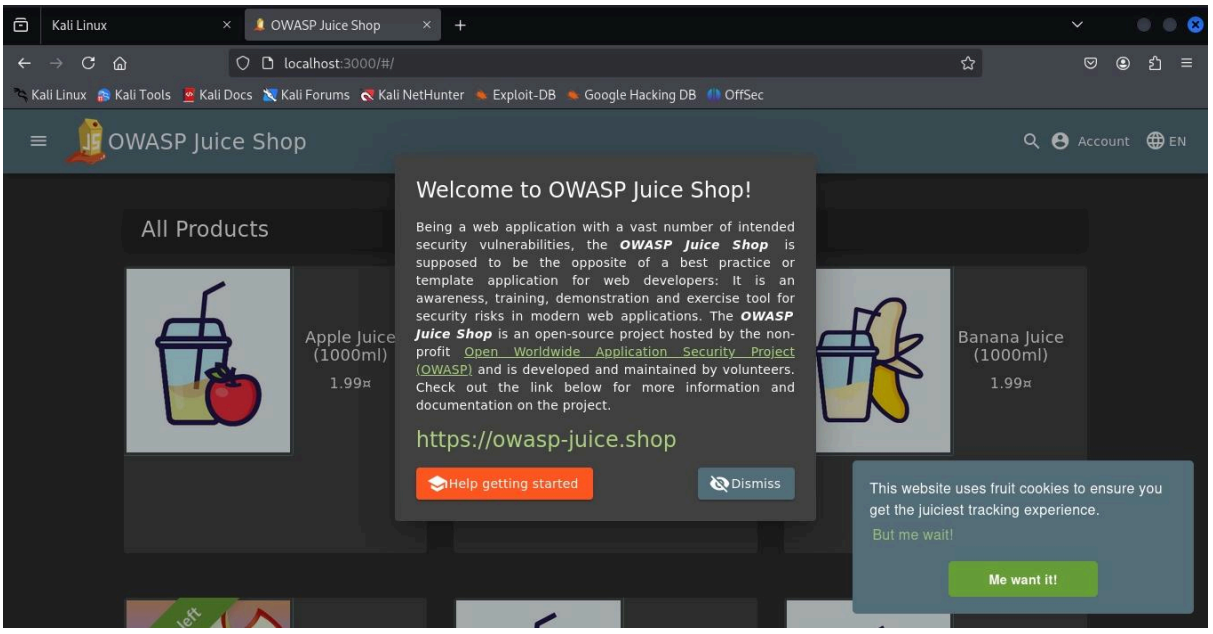
This section provides an overview of the tools, technologies, and platforms used to perform the web application security testing on OWASP Juice Shop during Task 1 of the cybersecurity internship.

Operating System:

- Kali Linux (Running on VirtualBox)
- Version: Kali Linux 2024.x (latest rolling)
- Use: Pre-installed penetration testing tools and terminal-based utilities.

Target Web Application:

- OWASP Juice Shop



- Description: Intentionally vulnerable web application used for security training.
- Deployed via Docker using the command:

docker run -d --name juice-shop -p 3000:3000 bkimminich/juice-shop

- Accessed using browser at: *http://localhost:3000* or Docker IP http://172.x.x.x:3000

Tools Used:

- **OWASP ZAP=** Automatic vulnerability scanning for OWASP Top 10 issues
- **Nikto=** Web server vulnerability scanner to detect outdated software and configs
- **SQLMap=** SQL Injection testing and database extraction

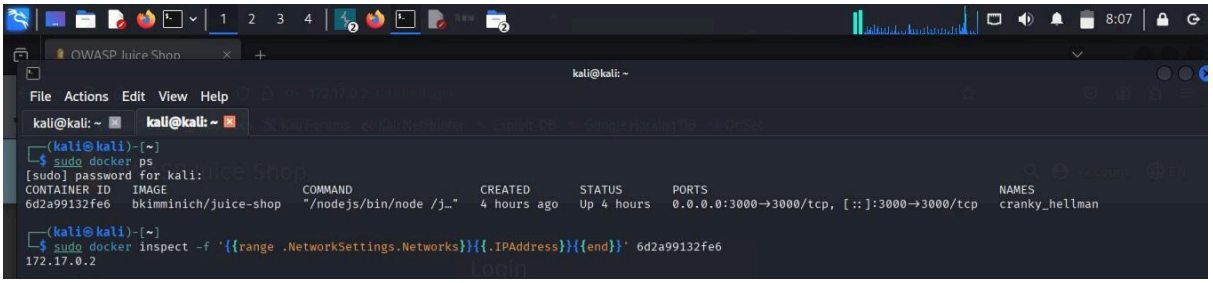
● **Burp Suite CE=** Manual testing for XSS, CSRF, and other client-side/server-side vulnerabilities
● **Docker=** Containerized deployment of OWASP Juice Shop
● **Browser (Firefox)=** Used for interacting with the target app and proxying traffic to Burp Suite

Networking Configuration:

● Browser Proxy Setup:
● Configured to route traffic through Burp Suite (127.0.0.1:8080)
● Manual switch between Burp and direct connections to handle Docker IP issues

Docker Network:

● Used default bridge network
● Juice Shop available on 172.17.0.2:3000 internally



Documentation Tools:

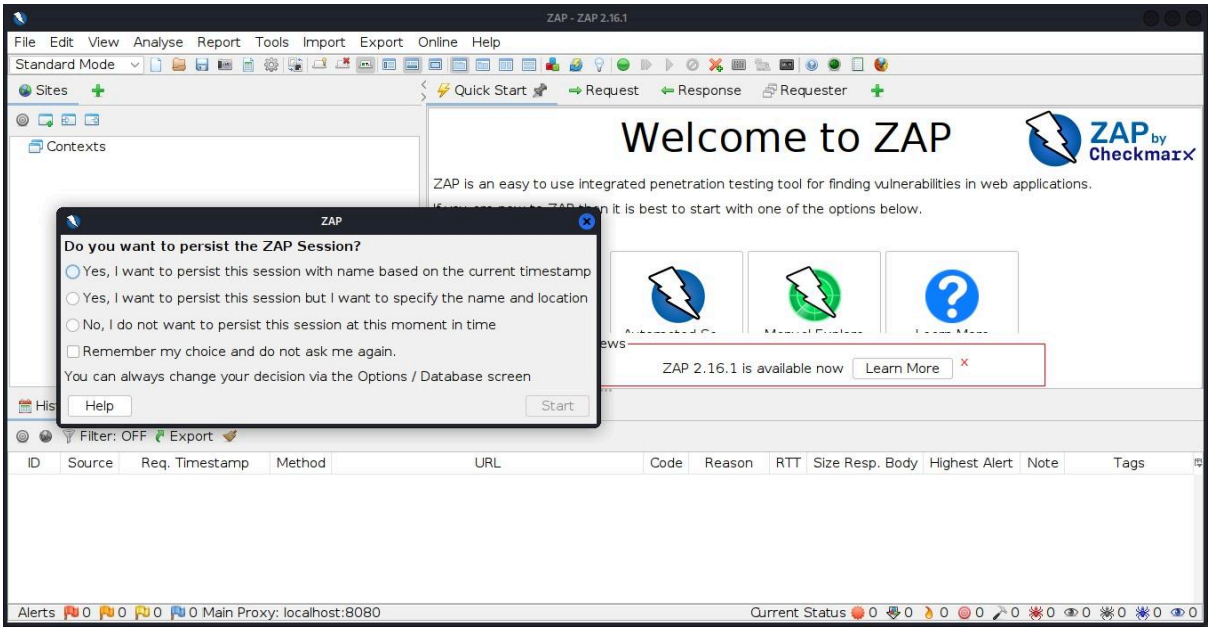
● Google Docs / MS Word – Report compilation
● PDF Export – Final submission format
● GitHub – Task submission repository (as per guidelines)

(OWASP ZAP Scan & Findings)

Objective:

The objective of this step was to perform an automated vulnerability scan on the OWASP Juice Shop web application using OWASP ZAP (Zed Attack Proxy). The scan aims to identify common web vulnerabilities as outlined in the OWASP Top 10.

OPEN OWASP ZAP:



Tool Used:

- Tool: OWASP ZAP (Zed Attack Proxy)
- Version: ***[Installed Version: 2.16.1]***
- Mode: Automated Active Scan
- Target URL: *http://localhost:3000 or http://172.17.0.2:3000*

Steps Followed

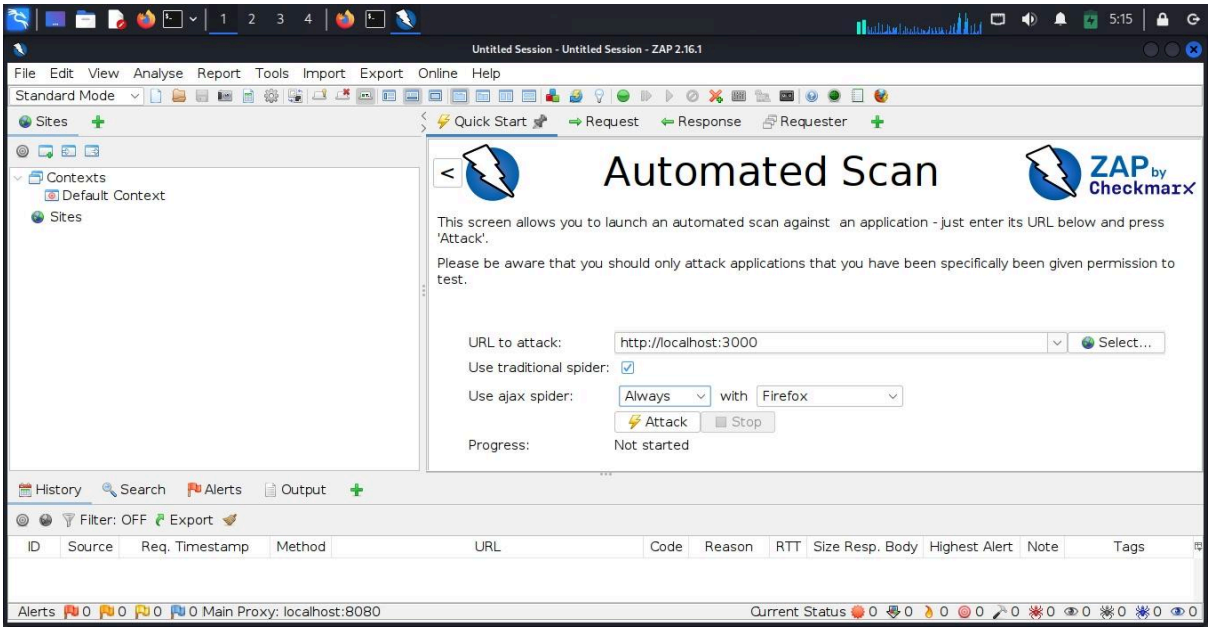
1. Launched OWASP ZAP on Kali Linux.

2. Set up the browser (Firefox) to use ZAP as the proxy (127.0.0.1:8080).

3. Opened Juice Shop in the browser to allow ZAP to record all requests.

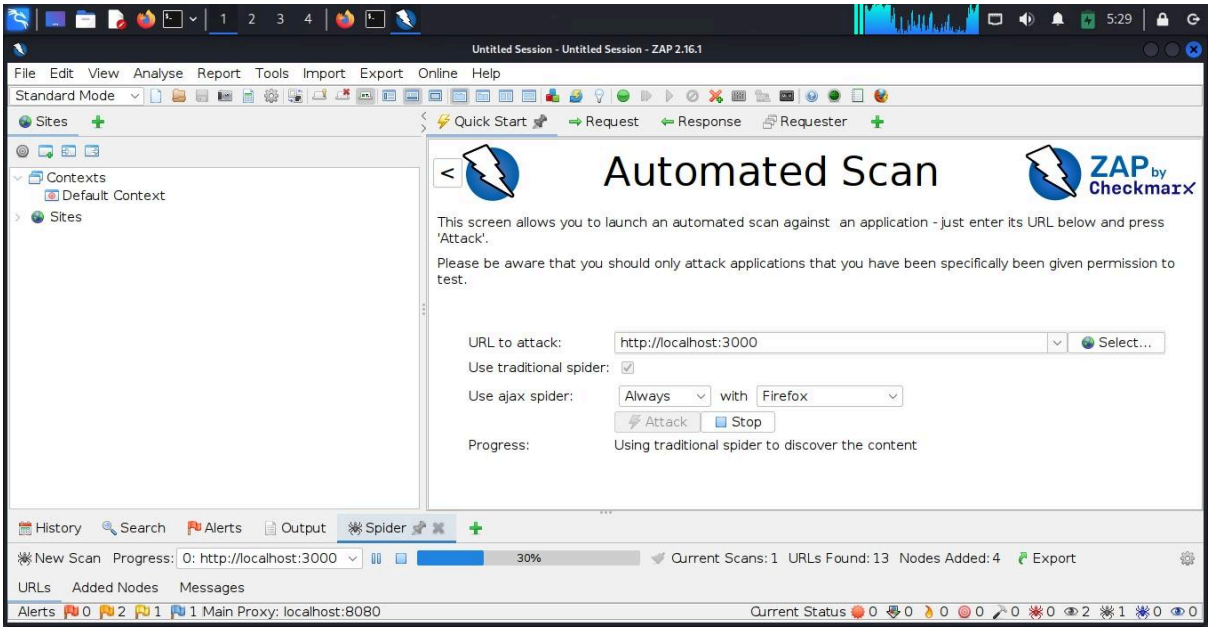
4. Right-clicked the root URL in the ZAP sitemap → Clicked "Attack" > Active Scan.

SET TARGET URL & OTHER OPTIONS:



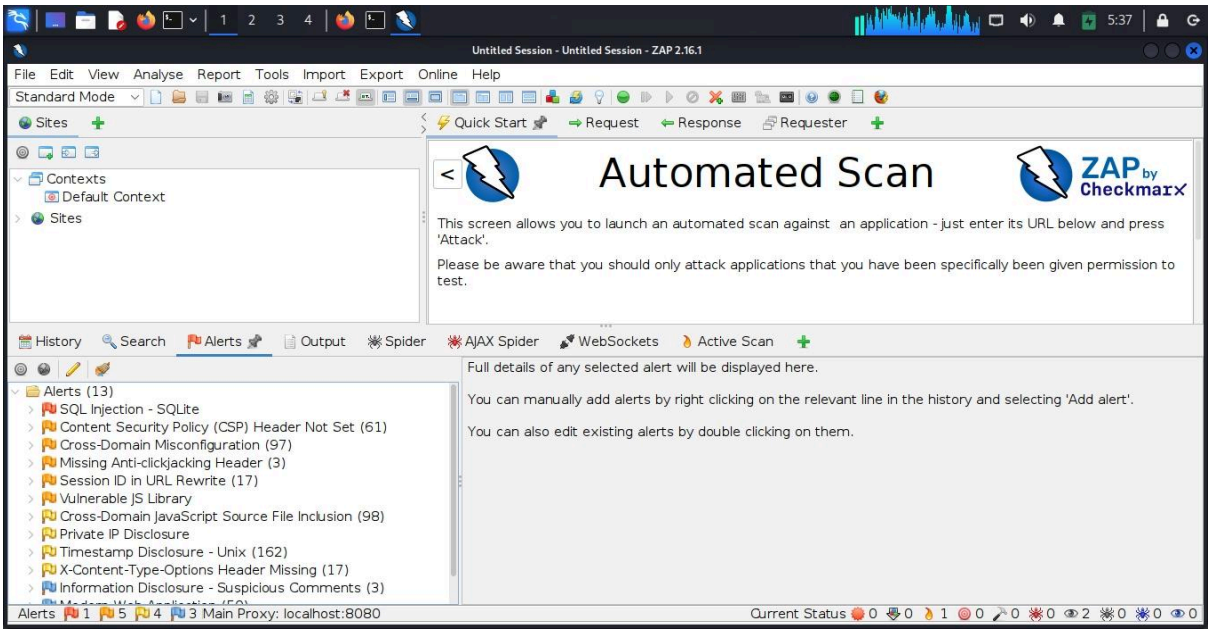
👇👇👇👇

NOW CLICK ON ATTACK:



5. Allowed ZAP to complete the scan of all endpoints and directories.

GOT ALL ALERTS:



6. Exported and reviewed the alerts from the ZAP scan results.

Vulnerabilities Identified:

Summary of Alerts:

Risk Level	Number of Alerts
High	1
Medium	5
Low	4
Informational	4

Alerts:

Name	Risk Level	Number of Instances
SQL Injection - SQLite	High	1
Content Security Policy (CSP) Header Not Set	Medium	61
Cross-Domain Misconfiguration	Medium	97
Missing Anti-clickjacking Header	Medium	3
Session ID in URL Rewrite	Medium	17
Vulnerable JS Library	Medium	1
Cross-Domain JavaScript Source File Inclusion	Low	98
Private IP Disclosure	Low	1
Timestamp Disclosure - Unix	Low	162
X-Content-Type-Options Header Missing	Low	17
Information Disclosure - Suspicious Comments	Informational	3
Modern Web Application	Informational	50
Retrieved from Cache	Informational	3
User Agent Fuzzer	Informational	120

Remarks:

The automated scan performed by OWASP ZAP helped in quickly identifying multiple serious vulnerabilities, especially the SQL Injection (SQLite) and missing CSP headers, which are commonly exploited in real-world attacks. Each issue was mapped to its corresponding OWASP Top 10 category.

