

|  |  |
| --- | --- |
| **QUALIFICATION :** Bachelor of Computer Science in Cyber Security | |
| **QUALIFICATION CODE:** 07BCCS | **LEVEL: 6** |
| **COURSE**: Web Application Security | **COURSE CODE**: WAS621S |
| **DUE DATE: 30 September 2025** | **PAPER:** Semester Project |
| **DURATION: N/A** | **MARKS: 100** |

|  |  |
| --- | --- |
| **STUDENT NAME:** | **STUDENT NUMBER:** |
| **Robin Y. Kuwanayo** | **220055114** |
| **James Smeer** | **220124329** |
| **Daniel Stephanus** | **200866745** |
| **Frieda Niikondo** | **222072903** |
| **Moses Naitembu** | **220021430** |
| **EXAMINER(S)** | **Mrs. Viktoria Shakela** |
| **MODERATOR:** | **Mr. Edward Nepolo** |

# **Table of Contents**

1. **Executive Summary**
2. **Methodology**
3. **Findings  
    3.1 SQL Injection (GET /rest/products/search)  
    3.2 Weak Password Policy (POST /rest/user/login)  
    3.3 Unnecessary Open Ports  
    3.4 Missing Security Headers**
4. **Recommendations**
5. **Verification & Retesting Plan**
6. **Assignment Mapping  
    6.1 Set up of Vulnerable App (1.1)  
    6.2 Test Cases & Evidence (1.2)  
    6.3 Remediation – SQL Injection (1.3)**
7. **Conclusion**

Penetration Testing Report: OWASP Juice Shop Web Application

# 1. Executive Summary

This report presents the findings of a penetration test conducted on the OWASP Juice Shop web application hosted on localhost. The objective was to identify and document security vulnerabilities, assess their severity, and recommend remediation strategies. The test revealed critical issues including SQL injection and weak password policies, which pose significant risks to data integrity and user authentication. Remediation steps have been proposed to enhance the application's security posture.

# 2. Methodology

The penetration test followed a structured approach using industry-standard tools and techniques:

- ***Reconnaissance:*** Nmap was used to identify open ports and services.  
- ***Vulnerability Assessment:*** Hydra and sqlmap were employed to test authentication and input validation.  
- ***Exploitation:*** SQL injection was confirmed and database schema was extracted.  
- ***Documentation:*** All findings were recorded with evidence and recommendations.

# 3. Findings

## 3.1 SQL Injection Vulnerability

**Endpoint:** GET /rest/products/search?q=apple  
**Tool:** sqlmap  
**Details:** The 'q' parameter is vulnerable to boolean-based blind SQL injection. Sqlmap successfully enumerated the backend SQLite database and extracted table names including 'Users', 'Products', and 'Cards'.

**Severity:** High

**Impact:** Unauthorized access to sensitive data, potential for full database compromise.  
 **Penetration Test Vulnerability Summary**

This table summarizes the vulnerabilities identified during the penetration test of the OWASP Juice Shop web application. Each entry includes the vulnerability name, severity classification, supporting evidence, and potential impact.

|  |  |  |  |
| --- | --- | --- | --- |
| **Vulnerability** | **Severity** | **Evidence/Details** | **Impact** |
| SQL Injection (GET /search) | High | sqlmap confirmed boolean-based blind SQL injection on parameter 'q'; backend DBMS is SQLite; tables enumerated. | Data theft, privilege escalation, full database compromise. |
| Weak Password Policy | High | Hydra tested common passwords for admin login; no lockout or MFA detected. | Account compromise, unauthorized access. |
| Multiple Open Ports | Medium | Nmap scan revealed open ports: 80, 3000, 3306, 45833; some services unrecognized. | Increased attack surface, potential lateral movement. |
| Missing Security Headers | Medium | Nmap HTTP fingerprinting showed absence of headers like Content-Security-Policy and Strict-Transport-Security. | Susceptibility to clickjacking, XSS, and other web attacks. |

## 3.2 Weak Password Policy

**Endpoint:** POST /rest/user/login  
**Tool:** Hydra  
**Details**: Brute-force attempts using common passwords for the user 'admin@juice-sh.op' were performed. Although no successful login was achieved, the lack of account lockout and weak password enforcement increases risk.

**Severity:** High

**Impact:** Potential unauthorized access through credential stuffing or brute-force attacks.

## 3.3 Unnecessary Open Ports

**Tool:** Nmap  
Details: The scan revealed open ports including 80 (Apache), 3000 (Node.js), 3306 (MySQL), and 45833 (Golang HTTP server). Some services are not essential for public access and increase the attack surface.

**Severity**: Medium

**Impact**: Exposure to service-specific vulnerabilities and lateral movement opportunities.

## 3.4 Missing Security Headers

**Tool:** Nmap HTTP fingerprinting  
**Details**: HTTP responses lack headers such as Content-Security-Policy and Strict-Transport-Security. This omission can lead to clickjacking, XSS, and other client-side attacks.

