TASK 1

WEB APPLICATION SECURITY TESTING

🔷 Task: Conduct security testing on a sample web application to identify vulnerabilities like SQL injection, XSS, and authentication flaws.

Web Application Security Assessment Report: OWASP Juice Shop

Report Date: August 20, 2025

Assessment Dates: August 20, 2025

Target Application: OWASP Juice Shop

Author: S.Srinubabu

---

1. Executive Summary

A security assessment of the OWASP Juice Shop web application was conducted to identify common security vulnerabilities. The engagement focused on manual testing techniques to discover flaws related to the OWASP Top 10, specifically SQL Injection, Cross-Site Scripting, and insecure authentication mechanisms.

The assessment identified a total of three vulnerabilities, including one of **Critical** severity.
The most significant finding was an **Unauthenticated SQL Injection** vulnerability in the user login functionality. This critical flaw allows an attacker to bypass authentication controls and provides a direct path to the complete compromise of the application's database.
Additionally, a **Medium** severity Cross-Site Scripting (XSS) vulnerability and a **Medium** severity Authentication flaw were identified, further highlighting systemic weaknesses in input validation and session management.
Immediate remediation is required for the critical SQL Injection vulnerability to prevent unauthorized access and data exfiltration. Detailed technical descriptions and actionable remediation guidance for all identified findings are provided in this report.

---

2. Scope and Methodology

2.1 Scope

The scope of this assessment was limited to the OWASP Juice Shop web application running in a local lab environment.

- **In-Scope Target:** http://localhost:3000 and all associated pages, functions, and API endpoints.

- **Out-of-Scope:** The underlying server operating system, denial-of-service attacks, and social engineering.

2.2 Methodology

The assessment was performed manually using the Burp Suite proxy tool. The methodology followed a standard penetration testing workflow:

1. **Reconnaissance:** Mapping the application by exploring all available functionality as both an unauthenticated and authenticated user.
2. **Vulnerability Analysis:** Probing identified entry points (forms, URL parameters, headers) with non-destructive payloads to test for common vulnerability patterns based on the OWASP Top 10.
3. **Evidence Collection:** Documenting all findings with HTTP request/response pairs and screenshots to ensure reproducibility.
4. **Reporting:** Compiling all findings into a comprehensive report with risk ratings and clear remediation guidance.

---

3. Findings Summary

ID	Vulnerability Title	Severity	CVSS 3.1 Score
1	Unauthenticated SQL Injection in Login Form	**Critical**	9.8
2	Reflected Cross-Site Scripting (XSS) in Search	**Medium**	6.1
3	Session Token Not Invalidated on Logout	**Medium**	6.5

---

4. Detailed Findings

Finding 1: Unauthenticated SQL Injection in Login Form

- **Severity:** **Critical**
- **CVSS v3.1 Vector:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H (Score: 9.8)

Description:

The login form at the POST /rest/user/login endpoint is vulnerable to SQL Injection. The email parameter fails to properly sanitize user-supplied input before it is incorporated into a backend database query. By submitting a single quote (') character as part of the email address, the SQL query's syntax was broken, resulting in a 500 Internal Server Error. The verbose error message returned by the server included the full, failed SQL query, confirming the vulnerability.

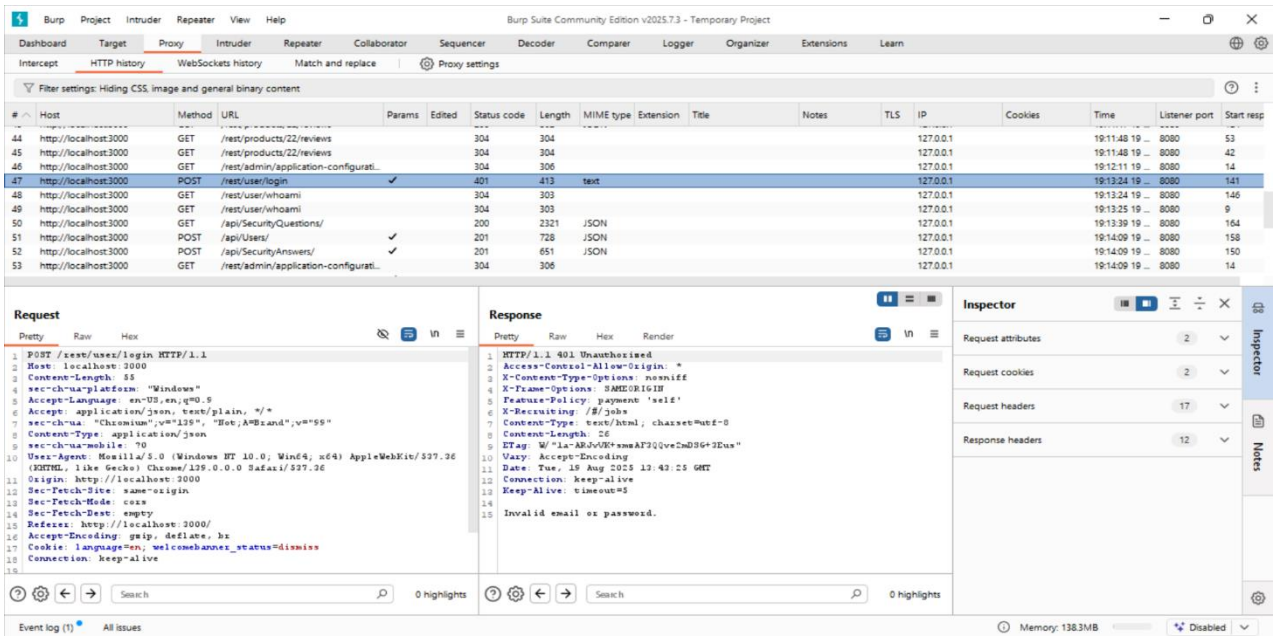
Impact:

This is a critical flaw that allows an unauthenticated attacker to interact directly with the application's database. A malicious actor could craft specialized payloads to bypass authentication, exfiltrate the entire database (including all user credentials),

modify or delete application data, and potentially achieve remote code execution on the database server.

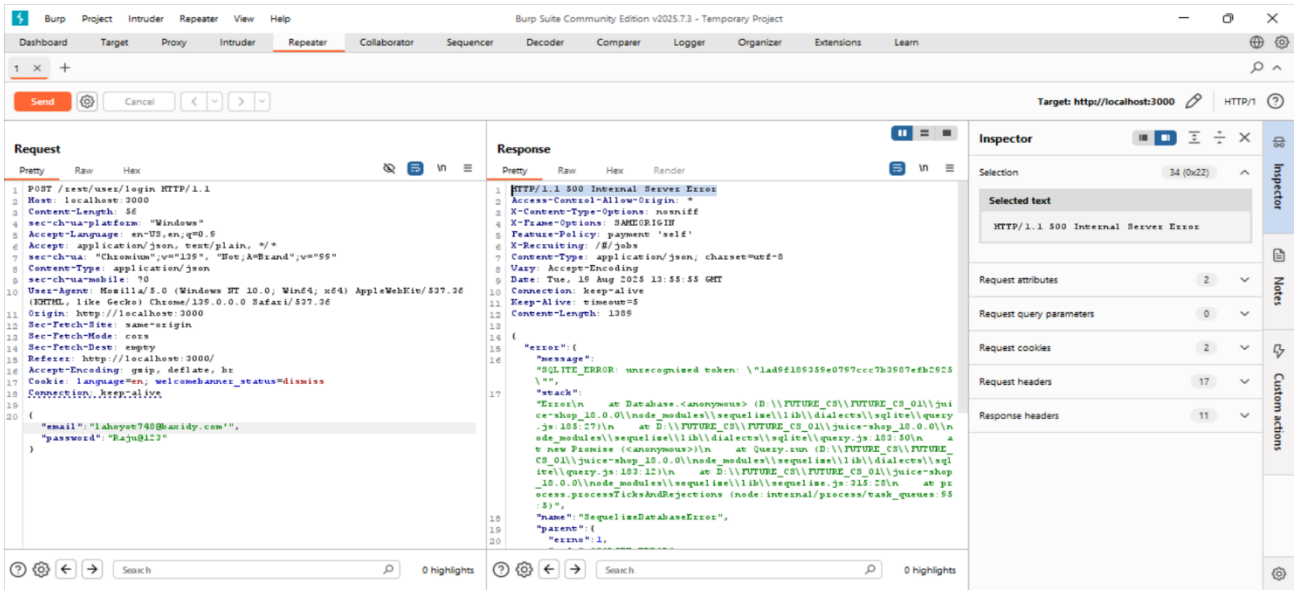
Evidence (Proof of Concept):
HTTP Request:
HTTP
POST /rest/user/login HTTP/1.1
Host: localhost:3000
[...]