🟩 **Next Step**: ***Proceeded to deeper server-side testing using Nikto and SQLMap for manual validation and data extraction.***

(Nikto Scan & Findings)

Objective:

● To perform a web server vulnerability assessment on the OWASP Juice Shop application using Nikto, a fast and open-source web server scanner. The goal was to identify insecure files, outdated components, and server misconfigurations.

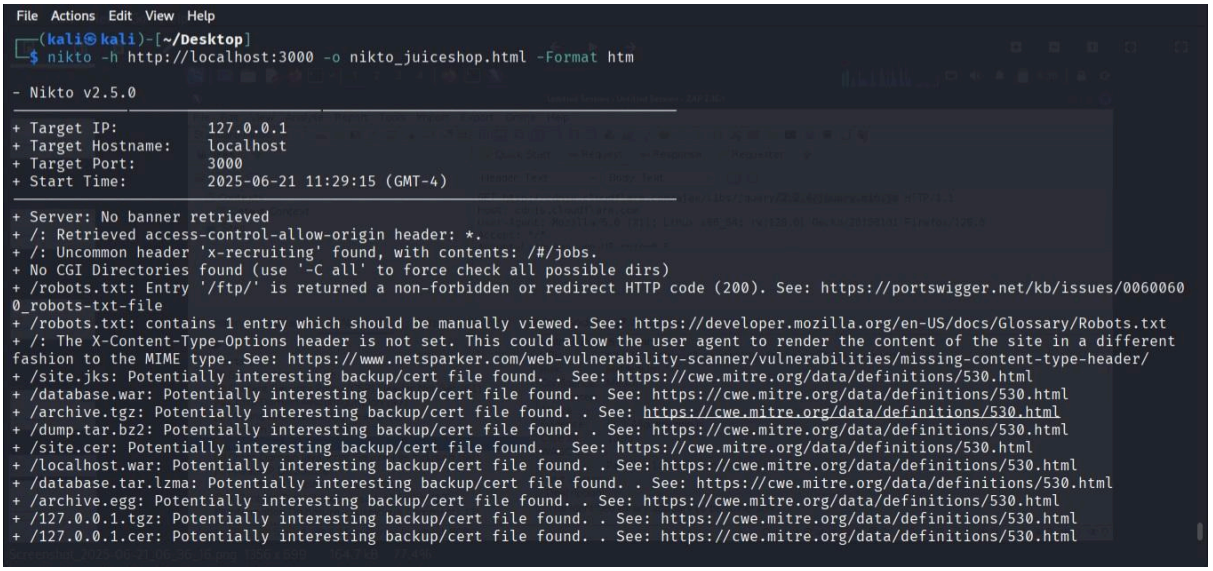
Tool Used:

● Tool: Nikto Web Server Scanner
● Platform: Kali Linux Terminal
● Target URL: http://localhost:3000 (mapped to OWASP Juice Shop)

Command Executed:

● ***nikto -h http://localhost:3000 -o nikto_juiceshop.html -Format htm***
● This command initiated a vulnerability scan on the Juice Shop web server hosted locally.

NIKTO COMMAND:



Summary of Key Findings from Nikto:

● The Nikto web server scanner identified multiple vulnerabilities and risky exposures on the OWASP Juice Shop web application hosted at http://localhost:3000. These include missing security headers, publicly accessible sensitive files, and misconfigurations.

MITIGATION STRATEGIES:

Description	Mitigation
Absence of this header allows MIME-sniffing attacks, potentially leading to XSS.	Add the header `X-Content-Type-Options: nosniff` to all server responses.
The `/robots.txt` file discloses hidden directories like `/ftp/`, which should not be accessible.	Restrict access or remove sensitive entries from `robots.txt`.
Files like `/site.jks`, `/database.war`, `/dump.tar.bz2`, etc. were found publicly accessible. These may contain sensitive or exploitable data.	Restrict file access or remove such files from the public web directory.
Custom headers like `X-Recruiting` may provide unnecessary internal information.	Remove unnecessary headers from server configuration.
The path `/wp-content/plugins/nextgen-gallery/.../jqueryFileTree.php` hints at potential LFI (Local File Inclusion) vulnerability.	Validate and sanitize all file access input paths.

Owasp Top 10 Mapping:

Mapped Finding from Nikto	OWASP Category
Publicly accessible sensitive files such as backups (`.jks`, `.pem`, `.tar`, `.war`, etc.)	A01:2021 – Broken Access Control
Potential LFI paths via vulnerable PHP scripts in WordPress plugin structures	A03:2021 – Injection
Missing security headers, misconfigured `robots.txt`, exposed sensitive files	A05:2021 – Security Misconfiguration
Presence of outdated/ known vulnerable paths (like old WordPress plugin structure references)	A06:2021 – Vulnerable & Outdated Components
Disclosure via misconfigured or accessible credential files (`site.jks`, `.pem`)	A07:2021 – Identification & Authentication Failures

Remarks:

Nikto is a highly effective tool for quickly identifying low-hanging vulnerabilities and web server misconfigurations. It provided valuable insight into header-based issues and confirmed the presence of an SQL injection vector that was exploited further using SQLMap.

🟩 **Next Step**: ***Proceed to deeper exploitation using Injection - Authentication Bypass & SQLMap to enumerate databases and extract sensitive user data***

(SQL Injection - Authentication Bypass)
This task will be focusing on injection vulnerabilities. Injection vulnerabilities are quite dangerous to a company as they can potentially cause downtime and/or loss of data. Identifying injection points within a web application is usually quite simple, as most of them will return an error. There are many types of injection attacks, some of them are:

SQL Injection:

SQL Injection is when an attacker enters a malicious or malformed query to either retrieve or tamper data from a database. And in some cases, log into accounts.

Command Injection:

Command Injection is when web applications take input or user-controlled data and run them as system commands. An attacker may tamper with this data to execute their own system commands. This can be seen in applications that perform misconfigured ping tests.

Email Injection:

Email injection is a security vulnerability that allows malicious users to send email messages without prior authorization by the email server. These occur when the attacker adds extra data to fields, which are not interpreted by the server correctly.

But in our case, we will be using SQL Injection.
*For more details visit **Injection***

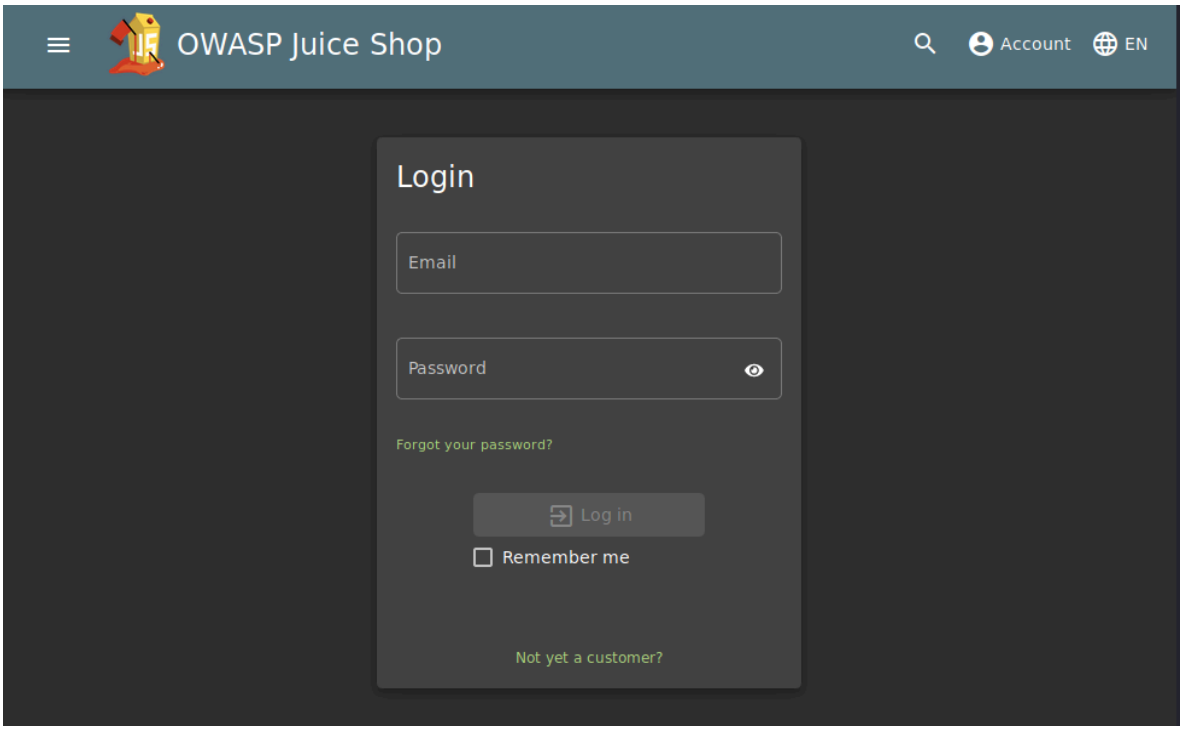
Vulnerability Overview:

I modified the login form input like this:
- Email: ' OR 1=1--
- Password: anything (e.g., test, a)

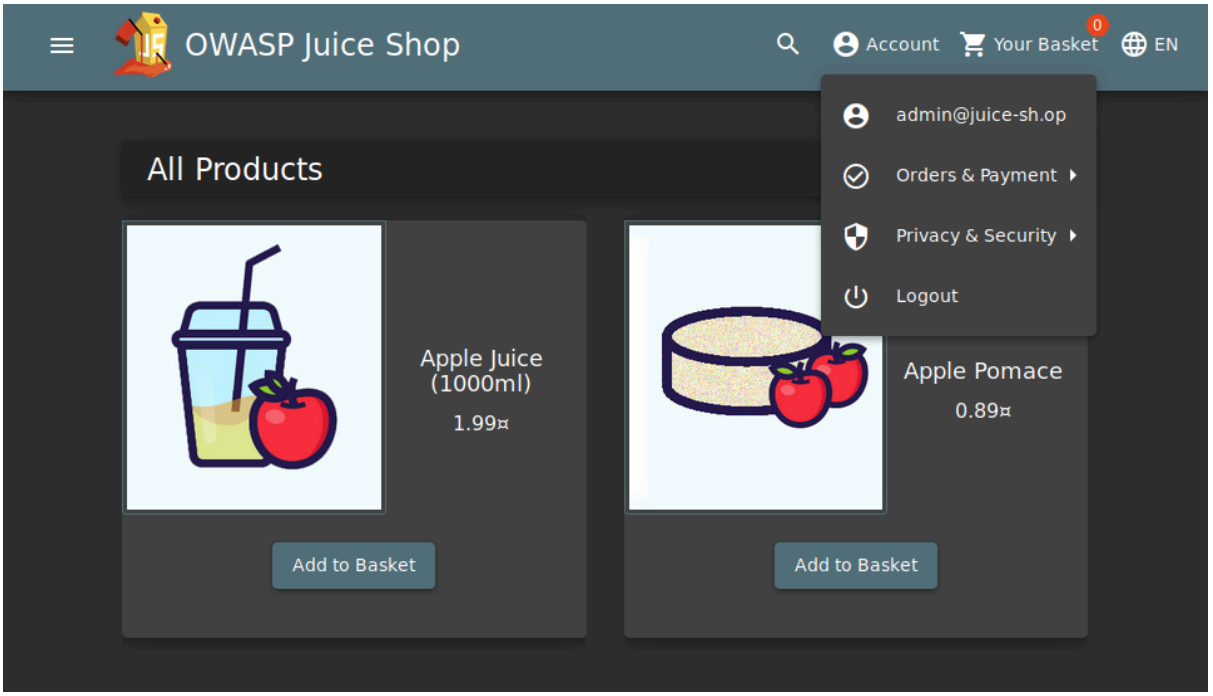
STEP1:

- Email: ' OR 1=1--

- Password: anything (e.g., test, a)

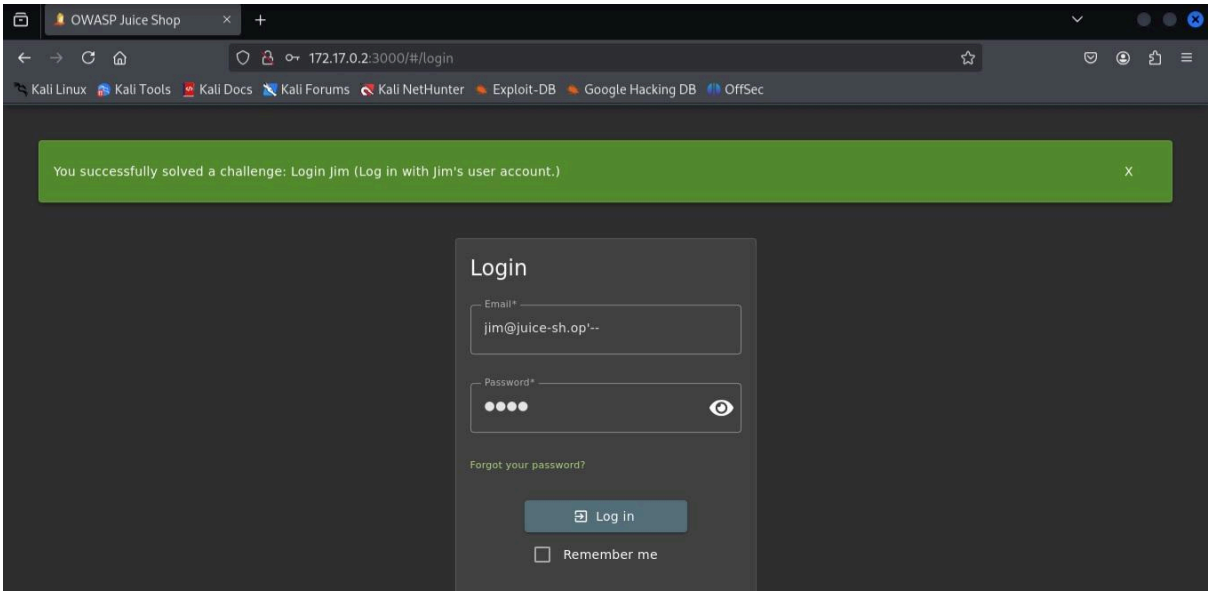


STEP 2: AFTER USING

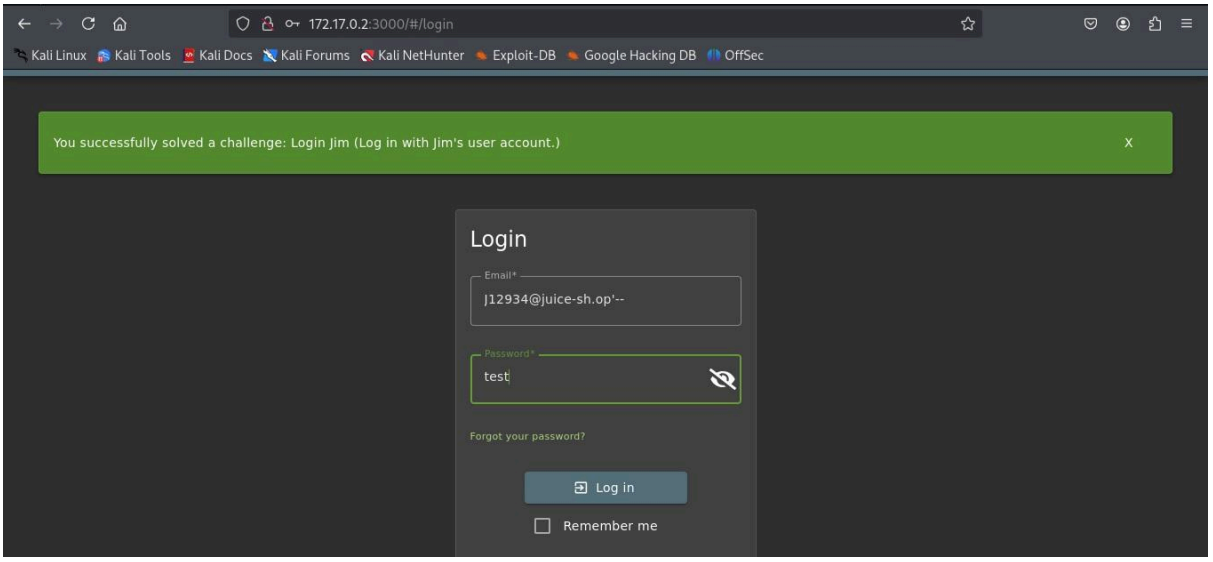


STEP 3: Bypass Other Login’s

- Email: Any Email with character's(eg: *juice@juice-sh.op’--)*
- Password: anything (e.g., test, a, etc)



👇(same)👇



This payload bypassed authentication by terminating the SQL query prematurely and injecting a logic clause that always evaluates to TRUE.