**Severity:** Medium

**Impact:** Increased risk of client-side exploitation.

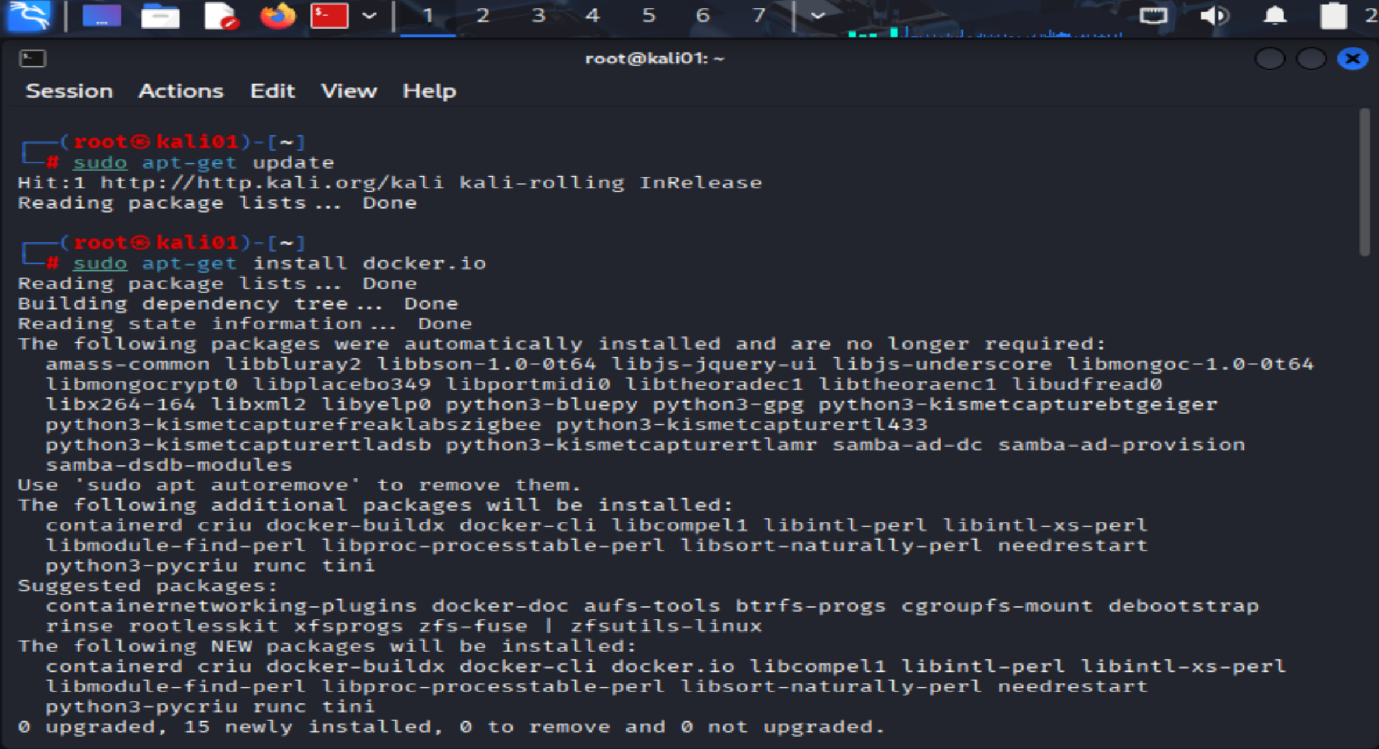
# 4. Recommendations

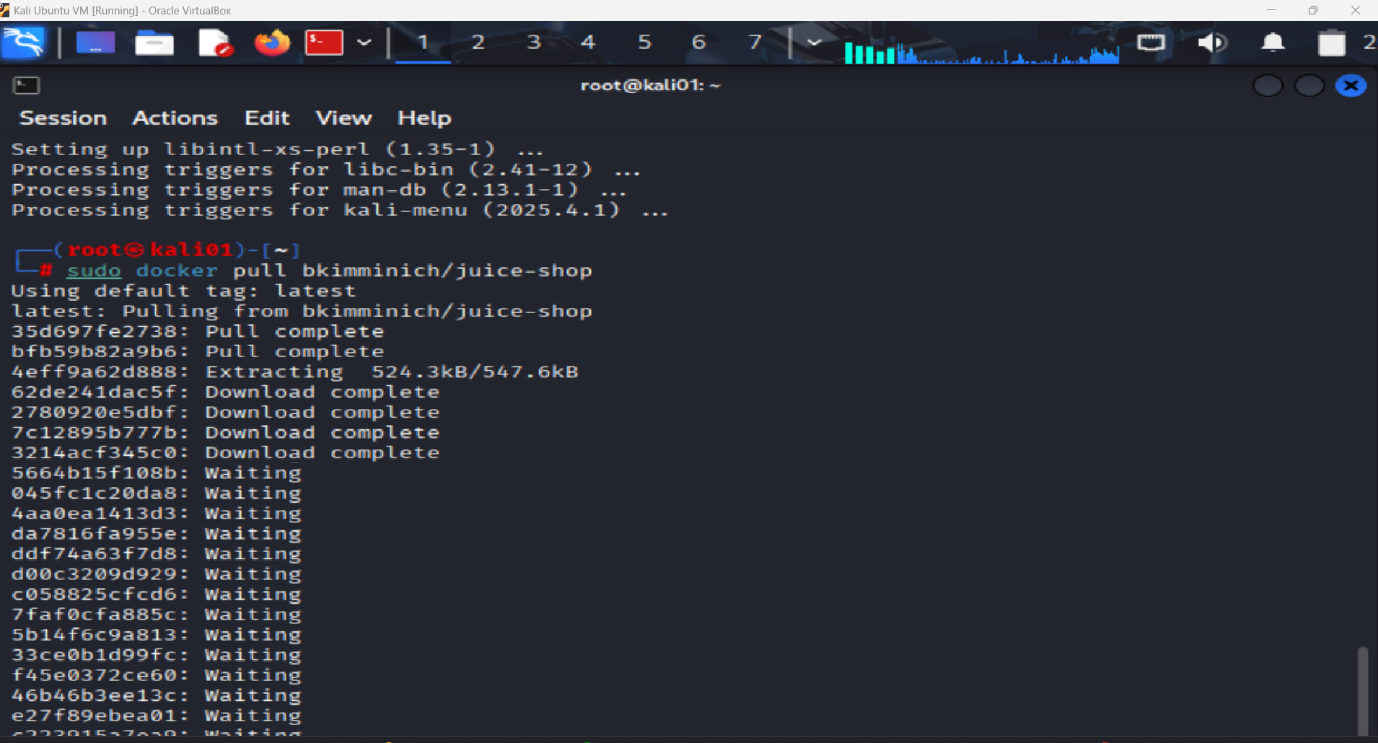
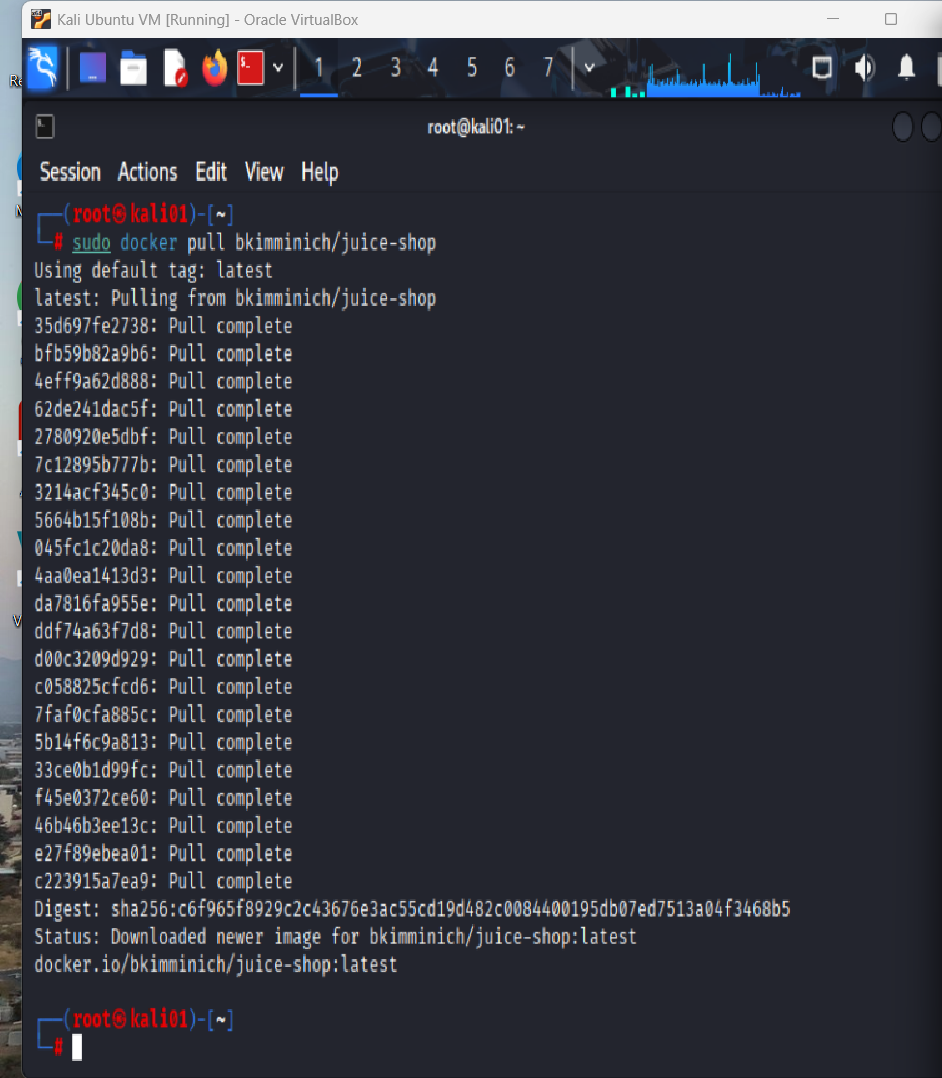
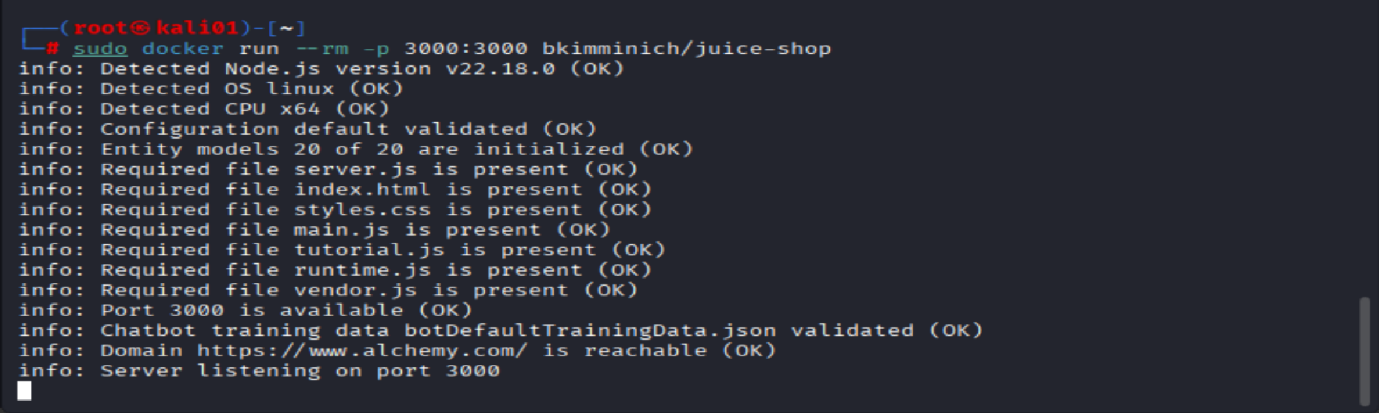
- Implement parameterized queries and input validation to prevent SQL injection.  
- Implement the blocking of SQL Injections  
- Enforce strong password policies and implement account lockout mechanisms.  
- Restrict access to non-essential ports using firewall rules.  
- Add security headers to HTTP responses to mitigate client-side attacks.  
- Deploy a Web Application Firewall (WAF) for additional protection.

