PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
Ministry of Higher Education and Scientific Research
University of Skikda — Department of Computer Science
Master's in Information Security — Semester 3 (M2) — UEF322
Web and Mobile Application Security
Updated on 2026-09-22 at 23:28:17

Lab 3 — Exploiting and Fixing an Injection and an XSS

Theoretical reminder

Chapter 3 identifies a common root cause for SQL injection and XSS: **the lack of strict separation between code and data**. This lab makes that idea tangible: in step 2, your input `' OR 1=1-` is not treated as mere data but, once concatenated, becomes a full-fledged SQL clause – exactly the mechanism described in the course example. Likewise, in step 3, your `<script>` tag is not displayed as text but interpreted as executable code by a third party's browser. Understanding *why* these two attacks, seemingly very different, obey the same principle is the central objective of this lab.

Objective

Carry out an SQL injection and an XSS on a legal target, understand their exact mechanism, then identify the appropriate fix (see chapter 3 of the course handbook).

Prerequisites

Having studied chapter 3: OWASP Top 10, SQL injection, XSS (reflected/stored/DOM-based), CSRF, protection techniques.

Tools and installation

- **Docker**: `https://www.docker.com/products/docker-desktop` (needed to run Juice Shop).
- **OWASP Juice Shop** *or* **DVWA** (deliberately vulnerable applications): see the Docker command in step 1.
- **Burp Suite Community Edition**: `https://portswigger.net/burp/communitydownload` (free, installer for Windows/Linux/macOS) *or* **OWASP ZAP**: `https://www.zaproxy.org/download/`.

- A browser configured to go through the proxy (in the browser's network settings, or via the *FoxyProxy* extension).

Ethical framework reminder

Juice Shop and DVWA are designed to be attacked for teaching purposes. **Never reproduce these techniques on a site you are not authorized to test** (chapter 1, section "test to defend better").

Procedure

Step 1 — Install the target and configure the proxy

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

Open `http://localhost:3000` in the browser. Configure the browser to send its traffic to the Burp/ZAP proxy (usually `127.0.0.1:8080`), then install the proxy's certificate to intercept HTTPS traffic as well if needed.

Step 2 — Authentication bypass through SQL injection

In the login form, enter as the e-mail address:

```
' OR 1=1--
```

and any password. Intercept the request in Burp/ZAP before it is sent, inspect the body of the POST request, then let it through. If the login succeeds without valid credentials, you have bypassed authentication through SQL injection (see the explanation of the mechanism in chapter 3, section "Major Web attacks").

Step 3 — Trigger a stored XSS

Look for a field that stores and redisplays user input (product review, comment, profile name). Enter:

```
<script>alert(document.cookie)</script>
```

Submit, then reload the page that displays this content. If an alert box appears showing the cookie's content, the script ran: you have confirmed a stored XSS.

Step 4 — Replay and modify an attack with the Repeater

In Burp (*Repeater* tab) or ZAP (equivalent tab), take the request from step 2, change the injected payload (for example `' OR '1'='1` instead of `' OR 1=1-`) and send it again. Compare the result.

Step 5 — Identify the fix

For each of the two flaws exploited, write the precise fix corresponding to the "Protection techniques" section of chapter 3: a parameterized query for the SQL injection, output encoding for the XSS.

Comprehension questions

1. Rewrite the complete SQL query as it is executed on the server side once your step 2 payload has been concatenated (see the chapter 3 example).
2. Why does the "parameterized query" fix prevent the injection, whereas a simple function that escapes apostrophes may still be bypassed?
3. Is the XSS in step 3 reflected, stored or DOM-based? Justify using the definition of each variant (chapter 3, section "Major Web attacks").
4. Does the session cookie used by Juice Shop carry the `HttpOnly` attribute? If so, could the `alert(document.cookie)` exploit have stolen this cookie and sent it to an external server (preview of chapter 6)?

Deliverable

A short report, in the format of a simplified penetration test report, with for each flaw exploited:

Flaw — Proof of exploitation (screenshot) — Impact — Proposed fix