Explanation (How it Works):

● Suppose the original backend SQL query is:
SELECT * FROM users WHERE email = '<input>' AND password = '<input>';

● My payload changes it to:
SELECT * FROM users WHERE email = '' OR 1=1--' AND password = 'test';

● Here ' OR 1=1-- closes the email string, adds a condition that is always true, and comments out the rest (--).
● The query always returns the first user record (usually an admin), bypassing password verification.

Vulnerability Chart (with OWASP Mapping):

Mitigation Strategy	OWASP Top 10 Mapping	Vulnerability	Description
- Use **parameterized queries / prepared statements**- Strict **input validation**- Enable proper error handling and avoid showing SQL errors to users	A01:2021 – Broken Access Control A03:2021 – Injection	SQL Injection – Authentication Bypass	Input like ' `OR 1=1--` allowed bypassing the login form and accessing the application as a valid user.

Why does this work?
1. The character ' will close the brackets in the SQL query

2. 'OR' in a SQL statement will return true if either side of it is true. As 1=1 is always true, the whole statement is true. Thus it will tell the server that the email is valid, and log us into user id 0, which happens to be the administrator account.

3. The -- character is used in SQL to comment out data, any restrictions on the login will no longer work as they are interpreted as a comment. This is like the # and // comment in python and javascript respectively.

SQLMap Exploitation & Data Extraction:

Objective:

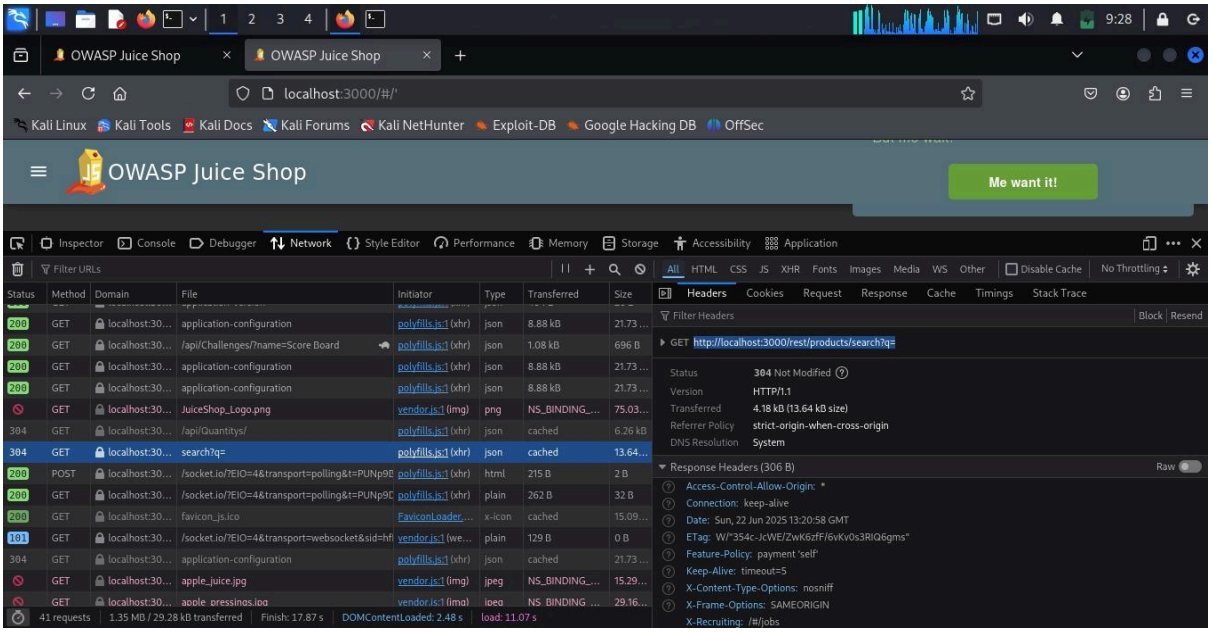
The purpose of this phase was to leverage the SQL Injection vulnerability identified in the q parameter of the /rest/products/search endpoint and automate data extraction using SQLMap, a powerful open-source SQLi tool.

Tool Used:

- Tool: SQLMap
- Platform: Kali Linux Terminal
- Target URL: *http://localhost:3000/rest/products/search?q=test*
- Database Detected: SQLite

Identifying the Injectable Parameter:

Before initiating the SQLMap scan, I manually analyzed the request flow of the application using Firefox Developer Tools (Network tab). While exploring the application, I noticed that the /rest/products/search endpoint accepts a GET parameter q, which is used to search for products.



The request looked like:

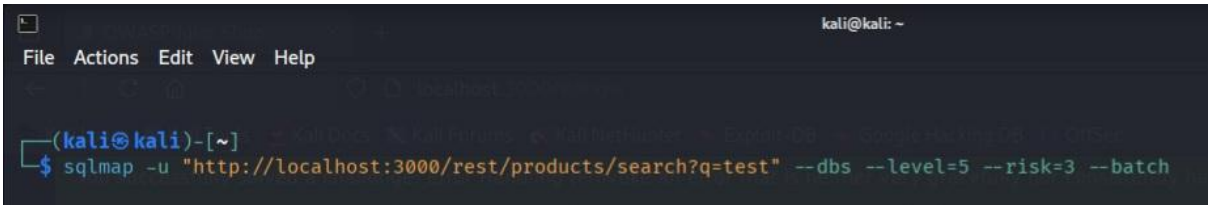
GET /rest/products/search?q=

- This suggested a possible injection point, and I showed its exploitability in next phase using SQLMap.

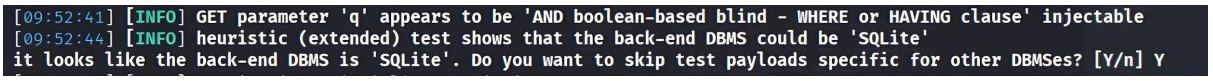
Steps & Command Executed:

Step	Action Performed	Command Used
1	Identified injectable parameter and extracted available databases	`sqlmap -u "http://localhost:3000/rest/products/search?q=test" --dbs --level=5 --risk=3 --batch`
2	Listed all tables in the default database	`sqlmap -u "http://localhost:3000/rest/products/search?q=test" --tables --batch`
3	Retrieved all column names from the `Users` table	`sqlmap -u "http://localhost:3000/rest/products/search?q=test" --columns -T Users --batch`
4	Dumped sensitive user data from the `Users` table, including **emails, usernames, and hashed passwords**	`sqlmap -u "http://localhost:3000/rest/products/search?q=test" --dump -T Users --batch`

STEP 1.

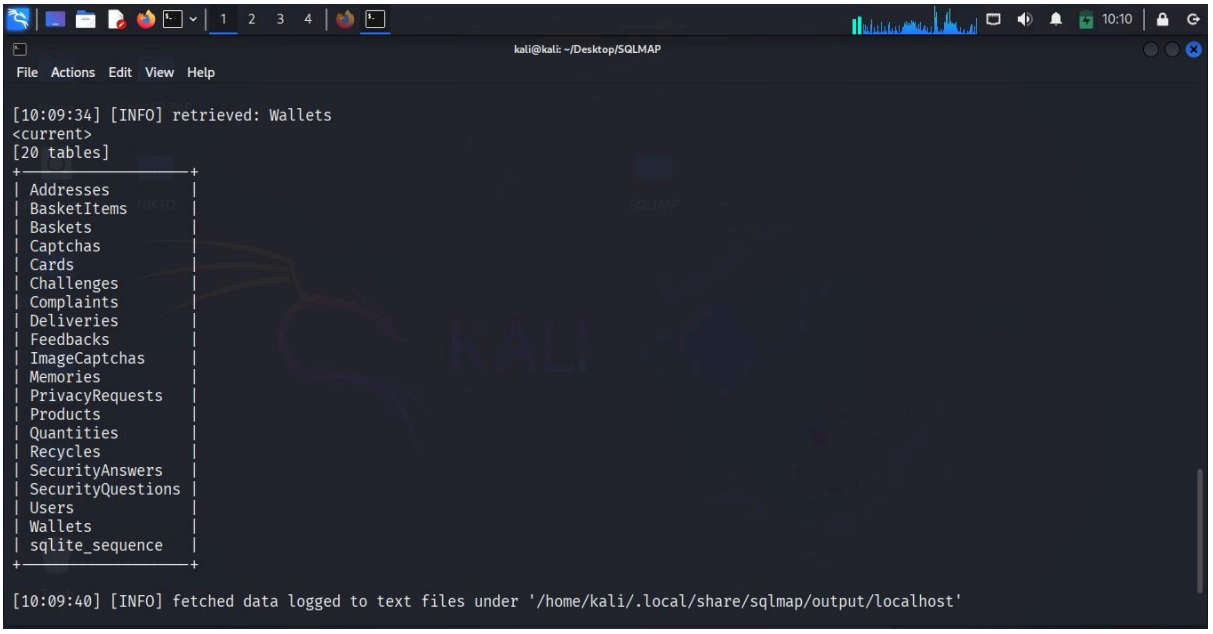


OUTPUT:



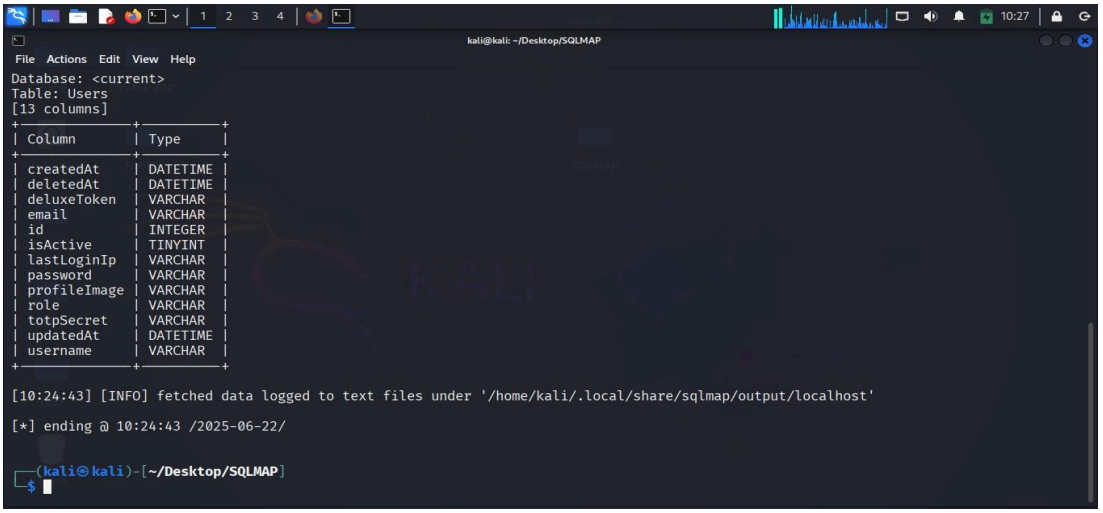
STEP 2.

OUTPUT:



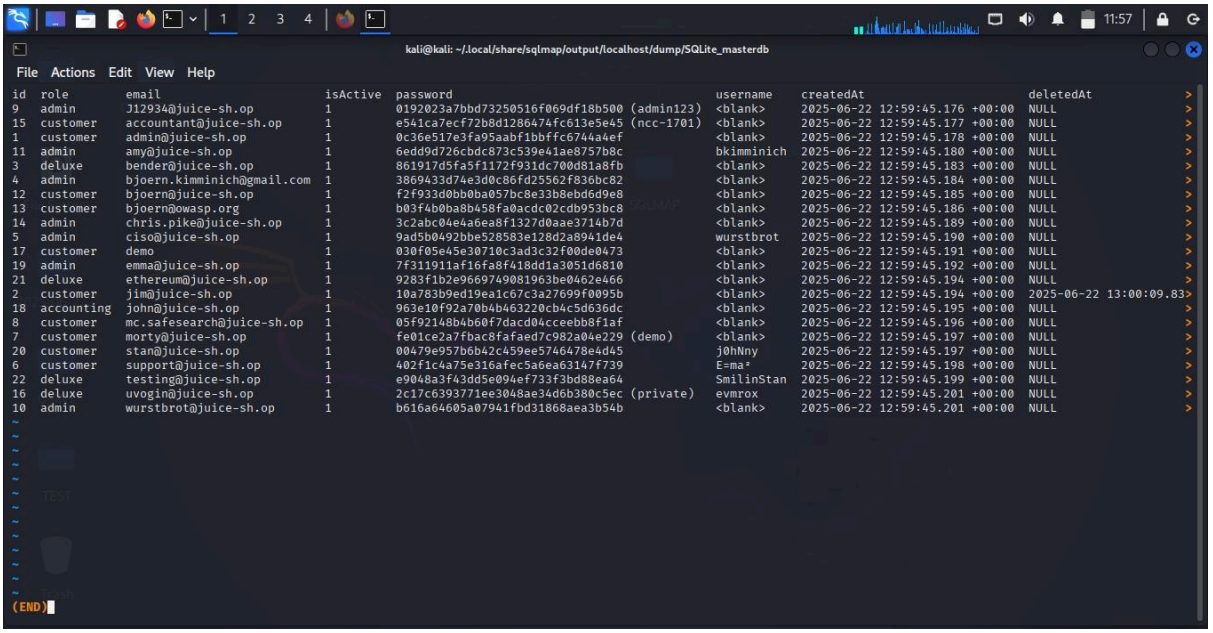
STEP 3.

OUTPUT:



STEP 4:

OUTPUT:



OWASP Top 10 Mapping:

Vulnerability	OWASP Top 10 Category
SQL Injection in Search	A01:2021 – Broken Access Control
Data Extraction via SQLMap	A03:2021 – Injection

Mitigation Strategies:

Issue	Recommendation
SQL Injection	Use **parameterized queries**, prepared statements, and strong input validation
Hashed Password Disclosure	Use **strong hashing** algorithms with salting (e.g., bcrypt, Argon2), and store passwords securely
Excessive Data Disclosure	Enforce **principle of least privilege**, limit DB read scopes, and monitor logs for extraction abuse

Extracted Data Summary:

- From the Users table, SQLMap successfully extracted:
- Email addresses of users
- Usernames
- Hashed passwords (in SHA256 or MD5 format)
- Timestamps and user role information

Remarks:

This SQL injection vulnerability enabled full read access to backend user data. Combined with hash cracking techniques, an attacker could impersonate users or escalate privileges. SQLMap made exploitation efficient and provided confirmation of database weakness.