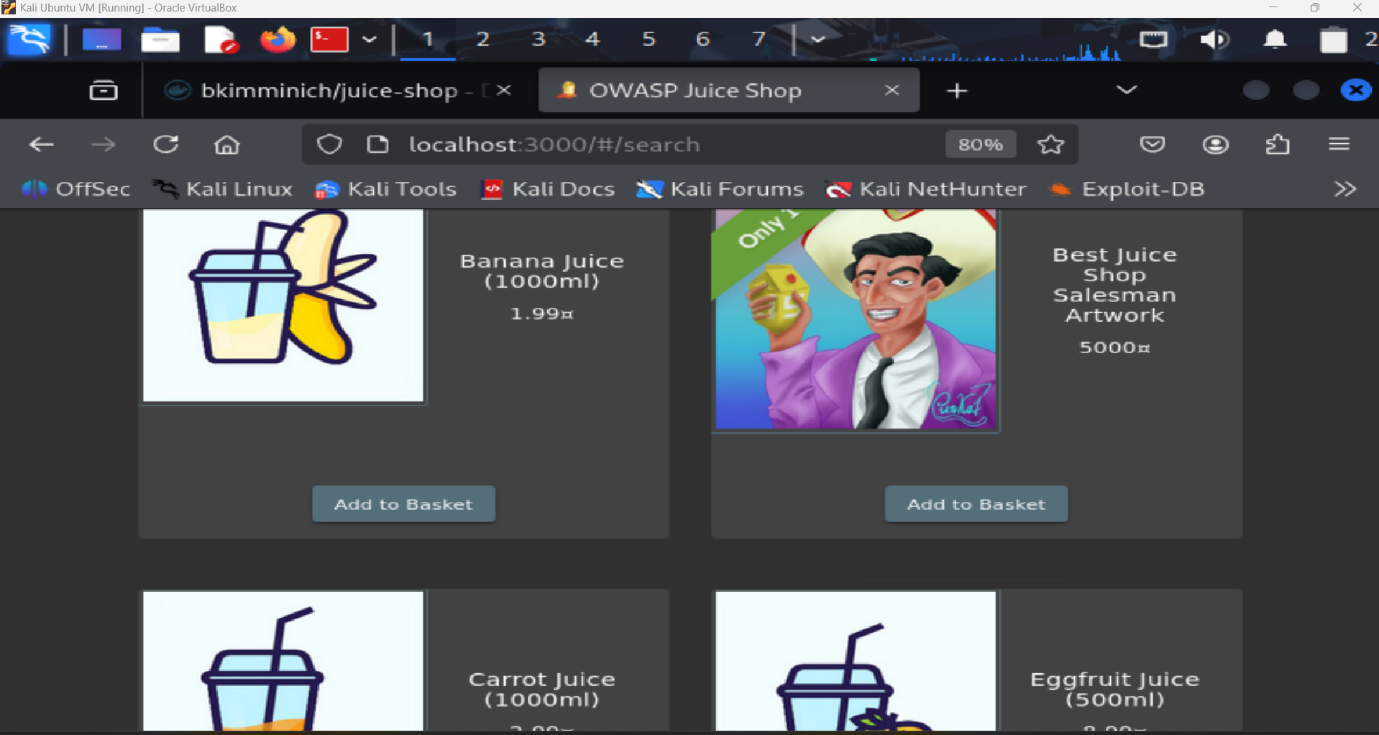
# 5. Verification & Retesting

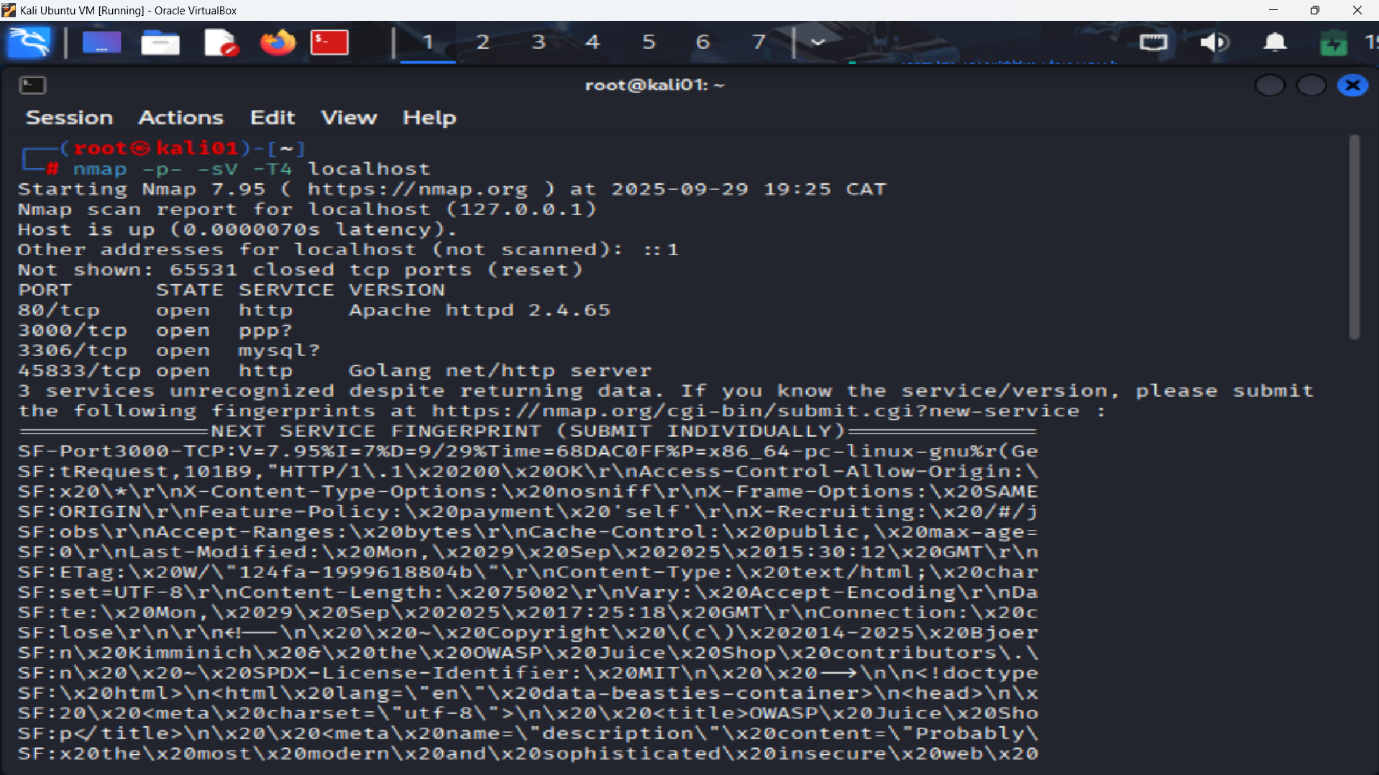
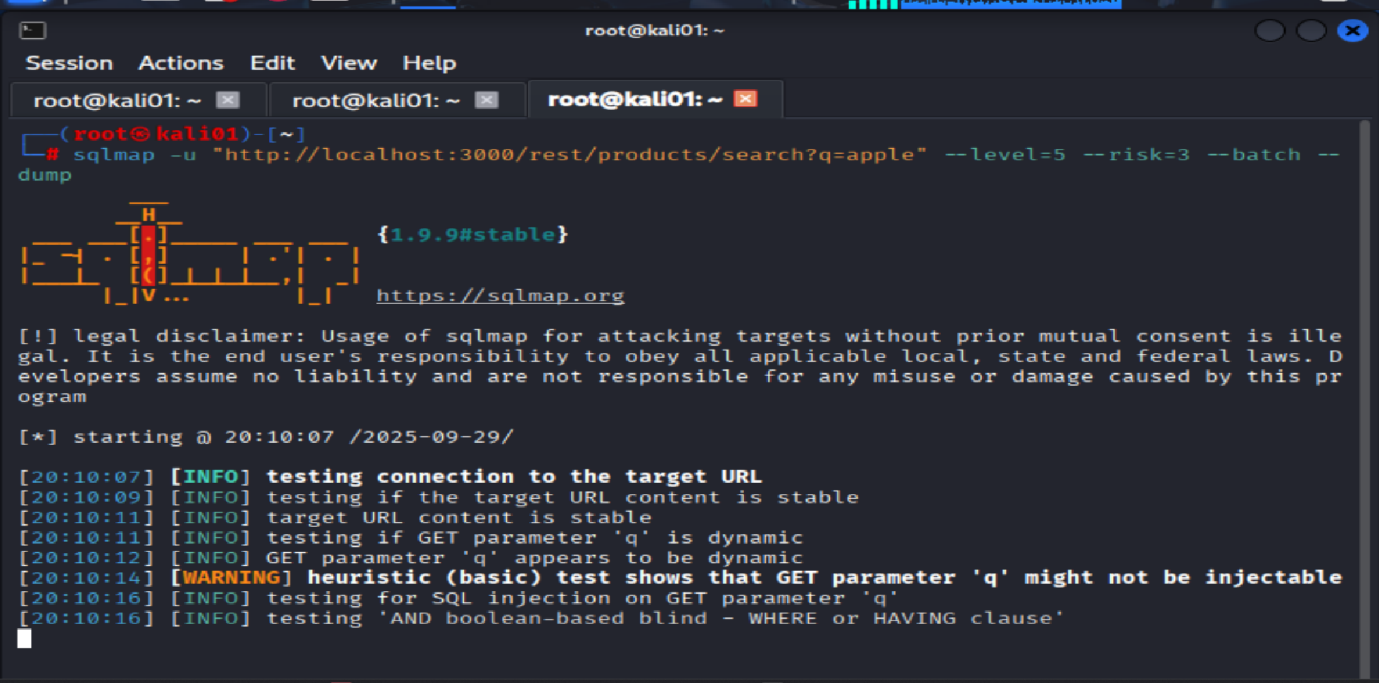
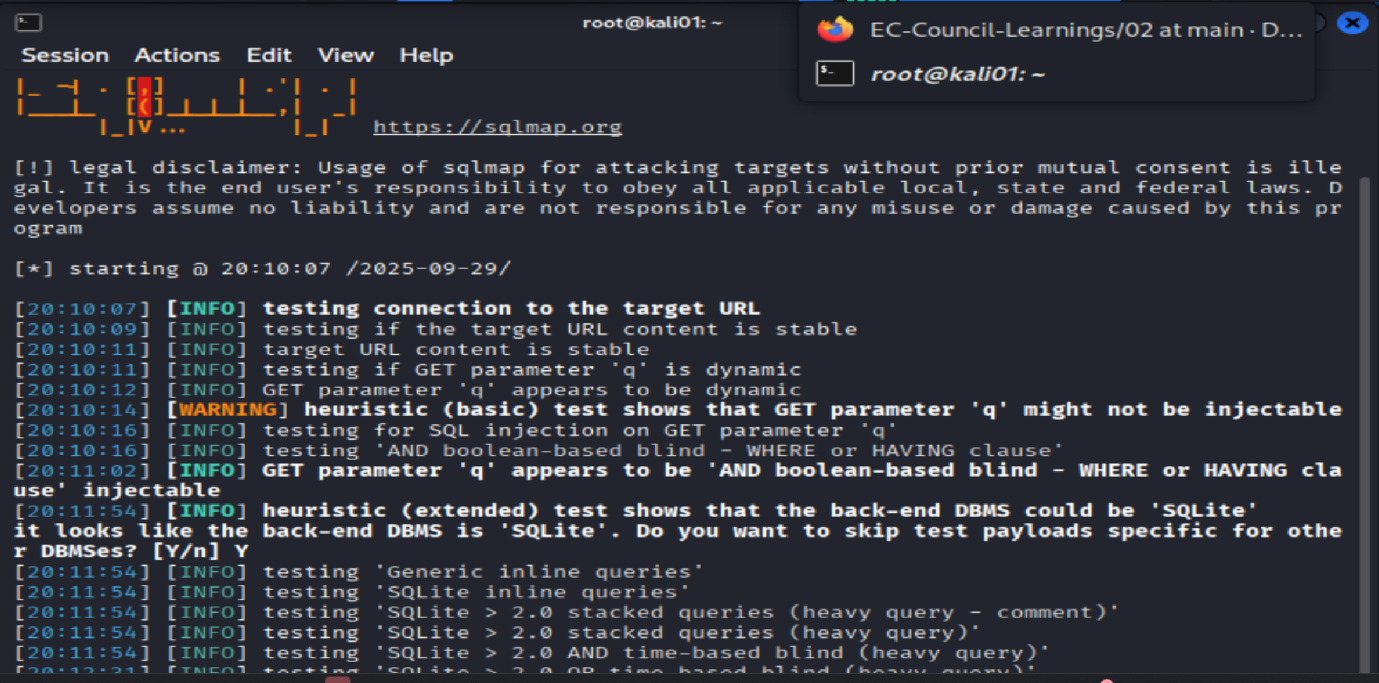
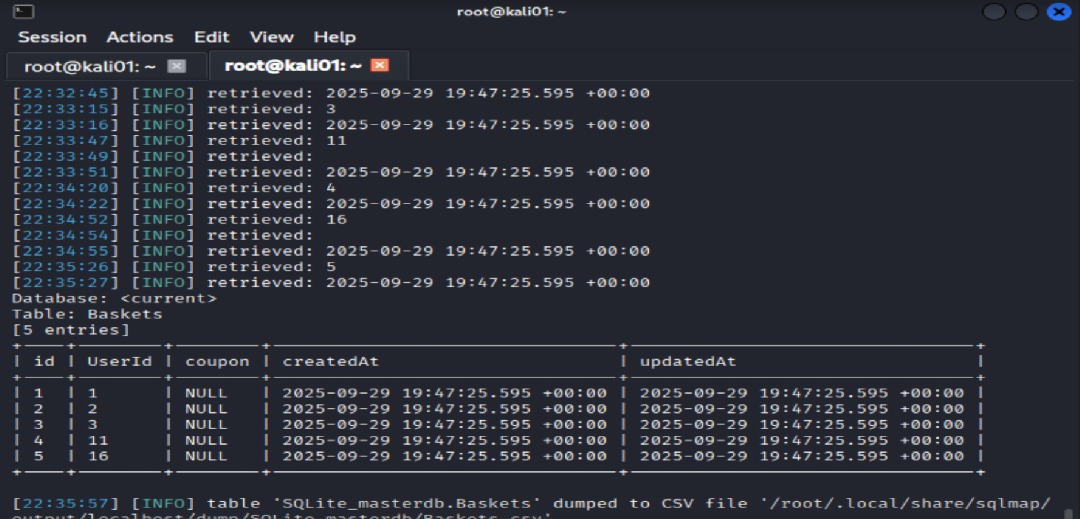
Post-remediation, the following steps should be taken to verify improvements:  
- Re-run sqlmap to confirm SQL injection is mitigated.  
- Test login endpoint with Hydra to ensure brute-force protection is effective.  
- Use Nmap to verify only essential ports are open.  
- Inspect HTTP responses for presence of recommended security headers.