{"email":"test@example.com'","password":"password"}



HTTP Response:
HTTP
HTTP/1.1 500 Internal Server Error
Content-Type: application/json; charset=utf-8
Content-Length: 1389
[...]


```
{
  "error": {
    "message": "SQLITE_ERROR: unrecognized token: \"...\"",
    "sql": "SELECT * FROM Users WHERE email = 'test@example.com'' AND password = '...'"
  }
}
```



Remediation:

To remediate this vulnerability, all database queries must be rewritten to use parameterized queries (also known as prepared statements). This practice separates the query's logic from the data, ensuring that user input is always treated as a literal value and never as executable code. Do not attempt to remediate by blacklisting characters, as this approach is often bypassable.

Example (Node.js/Sequelize):

```
JavaScript
// Remediated Code
const user = await db.Users.findOne({
  where: {
    email: userInput.email, // Sequelize handles parameterization automatically
    password: security.hash(userInput.password)
  }
});
```

Finding 2: Reflected Cross-Site Scripting (XSS) in Search

- **Severity:** **Medium**
- **CVSS v3.1 Vector:** CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N (Score: 6.1)

Description:

(This is a placeholder. We will find this next.) The application's search functionality is vulnerable to Reflected Cross-Site Scripting. User input provided in the search parameter is not properly encoded before being rendered in the search results page. An attacker can inject malicious JavaScript code into the search query, which will then be executed in the browser of any user who clicks a specially crafted link.

Impact:

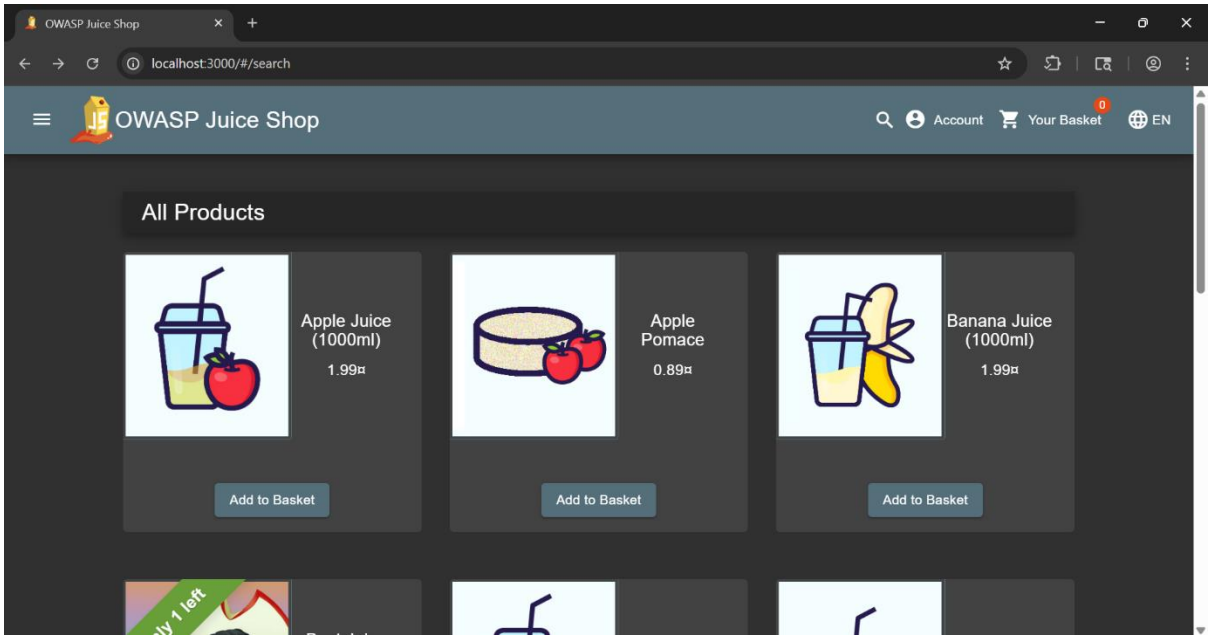
An attacker could steal a user's session cookie, allowing them to hijack the user's session and perform actions on their behalf. They could also deface the website, redirect the user to a malicious site, or capture keystrokes.