🟩 **Next Step:** ***Performed deeper exploitation using SQLMap to enumerate databases and extract sensitive user data.***
🟪 **Following That*****: Moved to manual testing using Burp Suite, which allowed identification of additional critical vulnerabilities including:***

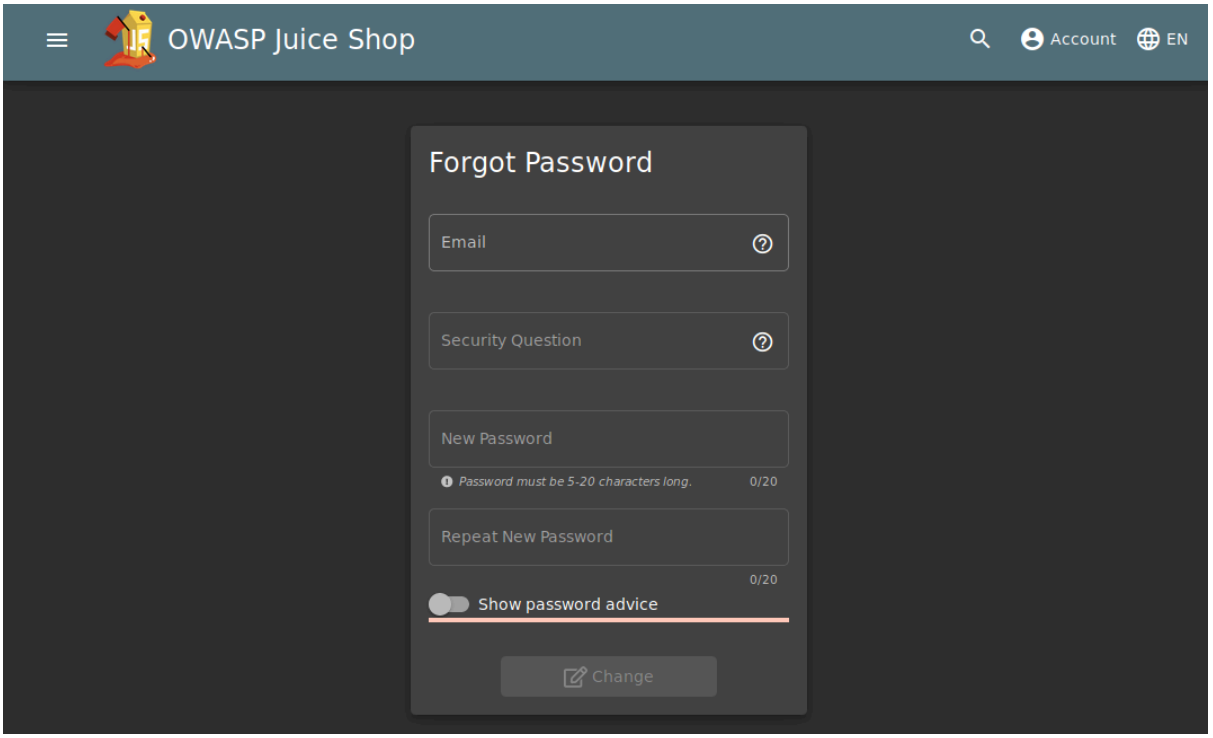
Burp Suite Manual Testing (Broken Auth, Sensitive Data, XSS, Access Control):

Broken Authentication – Exploitation Walkthrough (Burp Suite):

During manual testing using Burp Suite, I discovered a serious Broken Authentication vulnerability in the OWASP Juice Shop login mechanism, allowing unauthorized access to user accounts and password reset bypass.

Tool Used:

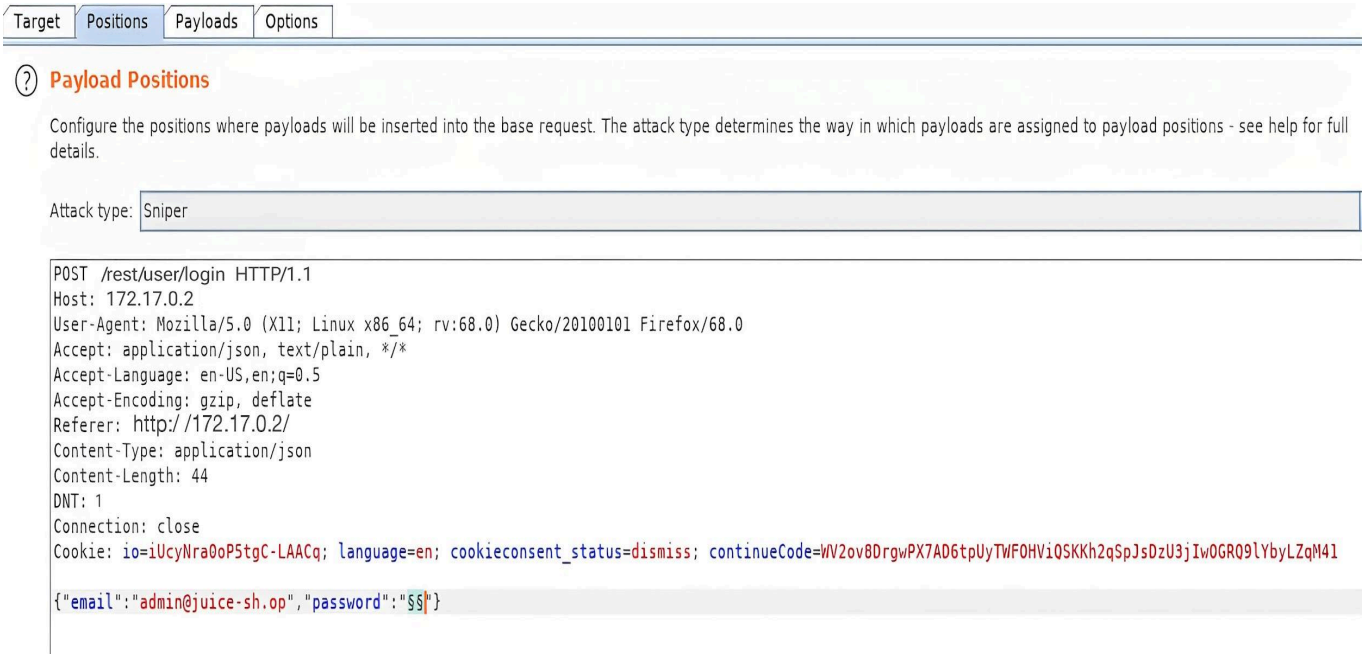
- Burp Suite Community Edition
- Tested via Firefox browser with proxy set to 127.0.0.1:8080
- ***Target: http://localhost:3000 or http://172.17.0.2:3000 (OWASP Juice Shop running via Docker)***



Phase 1: Brute-Force Login Using Burp Suite:

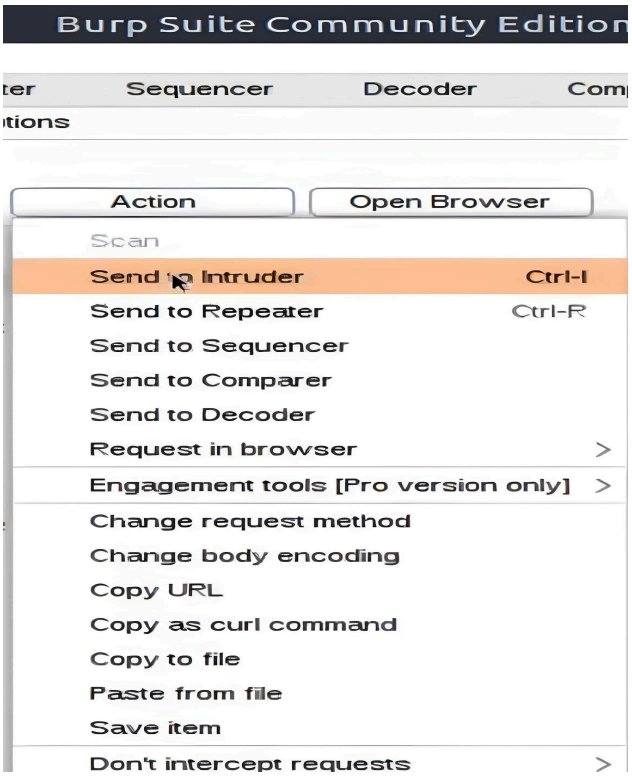
I began by capturing a login request using Burp Suite's Proxy and forwarded it to Intruder for a password brute-force attack. The target user was Jim, whose email was discovered earlier during the SQLMap enumeration phase.

STEP 1: CAPTURE LOGIN REQUEST



● *To perform the attack, I used a password wordlist located at:* ***/usr/share/wordlists/SecLists/Passwords/Common-Credentials/best1050.txt***

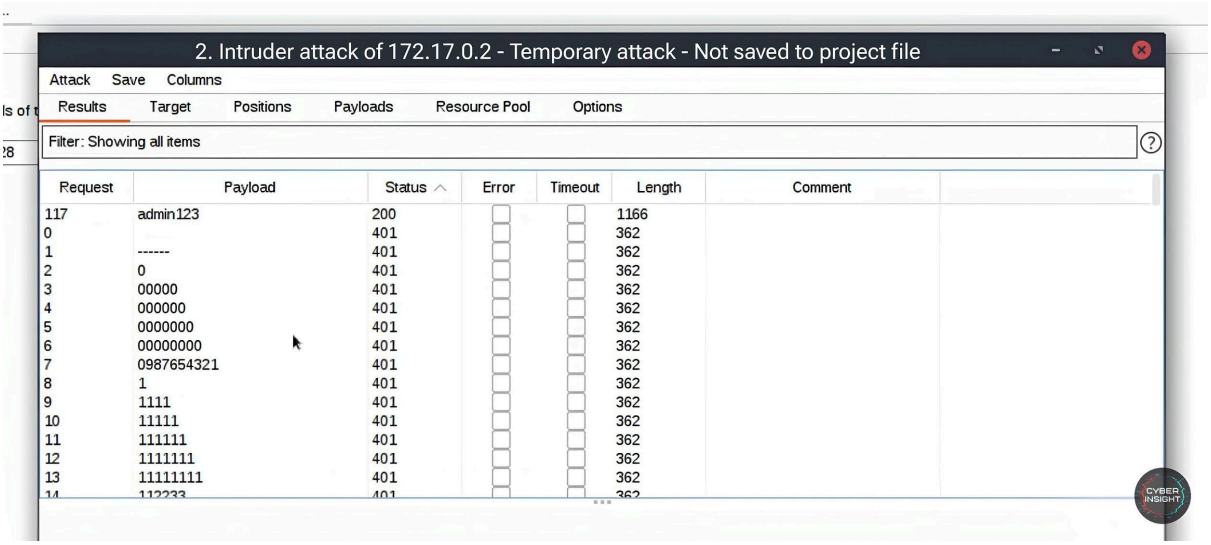
STEP 2: SEND TO INTRUDER



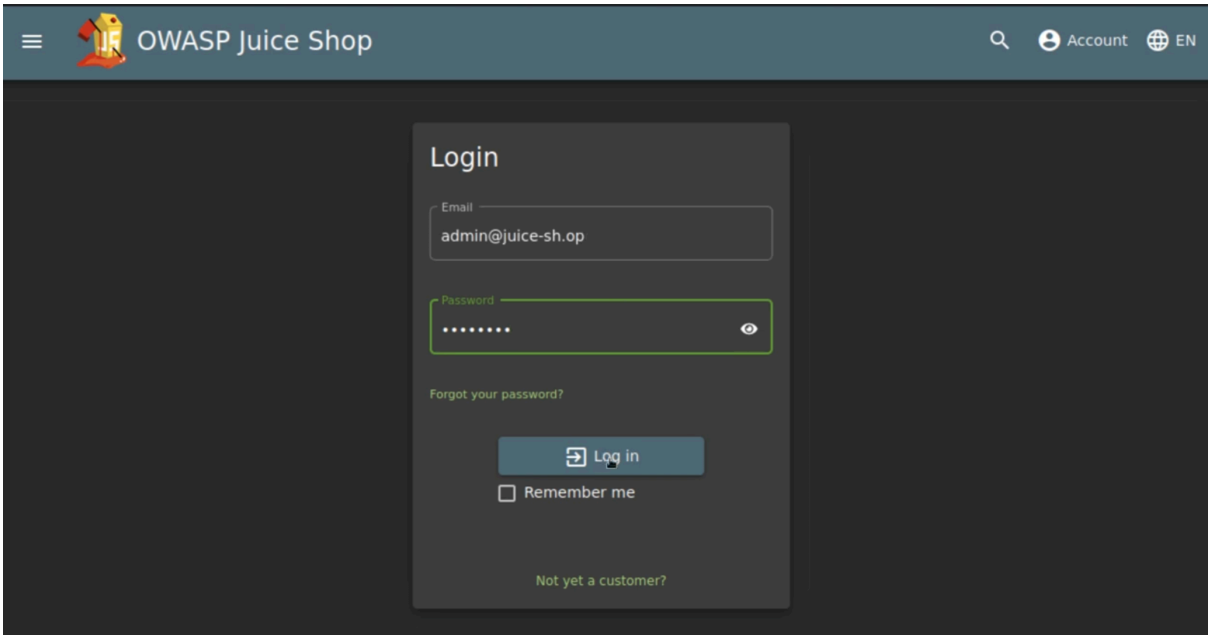
● This list comes from the Seclists package (installed using ***apt install seclists***) and contains common password guesses.

● The position marker was set on the password field, and the attack was initiated.

STEP3: PASSWORD BRUTEFORCED SUCCESSFULLY



STEP4: LOGIN USING THE PASSWORD



Detection Technique:

● A failed login attempt returned: 401 Unauthorized
● A successful login returned: 200 OK
● Once the status 200 OK appeared, I confirmed that the correct password was discovered and successfully logged in as Jim.

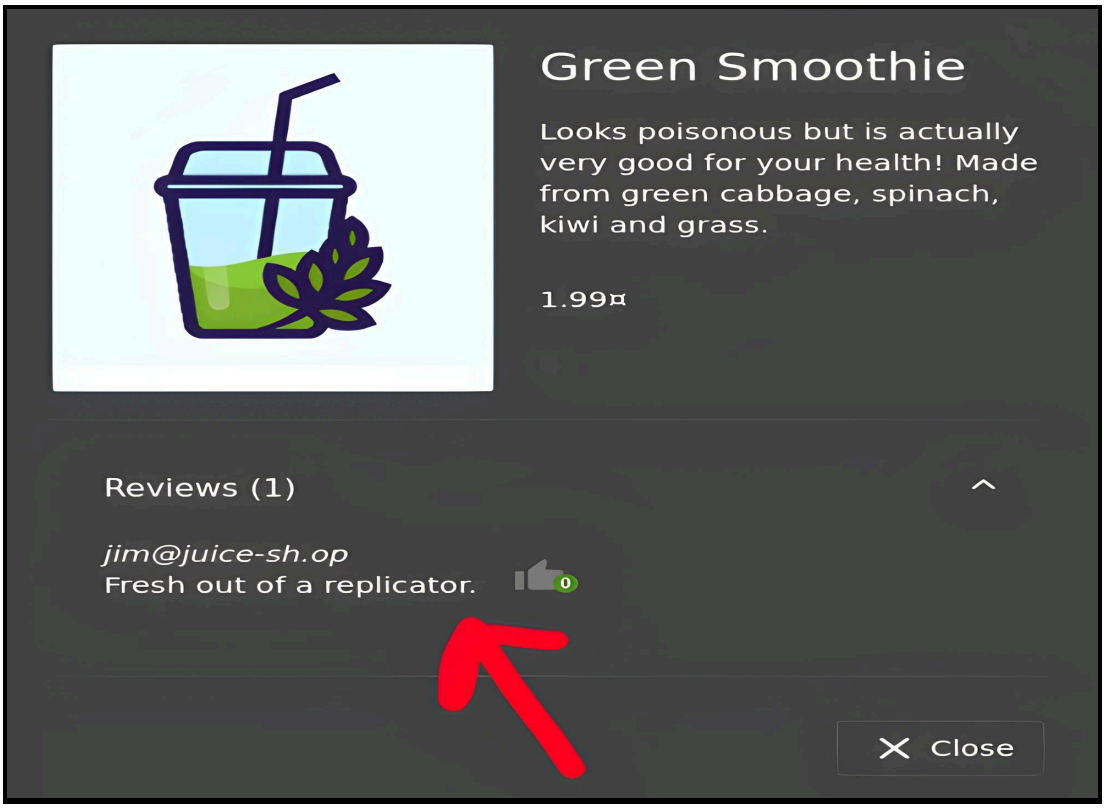
Phase 2: Bypassing Password Reset via Security Question:

Exploring further, I tested the Forgot Password feature using Jim's email. Upon submission, the application prompted a security question:

> "*Your oldest sibling's middle name?*"

● To solve this, I examined Jim's past activity — specifically a review he left on the Green Smoothie product. In his comment, he referenced a "**replicator**".