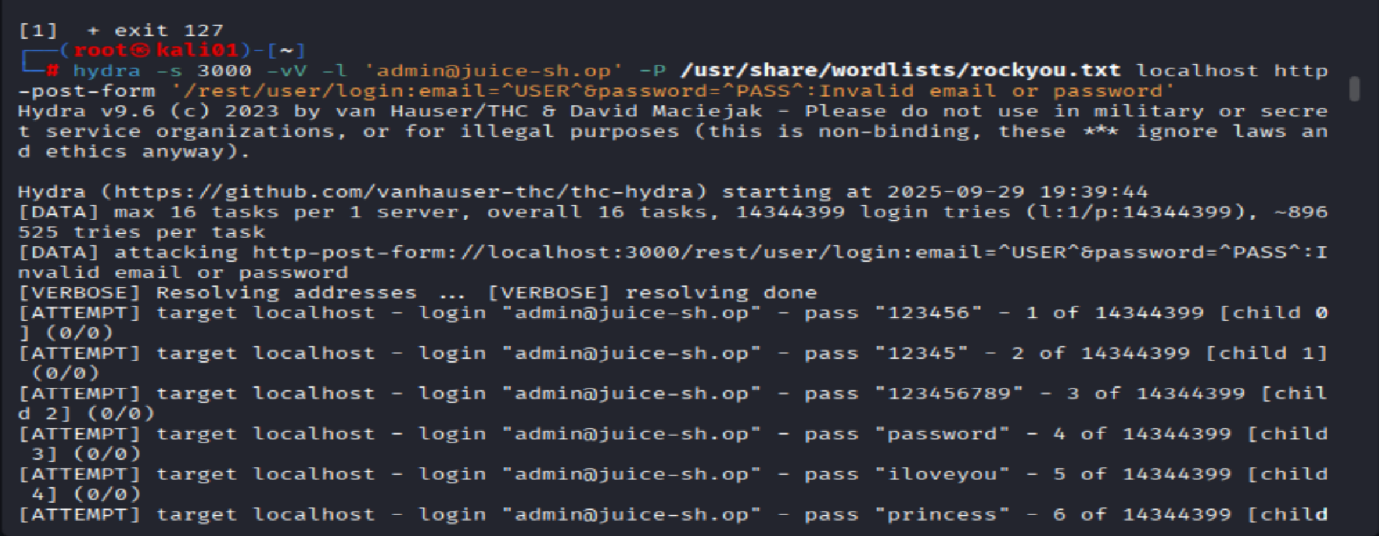
**1.1 – Set up a web app with known vulnerabilities (OWASP Juice Shop)**