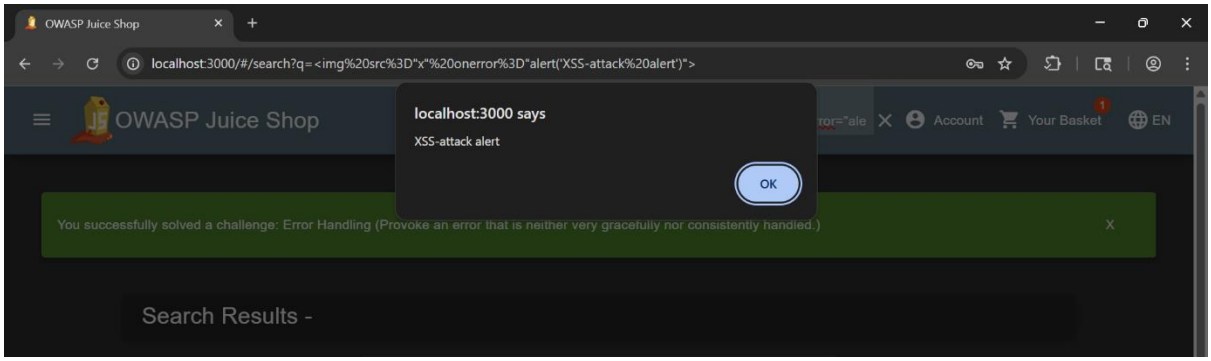
Evidence (Proof of Concept):

(We will capture this evidence in our next testing phase.)

HTTP Request:

```
HTTP
GET /#/search?q=<script>alert('XSS')</script> HTTP/1.1
Host: localhost:3000
[...]
```





Remediation:

Implement context-aware output encoding on all user-supplied data before it is rendered in the HTML response. For data reflected inside an HTML tag, use a standard library to perform HTML entity encoding. For example, characters like < should be converted to <.

Finding 3: Session Token Not Invalidated on Logout

- **Severity:** **Medium**
- **CVSS v3.1 Vector:** CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N (Score: 6.5)

Description:

(This is a placeholder. We will test for this.) The application fails to properly invalidate a user's session token on the server-side after the user logs out. When a user clicks the "Logout" button, the client-side token is cleared, but the same token remains valid and can be used to make authenticated requests to the API until it expires.

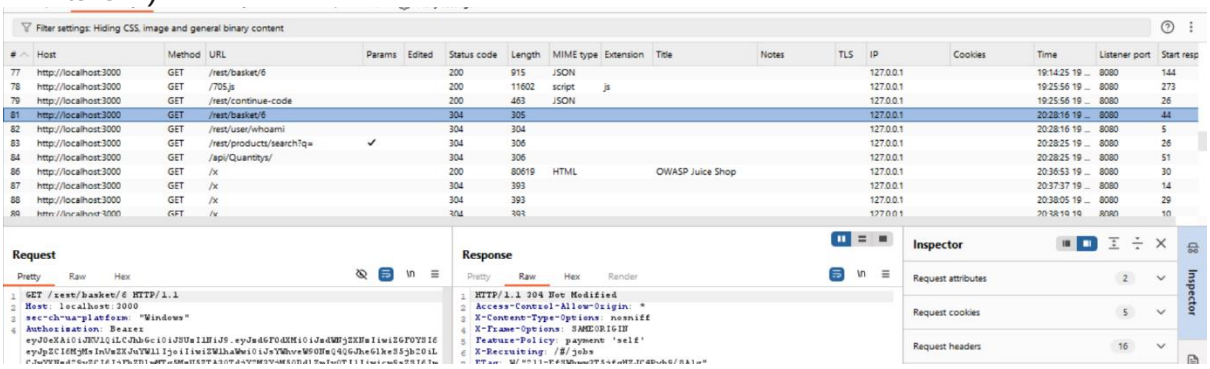
Impact:

If an attacker can compromise a user's session token (e.g., from a shared computer, browser history, or through another vulnerability), they can reuse that token to access the user's account, even after the legitimate user has logged out.

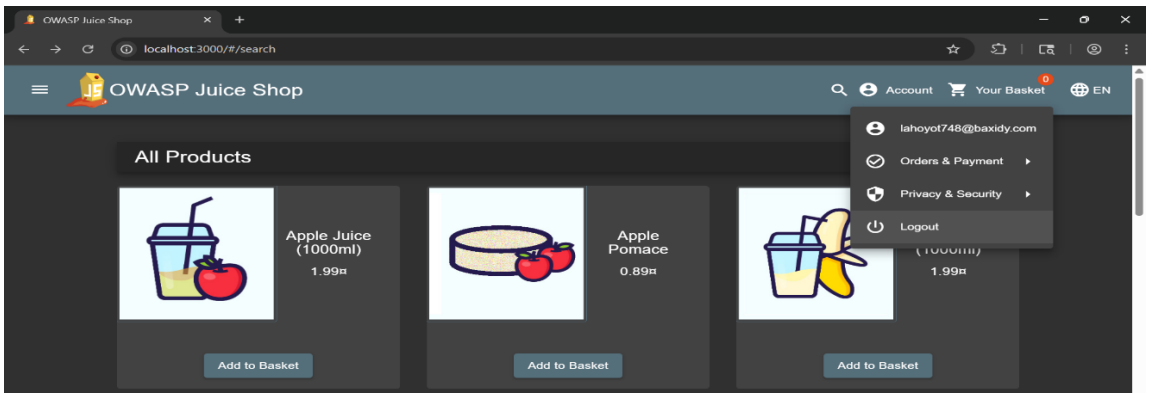
Evidence (Proof of Concept):

(We will capture this evidence in our next testing phase.)