STEP1: FOUND A WORD REPLICATOR



● Curious about the term, I searched "**replicator**" online, which led me to "**Star Trek**", a sci-fi franchise where replicators are used for food synthesis.

STEP2: SEARCHED REPLICATOR ONLINE

Replicator

Star Trek

In Star Trek a replicator is a machine that can create things. Replicators were originally seen to simply synthesize meals on demand, but in later series much larger non-food items appear. The technical aspects of replicated versus "real" things is sometimes a plot element. Wikipedia

Created by: Gene Roddenberry

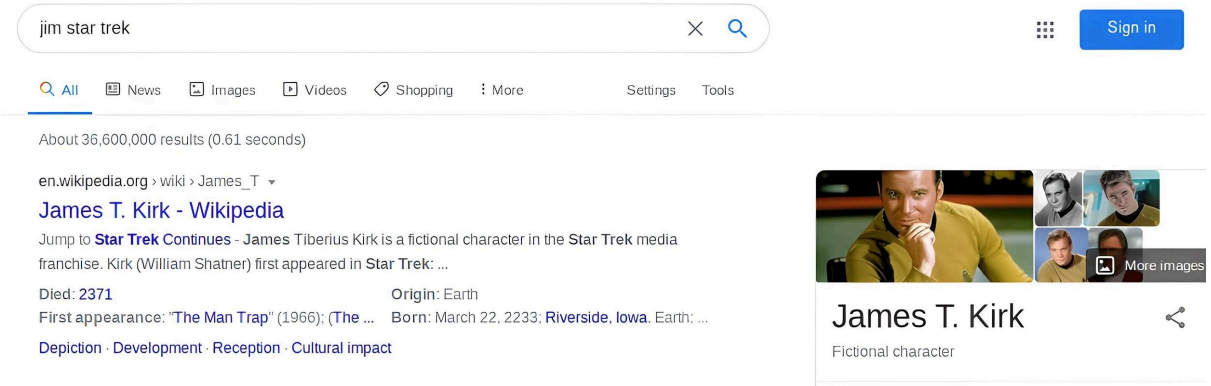
First appearance: Star Trek: The Next Generation

Function: Synthesis of organic and inorganic materials via rearrangement of subatomic particles

Feedback

● This clue led me to connect “**Jim**” with **James T. Kirk** from **Star Trek**. After reviewing his family background on a public wiki page, I found:

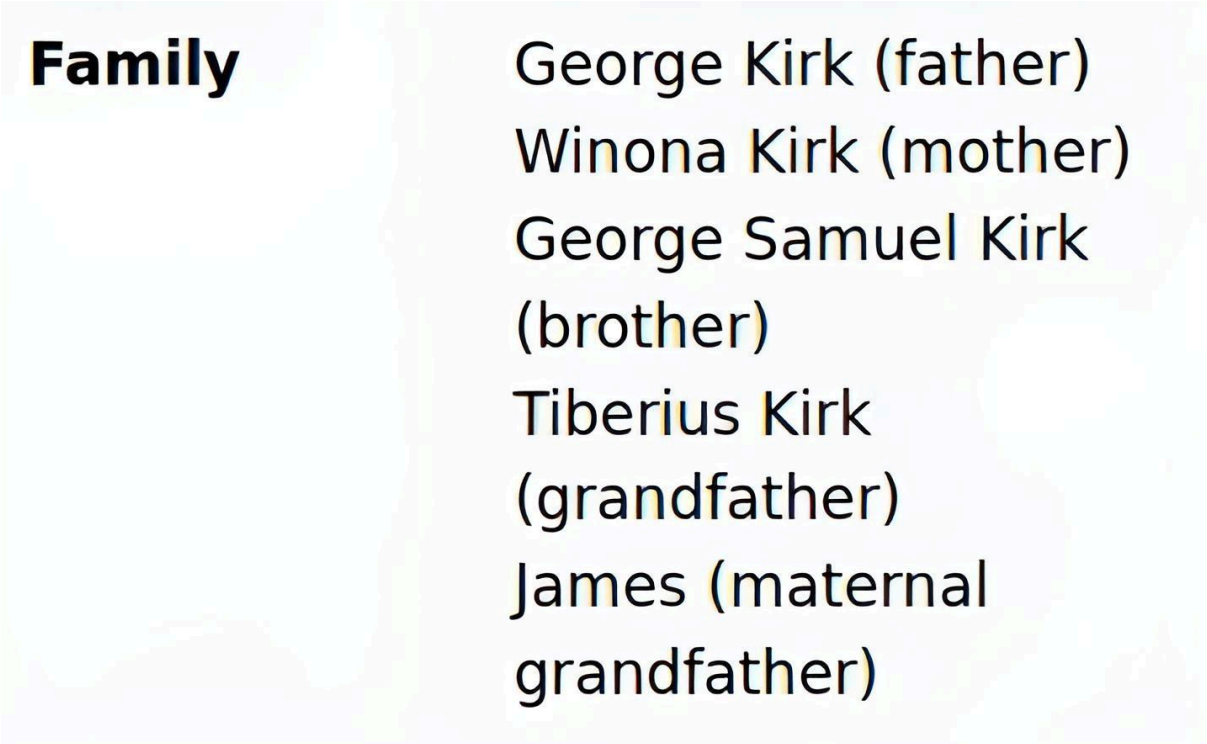
STEP3: SEARCHED JIM STAR TREK



Brother: George Samuel Kirk

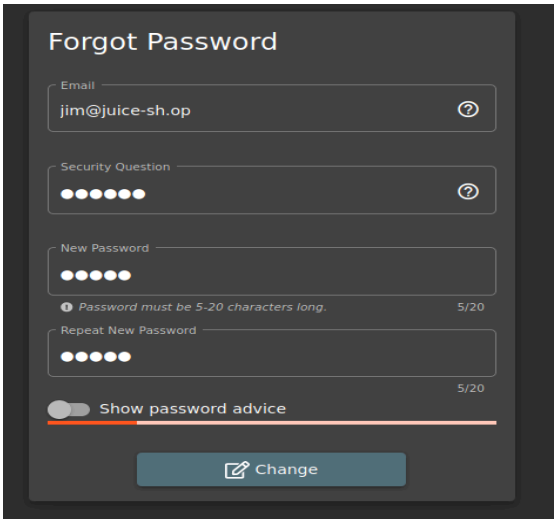
- ***This suggested that Samuel was likely the correct answer to the security question.***

STEP4: REVIEWED FAMILY DETAIL ONLINE



After submitting "**Samuel**", the application allowed me to reset Jim's password to any new value, fully confirming the Broken Authentication vulnerability via security question bypass.

STEP5: BYPASSED SECURITY QUESTION (PASS RESET SUCCESSFULLY)



Impact:

This vulnerability allows:
- Unauthorized login via brute-force
- Account compromise through weak or guessable security questions
- Full control of target user account (including password reset)

Mitigation Strategies:

Issue	Mitigation
Weak Login Mechanism	Implement rate-limiting and account lockout after multiple failed attempts
Insecure Forgot Password Flow	Use secure token-based password reset links with expiry and validation
Guessable Security Questions	Avoid using static security questions; use multi-factor authentication (MFA)
Lack of CAPTCHA	Implement CAPTCHA during login and reset flows to prevent automated abuse

OWASP Top 10 Mapping:

- **A01:2021 – Broken Access Control**
- **A07:2021 – Identification and Authentication Failures**
- ***FOR MORE INFORMATION ABOUT BROKEN AUTHENTICATION VISIT***

(Sensitive Data Exposure – Analysis and Exploitation)

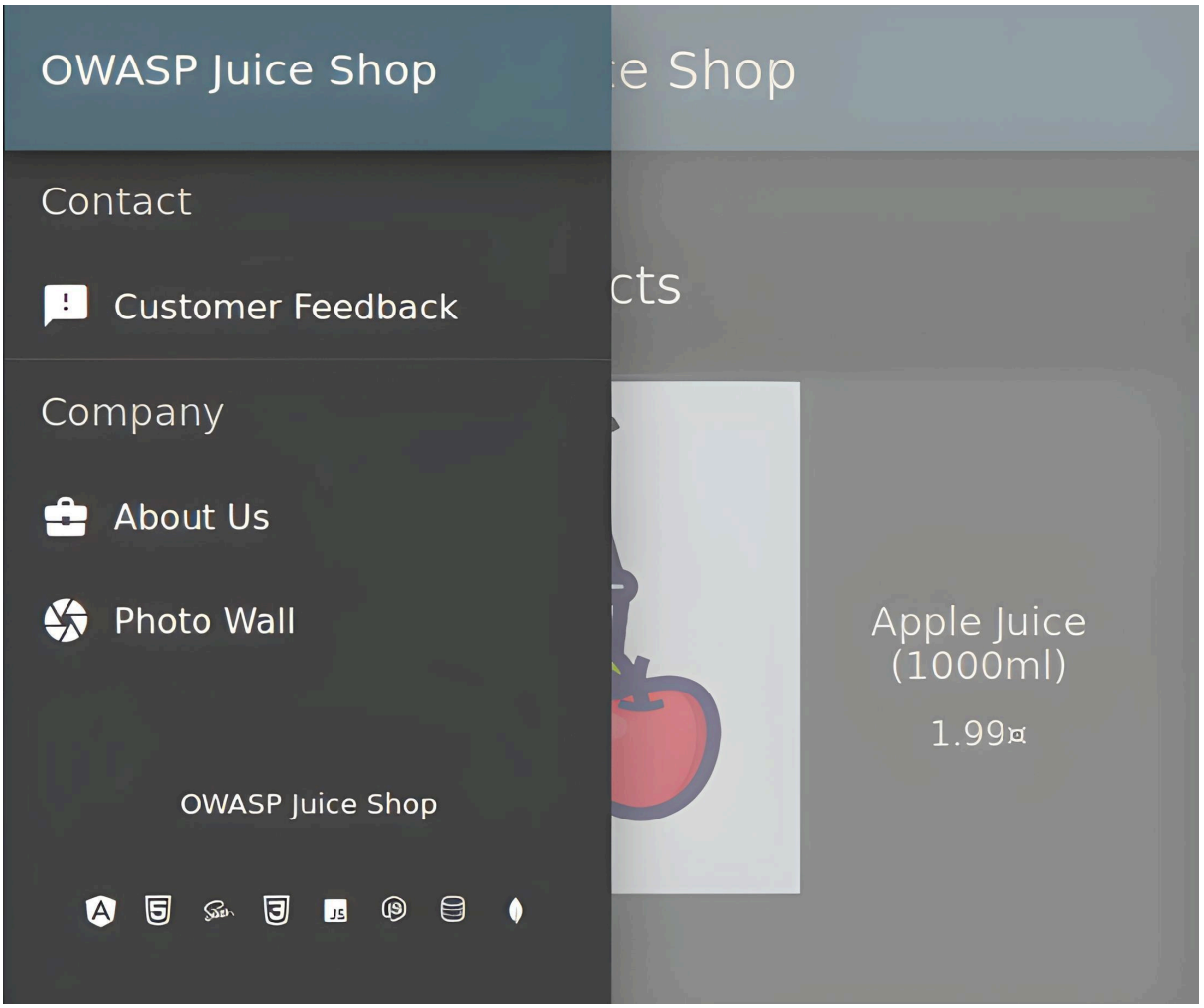
Overview:

Sensitive data exposure occurs when applications do not adequately protect data such as personal details, credentials, files, or internal documentation. In this case, I found that the OWASP Juice Shop instance hosted on my Docker IP (172.17.0.2) had publicly accessible sensitive files in an exposed directory.

Discovery Process:

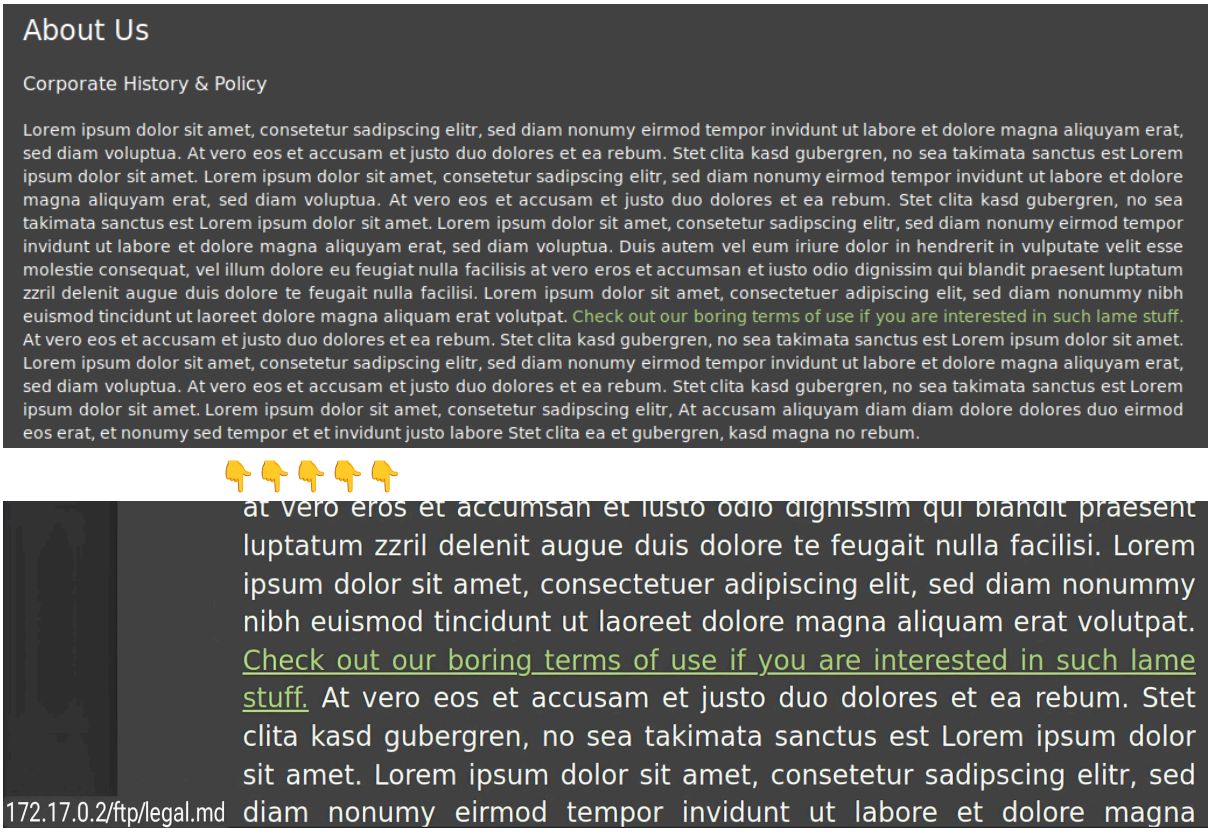
While browsing the application's "About Us" page, I noticed a hyperlink labeled: **Check out our terms of use**

STEP1: CLICK ON ABOUT US



● Hovering over the link revealed the following URL:
http://172.17.0.2:3000/ftp/legal.md

STEP2: HOVER OVER THE LINK

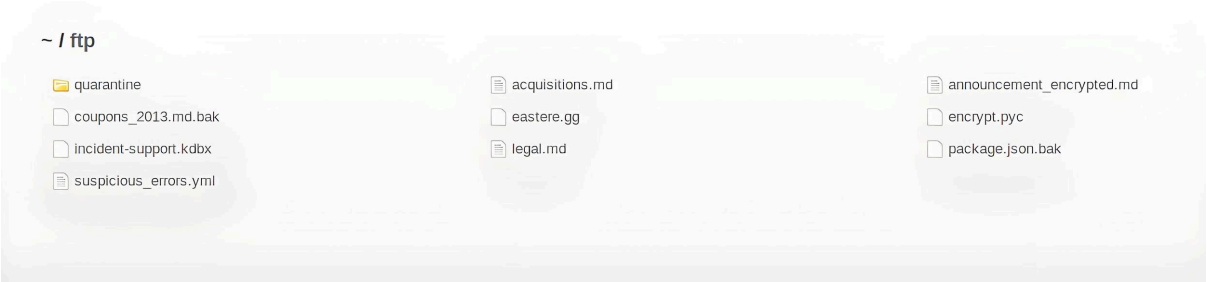


● This implied the existence of a publicly accessible /ftp/ directory.
● When I removed legal.md and directly navigated to: ***http://172.17.0.2:3000/ftp/*** then. I discovered that the entire /ftp/ directory was unrestricted, allowing anyone to list and download internal documents. This represents a serious data exposure flaw.