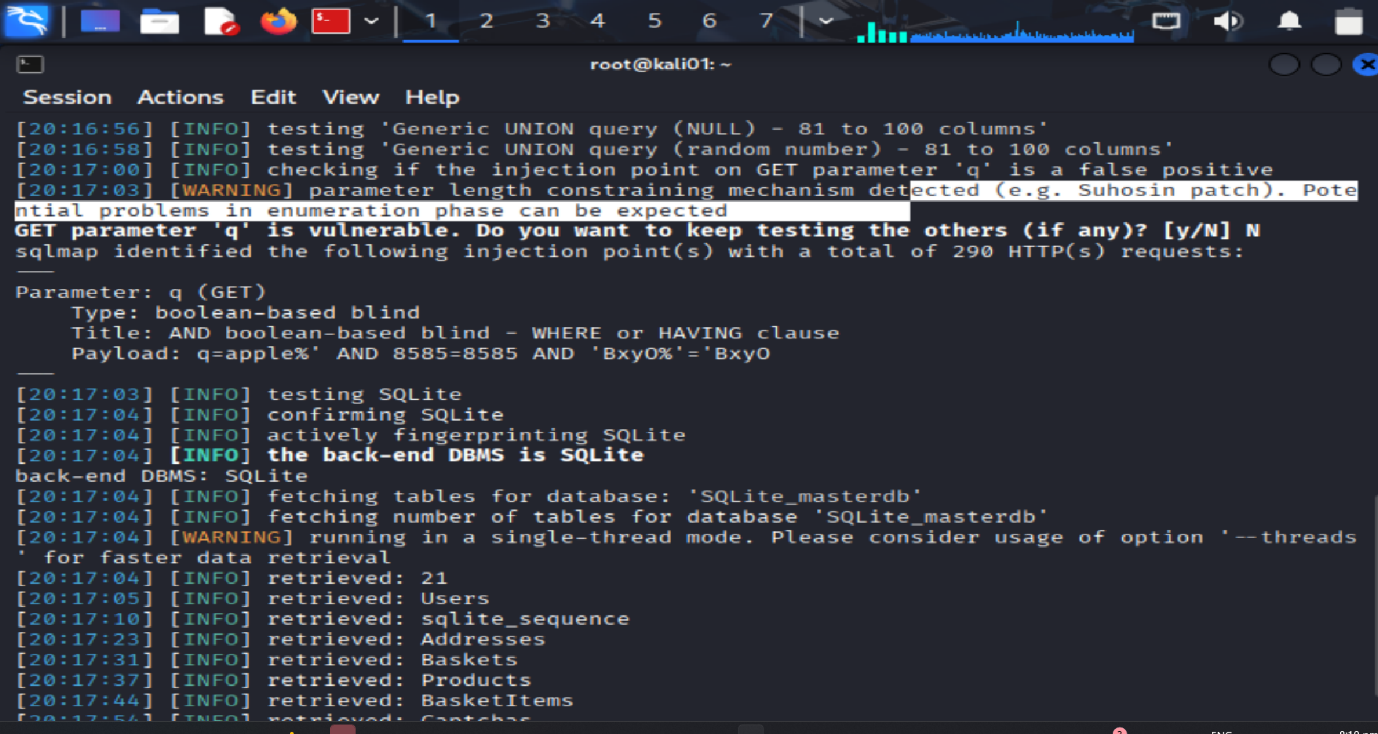
 **Figure 1 — Kali update & Docker installation (Environment prep for 1.1).**  
sudo apt-get update && sudo apt-get install docker.io confirms a fresh package index and installs Docker to run the vulnerable application container.

  
**Figure 2 — Pulling the OWASP Juice Shop container (1.1).**  
**sudo docker pull bkimminich/juice-shop** fetches the official vulnerable web app image used throughout testing.  
  
**Figure 2.1 — Pulling the OWASP Juice Shop container (1.1).**  
  
  
  
  
  
  
  
  
  
**Figure 3 — Container run & service boot logs (1.1).**  
**sudo docker run -p 3000:3000 bkimminich/juice-shop** maps the app to localhost:3000. Logs show successful initialization and server listening.  
  
  
  
**Figure 4 — Application reachable in browser (1.1 evidence).**  
**http://localhost:3000** loads the Juice Shop storefront, confirming deployment of a deliberately insecure web application for controlled testing.

  
  
  
**1.2 – Develop test cases to identify & exploit vulnerabilities (SQLi, XSS, CSRF & auth weaknesses)**

**Figure 5 — Reconnaissance scan of localhost (1.2: service discovery).**  
nmap -p- -sV -T4 localhost enumerates open ports and versions, identifying the Juice Shop service on **3000/tcp** and web stack components to guide attack paths.  
  
  
  
**Figure 6.0 — SQLi test case against product search endpoint (1.2: input injection).**  
**sqlmap -u "http://localhost:3000/rest/products/search?q=apple" --level=5 --risk=3 --batch --dump** exercises the **/rest/products/search** parameter **q** for SQL injection. Output shows dynamic parameter testing and injection heuristics.  
  