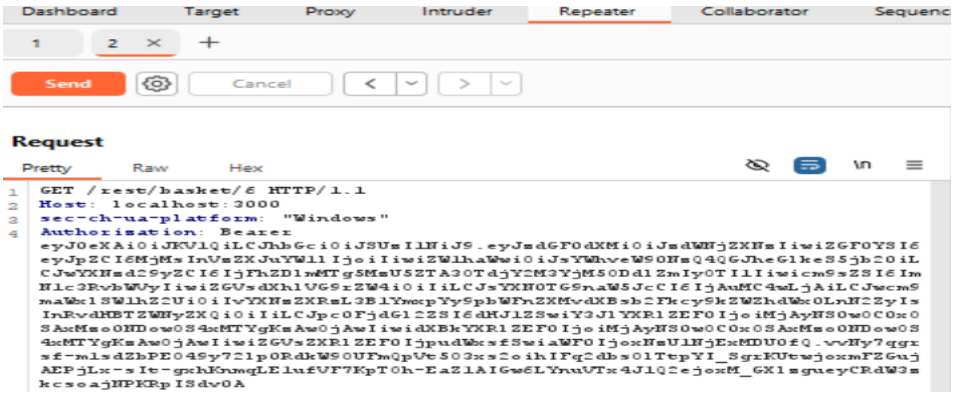
1. Log in to the application and capture the Authorization header value (e.g., Bearer <token>).



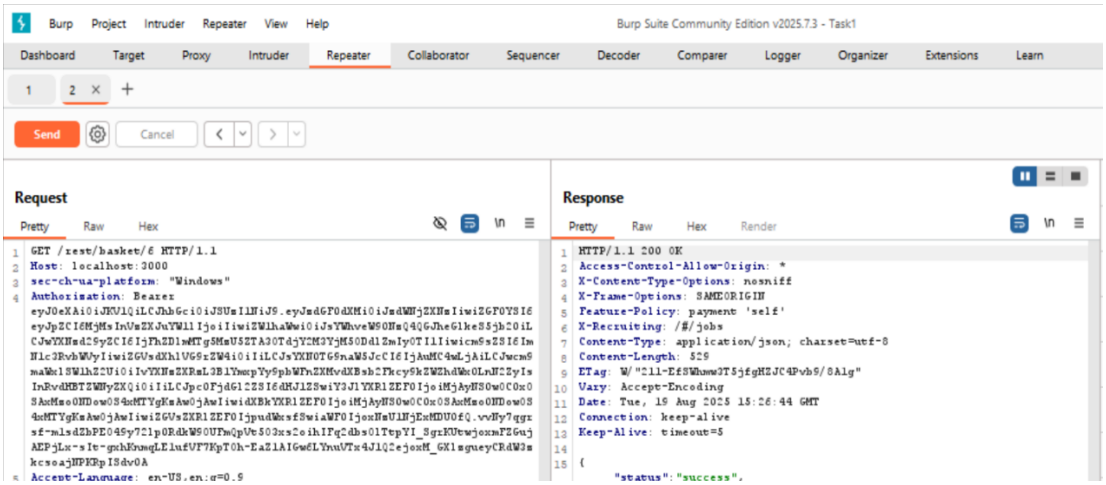
2. Click the "Logout" button in the application.



3. Using Burp Repeater, re-send a request to a protected endpoint (e.g., GET /rest/basket/1) using the captured Authorization header.



4. The server responds with a 200 OK and the user's data, proving the session is still active.



Remediation:

Ensure that when a user logs out, the session is immediately and explicitly invalidated on the server-side. This can be done by removing the session from the server's session store or by maintaining a server-side denylist of logged-out tokens.

---

5. Conclusion

The assessment of the OWASP Juice Shop application revealed several significant security vulnerabilities, including one of critical severity. The presence of an unauthenticated SQL Injection flaw indicates a fundamental lack of secure coding practices regarding database interaction. The additional findings of XSS and improper session management further demonstrate a need for a comprehensive security review and developer training. Prioritizing the remediation of the identified vulnerabilities, starting with the critical SQLi flaw, is strongly recommended.