STEP3: GONE TO THIS LINK WITHOUT */legal.md*

● These files can potentially leak internal business logic, marketing strategies, or user behavioral patterns to attackers or competitors.

STEP4: SOME DATA EXPOSED



Files Identified in the /ftp Directory:

Filename	Description
`legal.md`	Markdown file containing legal disclaimers
`acquisitions.md`	Document referencing internal company acquisitions
`coupons_2013.md`	Possible historical data related to e-commerce coupon activity

● These files can potentially leak internal business logic, marketing strategies, or user behavioral patterns to attackers or competitors.

Impact:

● Attackers can access sensitive internal files without authentication.
● Potential for information gathering, recon, or targeted phishing.
● This can also expose configuration files, credentials, or private customer data if not mitigated.

Mitigation & Strategies:

Issue	Recommendation
Public exposure of `/ftp/`	Restrict directory browsing via server config (`.htaccess`, Nginx, etc.)
Sensitive files unprotected	Move internal documents to authenticated, access-controlled storage
Lack of encryption or controls	Enforce HTTPS-only delivery, disable indexing, and apply file access policies

OWASP Top 10 Mapping:

Category	Finding
A02:2021 – Cryptographic Failures	Files publicly exposed without any encryption or token-based access
A05:2021 – Security Misconfiguration	Directory browsing not disabled; sensitive files left in public paths

- ***FOR MORE INFORMATION ABOUT SENSITIVE DATA EXPOSURE VISIT***

Remarks:

This vulnerability highlights a classic case of insecure file exposure through forgotten or misconfigured directories. Despite being a simple issue, it can lead to high-risk consequences in real-world scenarios when internal documentation or credentials are accidentally exposed.

(Broken Access Control – Discovery and Exploitation)

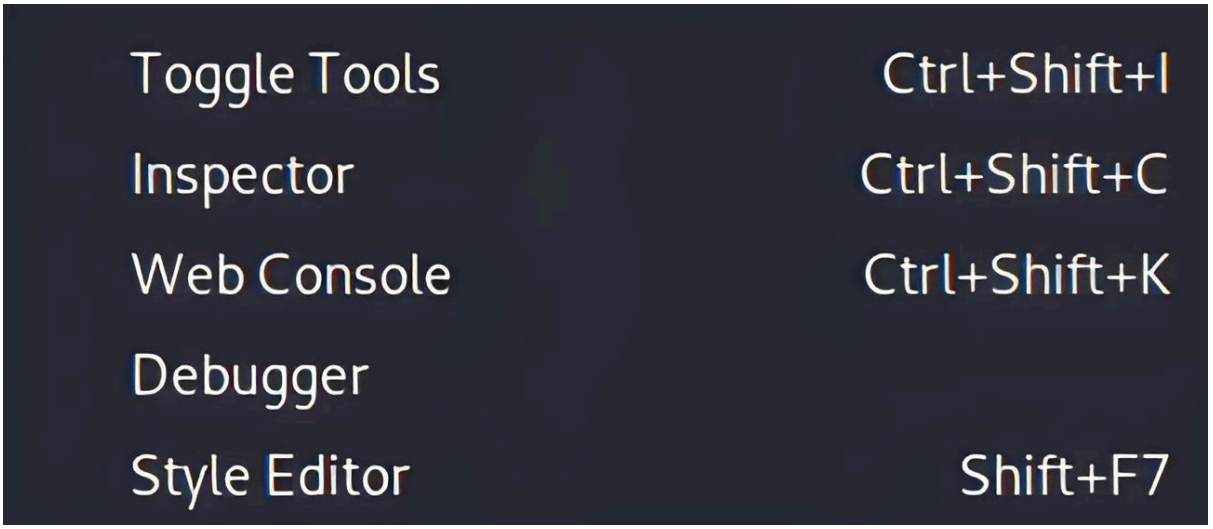
Overview:

Broken Access Control occurs when an application fails to enforce proper permissions on sensitive resources or actions. In this case, I discovered a hidden administration route by analyzing the JavaScript source files of the application — a clear case of vertical privilege escalation.

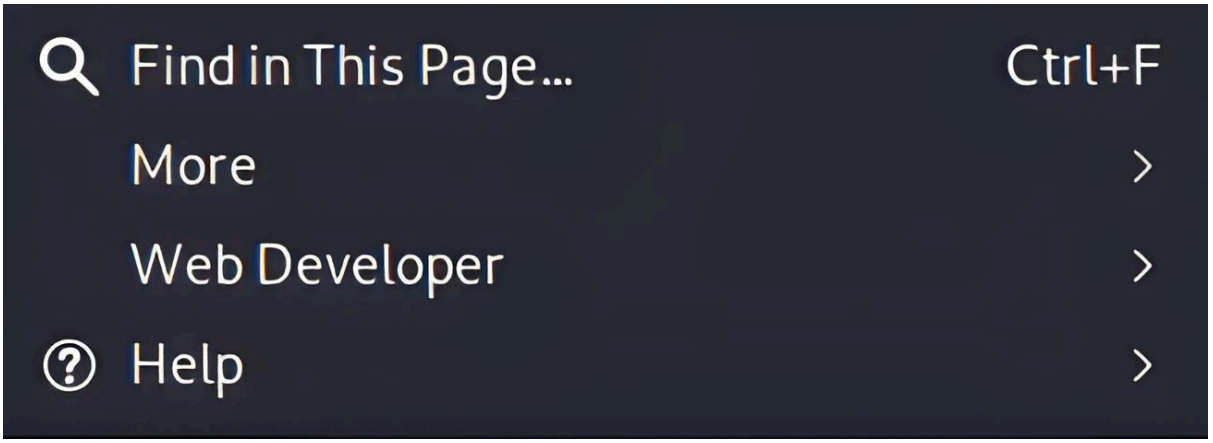
Discovery Method:

● I began by opening the OWASP Juice Shop instance in Firefox and launched the Developer Tools (Inspector > Debugger tab). After refreshing the application, I located a loaded JavaScript file named: ***main-es2015.js***

STEP1: CLICK ON DEBUGGER

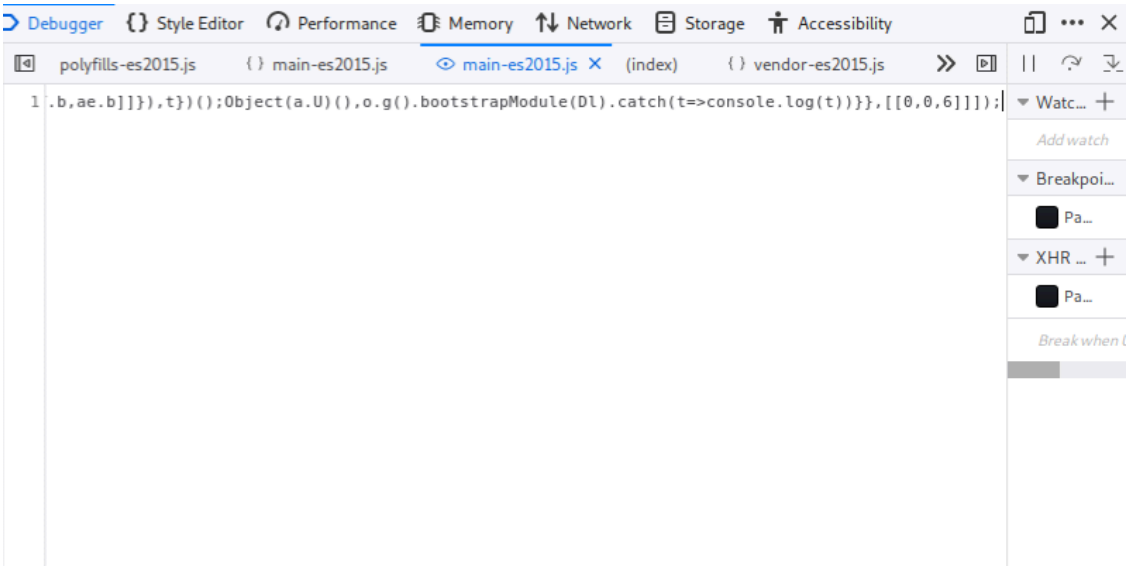


👇👇**CLICK ON WEB DEV** 👇👇



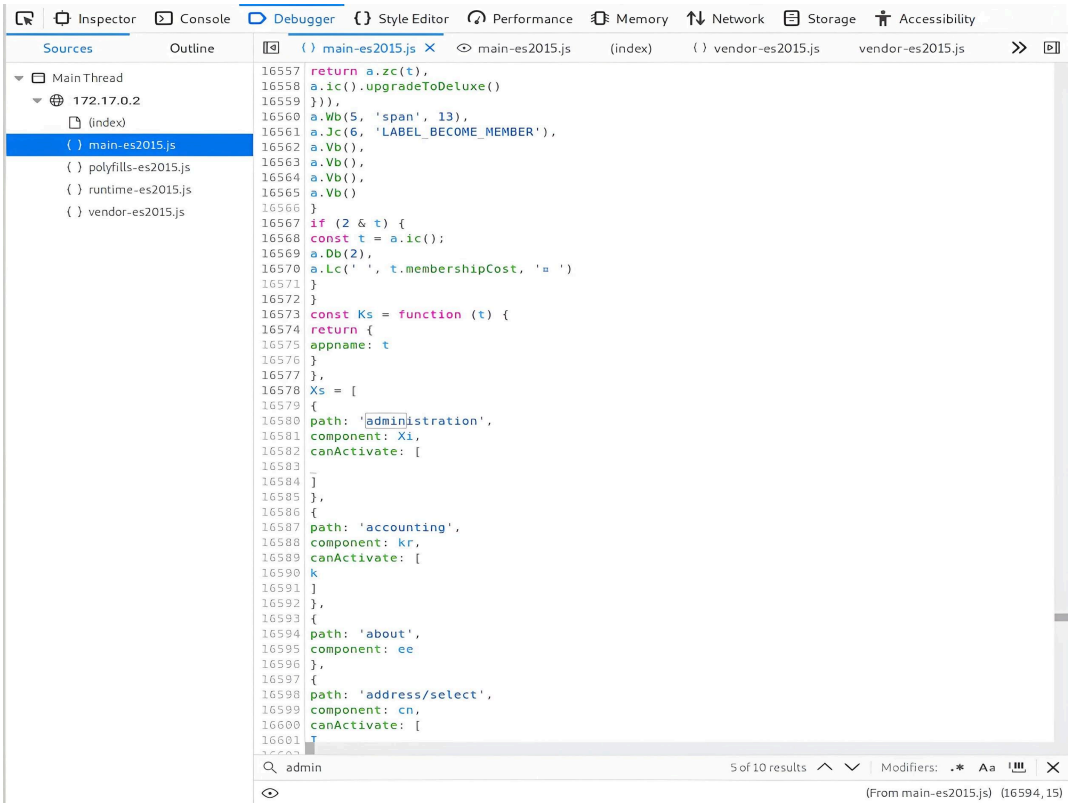
● Navigating directly to:
http://172.17.0.2:3000/main-es2015.js

STEP2: NAVIGATE TO *main-es2015.js in DEBUGGER*



● I opened the source code and searched for the keyword: **admin**
● This revealed the following route definition: path: 'administration'
● From this, I inferred that the hidden admin panel likely exists at:***http://172.17.0.2:3000/#/administration***

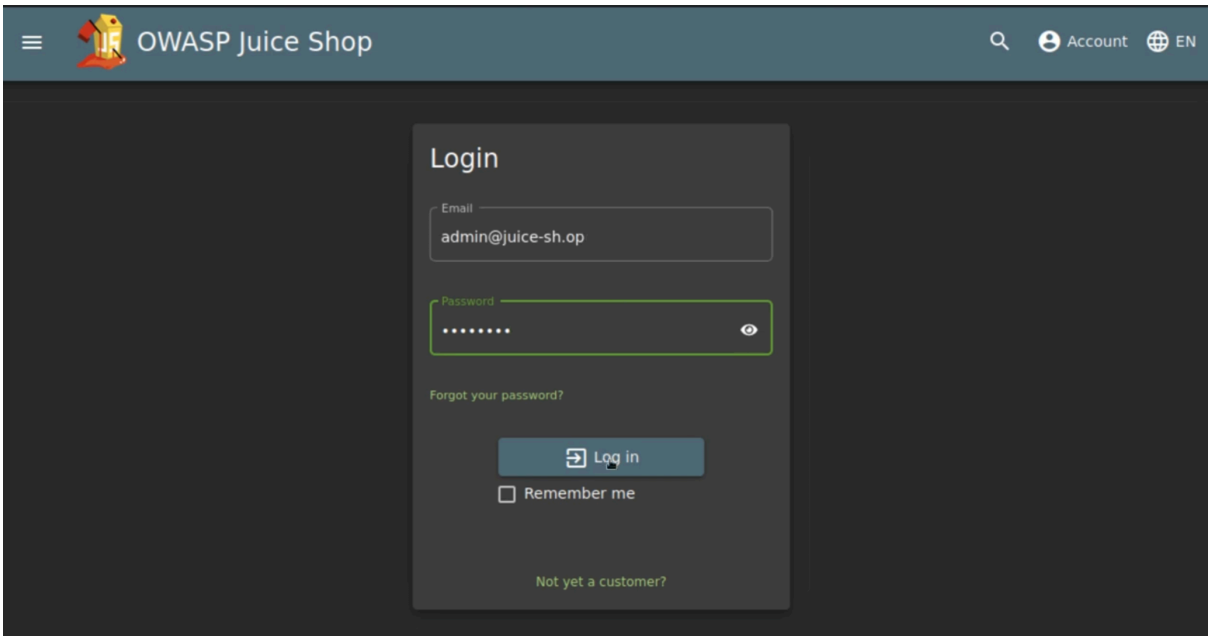
STEP3: SEARCH FOR ADMIN HERE IN SOURCE CODE



● Attempting to visit this URL as an unauthorized user initially failed, but when logged in using a valid account (found via SQL Injection attack earlier), the route successfully loaded — even though the user did not have admin privileges

.

STEP4: VISIT ADMIN by LOGGING IN FIRST



Impact:

● The admin panel becomes accessible to users without proper roles.
● This violates vertical privilege boundaries, allowing a lower-privilege user to interact with restricted functionality.

Exploit Type:

● **Exploit Category:** Vertical Privilege Escalation.
● **Description:** A lower-level user accessing an admin-specific page or action.

Mitigation Strategies:

Issue	Mitigation
Hidden but exposed admin route	Do not expose sensitive routes in client-side JS unless absolutely necessary
Lack of role-based access checks	Enforce proper role-based authorization server-side (e.g., isAdmin = true)
Unauthorized resource exposure	Deny access by default; grant based on validated session roles

OWASP TOP 10 Mapping:

Category	Finding
A01:2021 – Broken Access Control	Admin route accessible without proper role checks
A05:2021 – Security Misconfiguration	Sensitive routes exposed via frontend JavaScript bundle

- ***FOR MORE INFORMATION ABOUT BROKEN ACCESS CONTROL VISIT***

XSS(Cross Site Scripting)

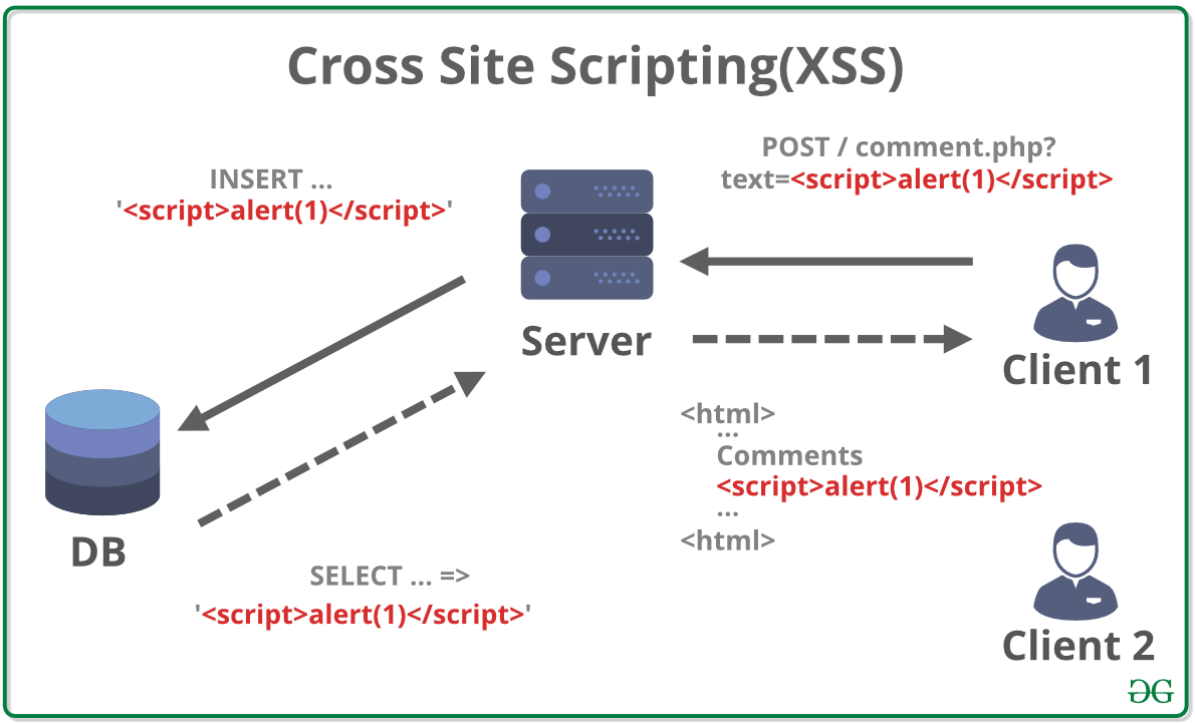
What is XSS?