  
**Figure 6.1 — SQLi test case against product search endpoint (1.2: input injection).**  
 **Figure 6.0 & 6.1— SQLi test case against /rest/products/search (1.2).  
sqlmap -u "http://localhost:3000/rest/products/search?q=apple" --level=5 --risk=3 --batch --dump   
*Performs high-coverage SQLi checks on parameter q. (Before mitigations) the tool can enumerate the DBMS and attempt to dump data, evidencing the vulnerability prior to hardening.*Figure 6.2 SQLi data exfiltration: Baskets table dumped (1.2).  
Using sqlmap --dump, the application’s database was enumerated and the Baskets table contents were extracted, revealing fields id, UserId, coupon, createdAt, updatedAt with 5 entries. This confirms successful SQL injection leading to data disclosure.**  
  
  
  
  
  
**Figure 6.3 SQLi data exfiltration: Addresses table enumerated (1.2).  
sqlmap --dump accessed the Addresses table (6 entries shown). Although many fields render as <blank> in terminal view, the dump demonstrates unauthorized read access to address records via SQL injection.**  
A screenshot of a computer screen

AI-generated content may be incorrect.  
  
**Figure 7.0 — Brute-force login test case (1.2: weak authentication).**  
hydra -s 3000 -Vv -l admin@juice-sh.op -P /usr/share/wordlists/rockyou.txt localhost http-post-form "/rest/user/login:email=^USER^&password=^PASS^:Invalid email or password" validates resilience of the login flow against credential stuffing and common-password attacks.  


**Figure 7.1 — Brute-force login test case (1.2: weak authentication).**  
  
  
**1.3 – Implement security strategies & controls (mitigations applied and validated)***We implemented defense-in-depth against SQL injection at the application, edge, and database layers. The search endpoint was refactored to use parameterized queries and strict input validation, complemented by rate-limiting to deter automated probing and a WAF (ModSecurity + OWASP CRS) to block advanced payloads. Post-mitigation verification with curl and sqlmap demonstrates blocked patterns (403/400), failed injection attempts (no dump), throttled bursts (429), and WAF rule hits.*

**1. Input validation & sanitization (SQLi)  
What we implemented (server-side):**

* Allow-list validation on query inputs
* Return **400/403** on suspicious patterns (e.g., quotes, --, /\* \*/, UNION SELECT, SLEEP().

Log blocked attempts with timestamp, client IP, and request path.  
  
**Implementation steps:**  
(Working NGINX config) **1.1 Save the config to a file, /home/you/nginx.conf.**

**1.2 Run:**

**docker run -d --name waf -p 8080:8080 \**

**-v /home/you/nginx.conf:/etc/nginx/nginx.conf:ro \**

**nginx:stable**

**Self-contained config:**

worker\_processes 1;

events { worker\_connections 1024; }

http {

log\_format sqli '$remote\_addr - $time\_local "$request" $status '

'"$query\_string" "$http\_user\_agent"';

access\_log /var/log/nginx/access.log sqli;

error\_log /var/log/nginx/error.log warn;

upstream juice\_upstream { server 127.0.0.1:3000; }

server {

listen 8080;

server\_name localhost;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

if ($query\_string ~\* "(union.\*select|sleep\(|benchmark\(|or\s+1=1|'|--|/\\*|\\*/)") {

return 403;

}

location / {

proxy\_pass http://juice\_upstream;

}

}

}

**1.3 Test:***curl -I http://localhost:8080/* **Command (malicious probe) expect block:**  
*curl -i "http://localhost:8080/rest/products/search?q=1' or 1=1 --"*

**Expected:** HTTP/1.1 403 Forbidden (if blocked at reverse proxy) **or** 400 Bad Request (if blocked by app validation).

**2. Parameterized queries (primary SQLi fix)  
What we implemented (application layer):**

* Replaced string-concatenated SQL with **prepared/parameterized** queries in the search endpoint.

**Verification (sqlmap vs. hardened app direct port, no WAF):**  
sqlmap -u "http://localhost:3000/rest/products/search?q=apple" -p q --level=5 --risk=3 --batch --dump

**Expected:** sqlmap fails to identify an injectable point (no dump).  
  
*“Parameterized queries neutralize user input as data, preventing it from being interpreted as SQL syntax. Automated enumeration fails.”*

**Anti-automation on SQLi probes (rate-limit the search)**  
**What we implemented (edge/app):**

* **Rate limiting** on /rest/products/search to slow/break automated SQLi scans (e.g., 10 req/min/IP, burst 20).
* Optional **lockout** or temporary ban for repeated blocks.

**Expected:** initial 200 responses, then **429 Too Many Requests** as the limit trips.  
 **3. WAF rules specifically for SQLi**

**What we implemented (edge):**

Put a WAF in front (e.g., **ModSecurity + OWASP CRS**) to block advanced SQLi payloads that may bypass simple regexes.

**Run a WAF container in front of your app (example):**

docker network create juice-net

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

docker run -d --name waf --network juice-net -p 8080:80 \

-e PARANOIA=1 -e BACKEND=http://juice:3000 \

owasp/modsecurity-crs:nginx

**Commands (expect WAF block 403 with rule IDs):**  
*curl -i* [*http://localhost:8080/rest/products/search?q=1 UNION SELECT 1,2,3*](http://localhost:8080/rest/products/search?q=1%20UNION%20SELECT%201,2,3) *curl -i "http://localhost:8080/rest/products/search?q=1' AND SLEEP(5)--"*

**Expected:** HTTP/1.1 403 Forbidden and response headers/body mentioning ModSecurity rule IDs.

**Conclusion:**  
  
*This work satisfied the assignment from start to finish. We set up a deliberately vulnerable app (OWASP Juice-Shop), confirmed SQL injection by pulling data from the database, and showed weak login protections. After the fixes, automated SQLi attempts were either blocked or produced no useful output; sqlmap couldn’t extract anything, and burst requests were rate-limited. The login endpoint is now harder to brute-force, and added security headers strengthen the client-side security posture.*

*The main takeaways: start by fixing the code (use parameterized queries with strict input checks), then add simple edge defenses (rate limiting and basic request filtering), and use a WAF for extra depth when needed. Our re-tests and screenshots clearly demonstrate the improvements and can be reused to prevent future regressions.*