Cross-Site Scripting (XSS) is a common and critical web application vulnerability that allows attackers to execute malicious JavaScript in the context of a user's browser. This can lead to:

- Credential theft
- Session hijacking
- Malware injection
- UI redressing (defacements)

XSS vulnerabilities are among the most frequently discovered issues during web application assessments. Their impact ranges from minor user annoyance to full account takeover, depending on the execution context and data exposure.

HOW CROSS SITE SCRIPTING HAPPENS:



Types of XSS:

Type	Description
DOM-Based XSS	Occurs when JavaScript in the client-side code modifies the DOM using unsanitized user input. Payloads are often injected through `<script>` tags.
Persistent XSS	Also known as stored XSS, this type occurs when user input is permanently stored on the server (e.g., in databases or comment sections) and executed later.
Reflected XSS	The injected script is reflected off a server, usually in error messages, search results, or headers. It executes immediately without being stored.

1. DOM-Based Cross-Site Scripting (XSS) – Exploitation

Vulnerability Type:

DOM-Based XSS (also referred to as XFS – Cross-Frame Scripting)

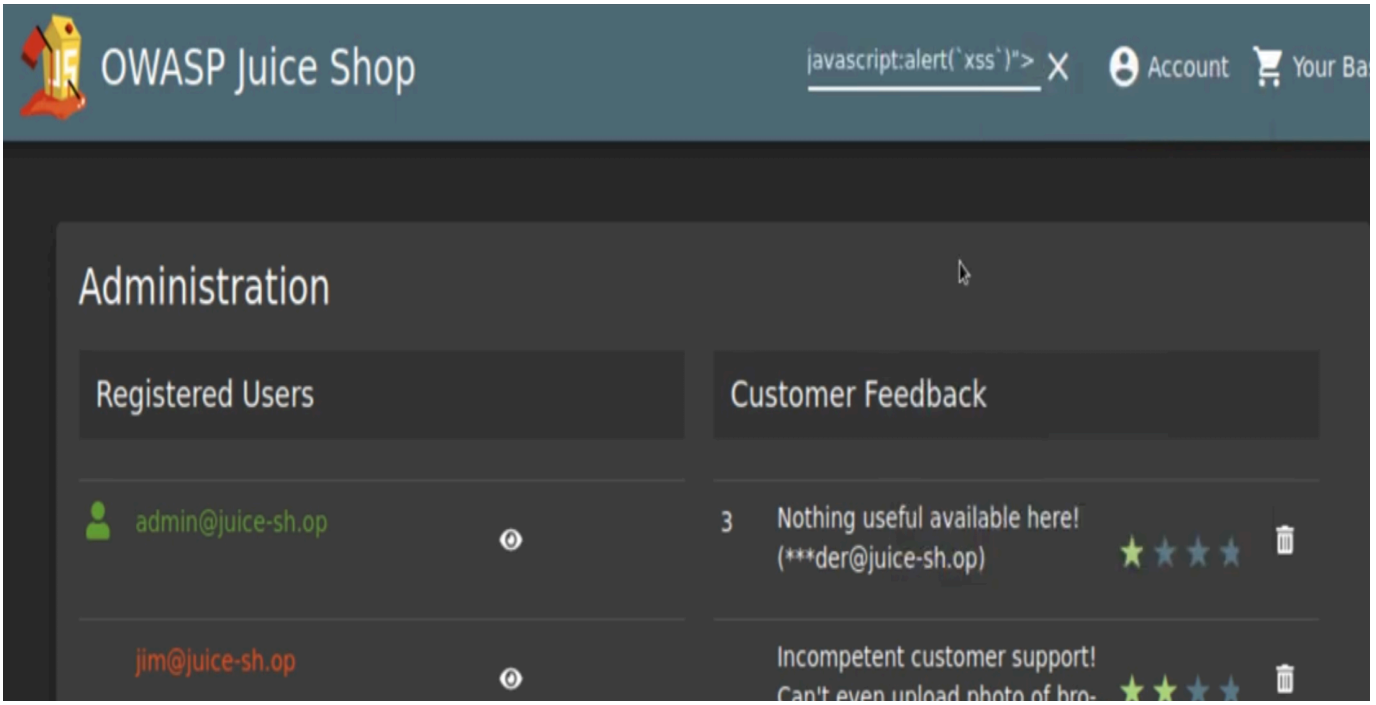
Discovery & Exploit Process

● While testing the search functionality of OWASP Juice Shop, I discovered a vulnerability that allows client-side injection of arbitrary JavaScript. The flaw lies in how the application handles user input and reflects it back into the DOM without proper sanitization.

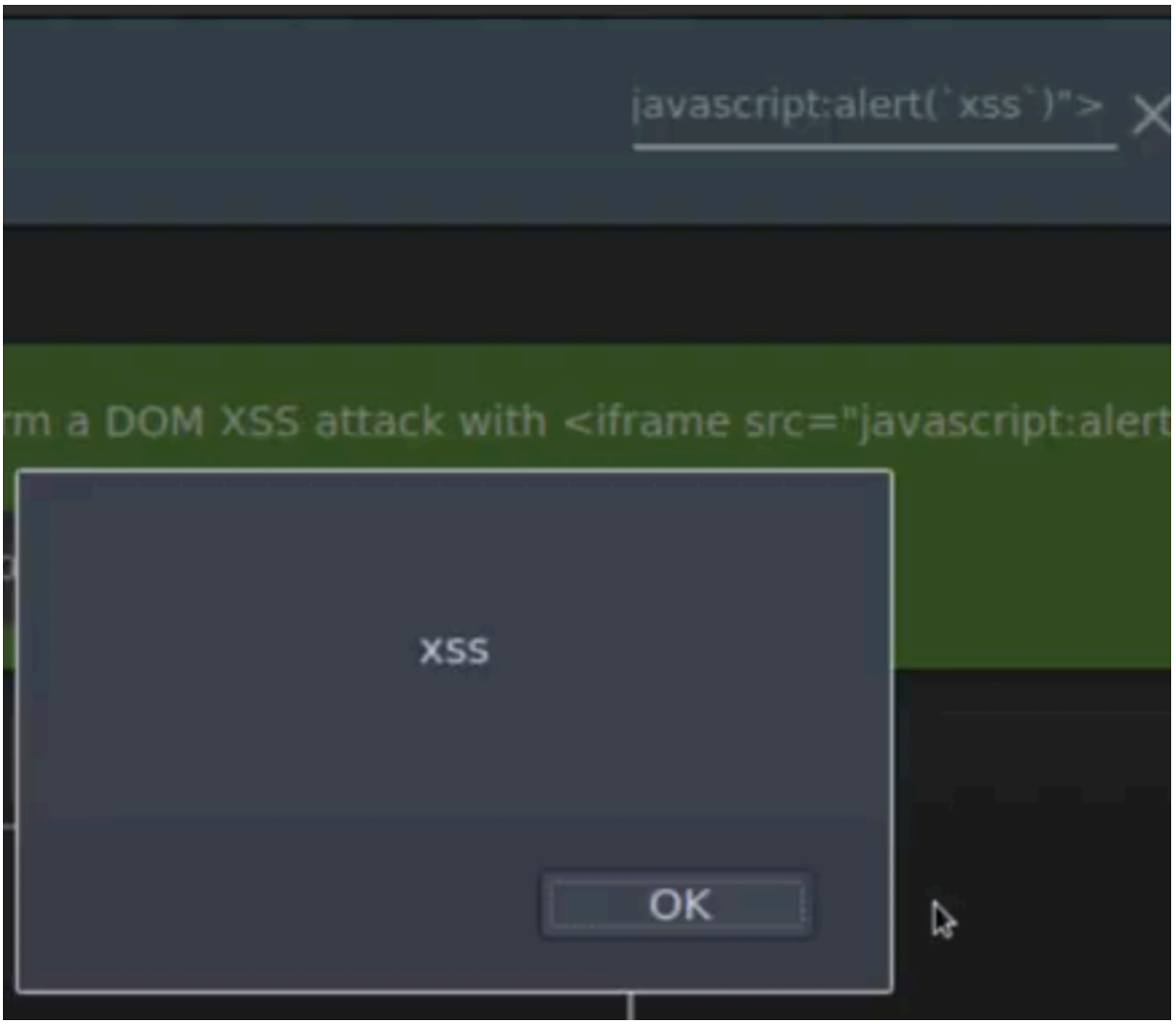
● To confirm the vulnerability, I used the following payload: ***<iframe src="javascript:alert('xss')">***

● This payload was injected directly into the search bar. Upon submission, the page returned a visible JavaScript alert, confirming that the script was executed in the browser context — a clear sign of DOM XSS

.**Step1: INJECT PAYLOAD DIRECTLY INTO SEARCH BAR**



Step2: CLEAR SIGN OF DOM XSS



Why This Worked:

The search feature sends the user's input back into the page without performing any input validation or output encoding. As a result:

- The injected iframe element was rendered on the page.
- The src="javascript:..." was interpreted and executed by the browser.
- The alert('**xss**') code executed successfully — proving DOM manipulation through user input.

Mitigation Strategies:

Issue	Recommended Fix
DOM XSS via iframe	Always **sanitize and encode** user input before inserting into the DOM
Unsafe iframe behavior	Implement **Content Security Policy (CSP)** to block inline scripts and `javascript:` URLs
No input validation	Apply strict input filtering using libraries like DOMPurify or server-side regex checks

OWASP TOP 10 Finding:

Vulnerability	OWASP Category
DOM-Based XSS	A03:2021 – Injection
XFS (Cross-Frame)	A07:2021 – Cross-Site Scripting (XSS)

Remarks:
● This DOM-based XSS vulnerability proves how dangerous unfiltered user input can be — especially when directly manipulated inside the browser's DOM. By injecting an iframe with a javascript: source, attackers can run arbitrary code, steal cookies, or redirect users to malicious pages.

2. Persistent Cross-Site Scripting (Stored XSS) – Exploitation

Vulnerability Type:

● Persistent XSS (Stored Cross-Site Scripting) via HTTP Header Injection

Discovery & Exploit Process:

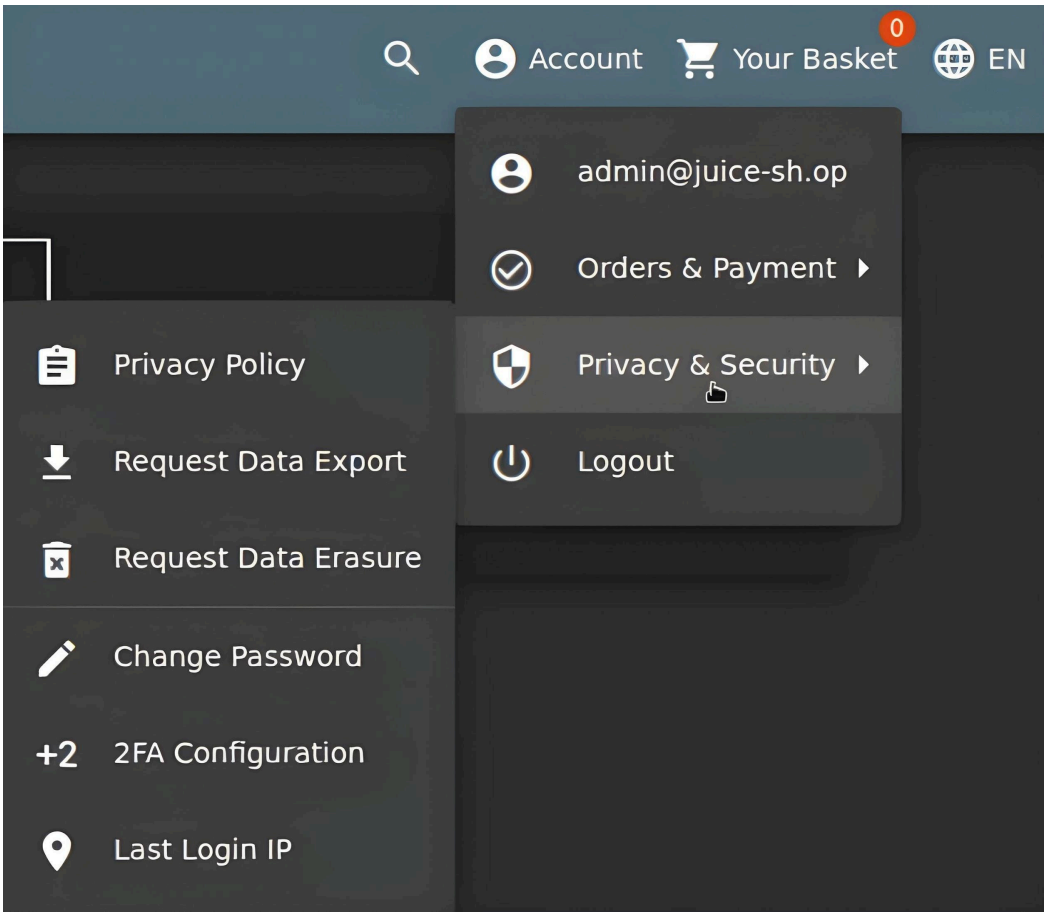
● While testing session logging features in OWASP Juice Shop, I discovered that the application logs the user's IP address upon login and displays it later in the "Last Login IP" section of the admin dashboard.

This data is rendered into the page without sanitization, making it a prime target for Persistent XSS — where the payload is stored on the server and executed when the page is viewed later.

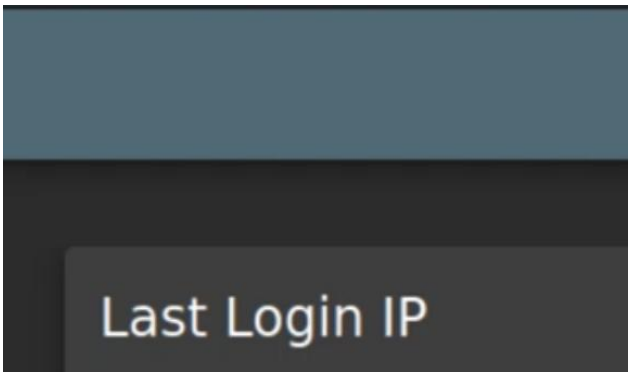
Steps to Exploit:

1. Logged into the admin account to access the IP logging section.

STEP1: GO TO PRIVACY THEN LAST LOGIN

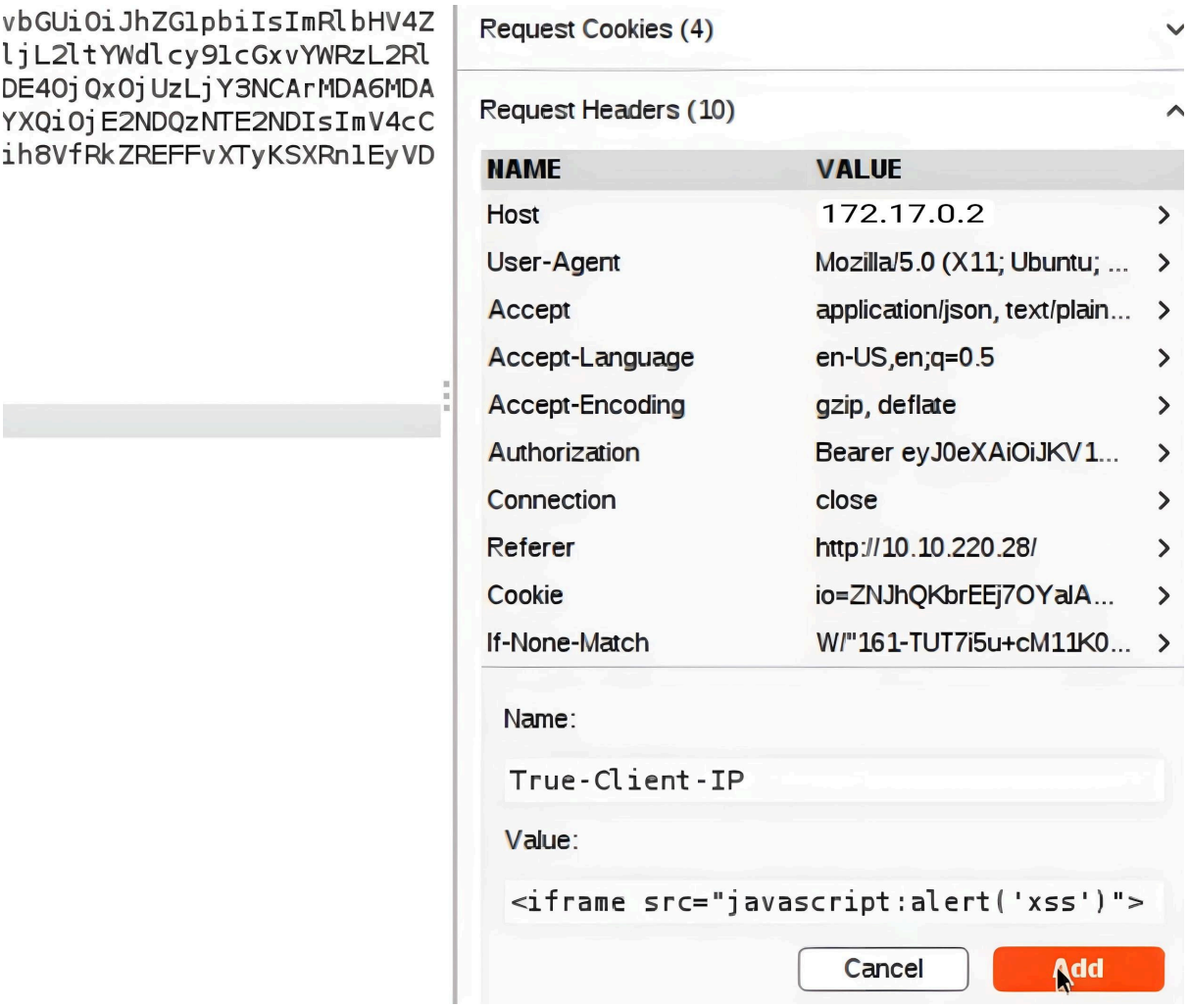


👇👇👇👇**VIEW YOUR LAST IP**👇👇👇👇👇👇👇👇👇



2. Verified that the "***Last Login IP***" was set to a default such as **0.0.0.0** or **127.0.0.1** or **172.17.0.2**
3. Logged out to trigger a new IP logging event on the next login.
4. With Burp Suite's ***Intercept enabled***, I captured the login request.

STEP2: INTERCEPT USING BURP SUITE

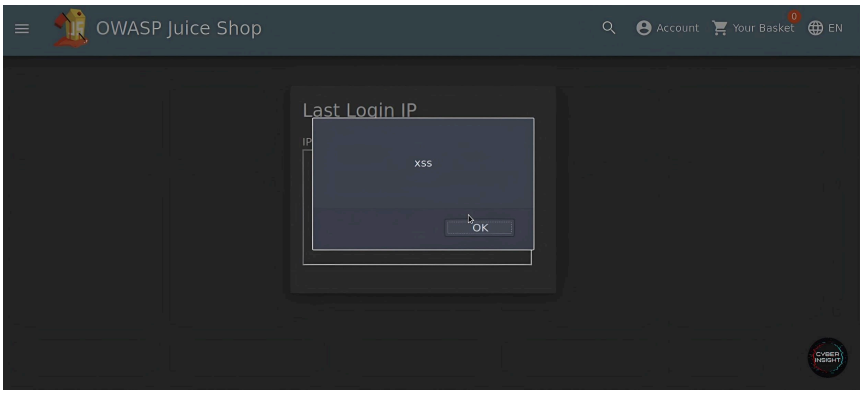


5. In the Headers tab, I injected the following XSS payload into the True-Client-IP header: ***True-Client-IP:<iframesrc="javascript:alert('XSS')">***

6. Forwarded the request to the server.

7. Logged back in as admin and navigated to the "***Last Login IP***" section.

STEP3: LOGGING IN AGAIN GOT (PERSISTENT XSS)



8. The malicious JavaScript was executed automatically, confirming the presence of ***Persistent XSS***.

Why Does This Work?

● The application trusted and stored the value of the True-Client-IP header — which is commonly used for identifying the originating client IP behind proxies. However, it failed to sanitize this header before rendering it in the admin dashboard.
● This allowed an attacker to store JavaScript code that executes in the browser of any user (admin in this case) viewing the "Last Login IP" section.

Mitigation Strategies:

Issue	Recommended Fix
Stored XSS in HTTP headers	Sanitize and encode all user-controllable headers before rendering on the frontend
Improper IP handling	Avoid trusting client-supplied IP headers without validation
No output encoding	Use output encoding libraries to prevent raw HTML injection

OWASP TOP 10 Finding:

Vulnerability	OWASP Category
Persistent XSS via Header	A03:2021 – Injection
Admin panel auto-execution	A07:2021 – Cross-Site Scripting (XSS)

Reflected Cross-Site Scripting (XSS) – Exploitation

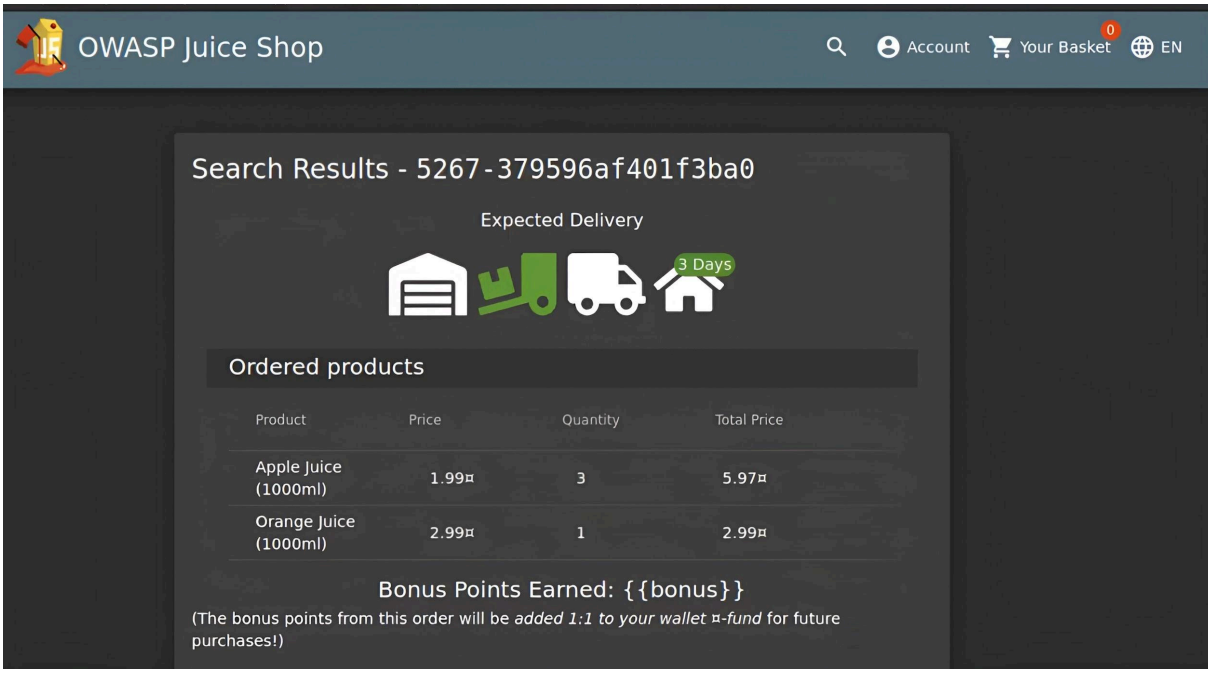
Vulnerability Type:

Reflected XSS (Client-Side) via Unsanitized URL Parameter

Discovery & Exploit Process:

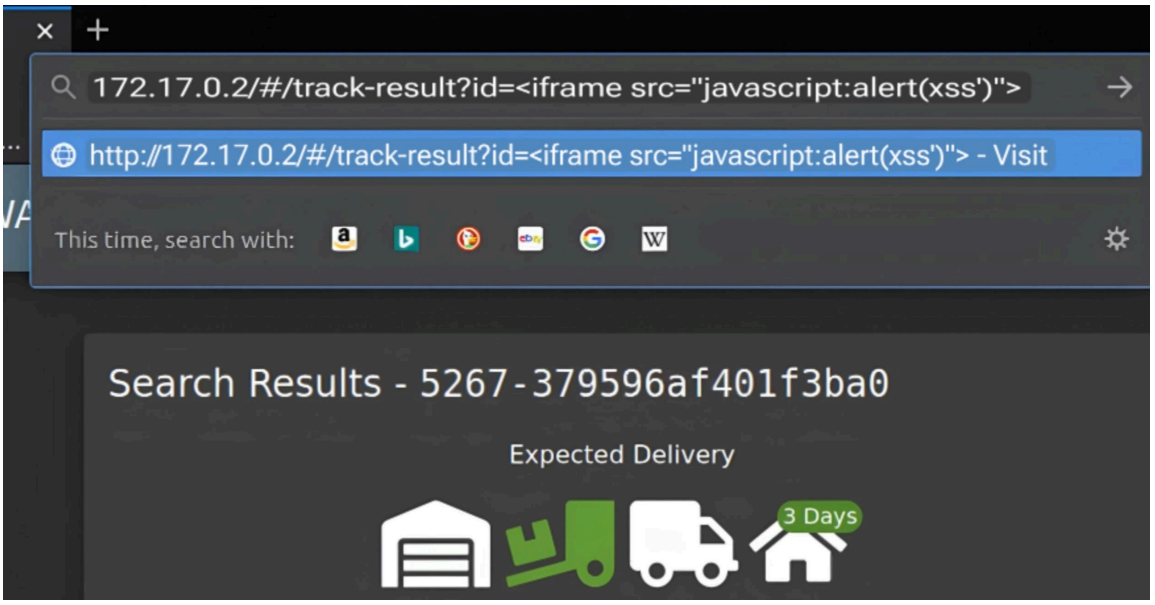
● While exploring features available to the admin account, I navigated to the Order History section of OWASP Juice Shop.

STEP1: Order History section of OWASP Juice Shop



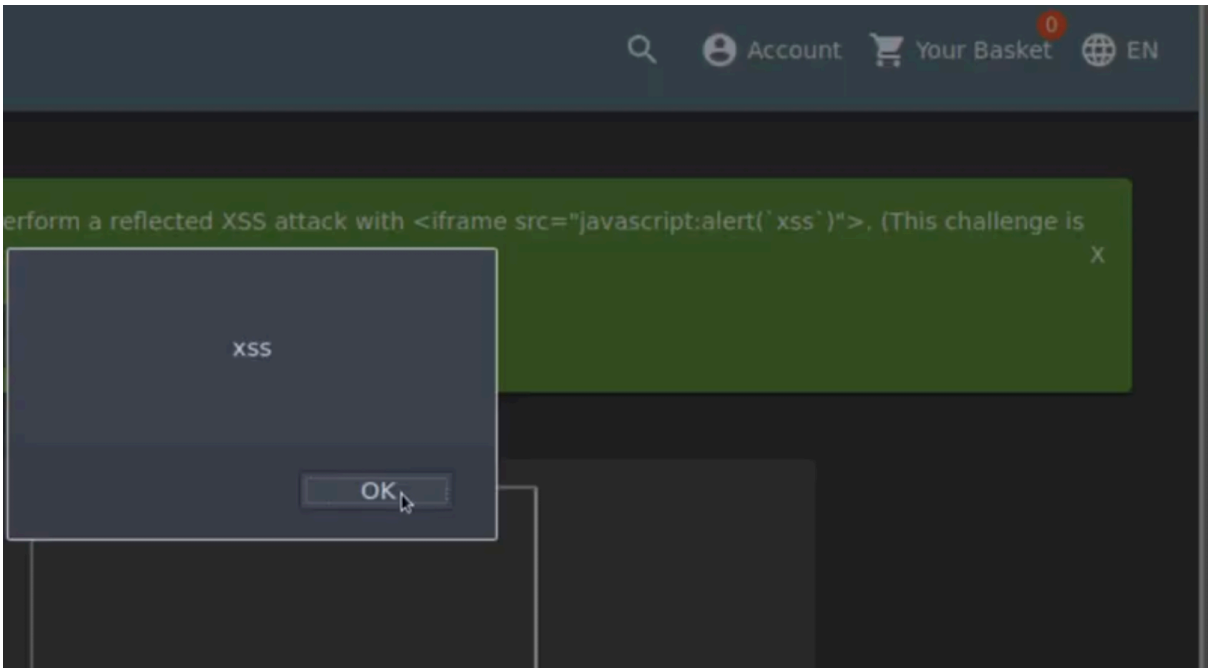
● Each order in this section had an associated tracking ID accessible by clicking on the “truck” icon. This redirected to a URL resembling:
http://172.17.0.2:3000/#/track-result?id=5267-f73dcd000abcc353
● This id parameter looked like a potential candidate for injection. To test for Reflected XSS, I replaced the order ID value with the following payload: ***<iframe src="javascript:alert('xss')">***
● Updated URL: ***http://172.17.0.2:3000/#/track-result?id=<iframe src="javascript:alert('xss')">***

STEP2: MAKE CHANGES IN URL



After refreshing the page, a JavaScript alert box was triggered, proving successful reflected XSS.

STEP3: REFRESH THE PAGE THAT'S ALL



Why Does This Work?

The application reflects the id parameter value directly into the DOM without sanitization or encoding. Because it failed to validate the content before rendering, injecting HTML/JS code caused the browser to execute it immediately upon loading the page. This behavior confirms a client-side reflected XSS, where user input is echoed back and executed without being stored server-si

Mitigation Strategies:

Issue	Recommended Fix
Unsanitized URL Parameters	Apply input validation and output encoding for all query string values
Executable payloads allowed	Implement proper escaping/encoding before inserting into HTML
Lack of CSP headers	Use **Content Security Policy** to block inline scripts and `javascript:` schemes

OWASP TOP 10 Finding:

Vulnerability	OWASP Category
Reflected XSS	A03:2021 – Injection
Client-side echo	A07:2021 – Cross-Site Scripting (XSS)

FOR MORE INFORMATION ABOUT XSS(CROSS SITE SCRIPTING) VISIT

Remarks:

Reflected XSS is one of the most commonly exploited vulnerabilities in web apps due to its simplicity and impact. In this case, exploiting the vulnerable id parameter in the tracking URL allowed me to execute JavaScript without any prior authentication bypass or stored interaction.
This issue could be weaponized to steal admin session tokens or redirect users to malicious payloads if delivered via crafted phishing URLs.

(CONCLUSION)

Through this project, I performed a comprehensive security assessment of the OWASP Juice Shop web application using industry-standard tools and techniques. The testing process revealed multiple critical vulnerabilities including:

- SQL Injection
- Cross-Site Scripting (XSS)
- Broken Authentication
- Sensitive Data Exposure
- Broken Access Control

Each vulnerability was identified, validated, and documented with proper mitigation strategies and OWASP Top 10 mappings.

This assessment not only strengthened my understanding of web application security but also gave me hands-on experience in ethical hacking, vulnerability analysis, and reporting.