SQL Injection & XSS Exploitation Lab

Penetration Testing Report

SecureBank Vulnerable Web Application

Date:	February 21, 2026
Environment:	Ubuntu 24 / PHP 8.4 / MySQL
Tools:	PHP Built-in Server, Chrome

Executive Summary

This report documents the full exploitation of a deliberately vulnerable web application (SecureBank™) to demonstrate two critical OWASP Top 10 vulnerabilities: SQL Injection (A03:2021) and Cross-Site Scripting/XSS (A07:2021).

The application was built with two versions: a vulnerable version containing intentional security flaws, and a hardened version implementing industry-standard defenses. Both were tested side-by-side to prove that the attacks succeed on the vulnerable codebase and are completely neutralized by the secure implementation.

Vulnerability	Severity	Vulnerable	Secure
SQL Injection	**CRITICAL**	✗ Exploitable	✓ Blocked
Reflected XSS	**HIGH**	✗ Exploitable	✓ Blocked

1. Environment Setup

The lab environment was set up on Ubuntu with PHP 8.4.11 and MySQL. Below are the exact commands used to initialize the application.

1.1 Install Dependencies

```
sudo apt update
sudo apt install php php-mysql mysql-server -y
```

1.2 Import the Database

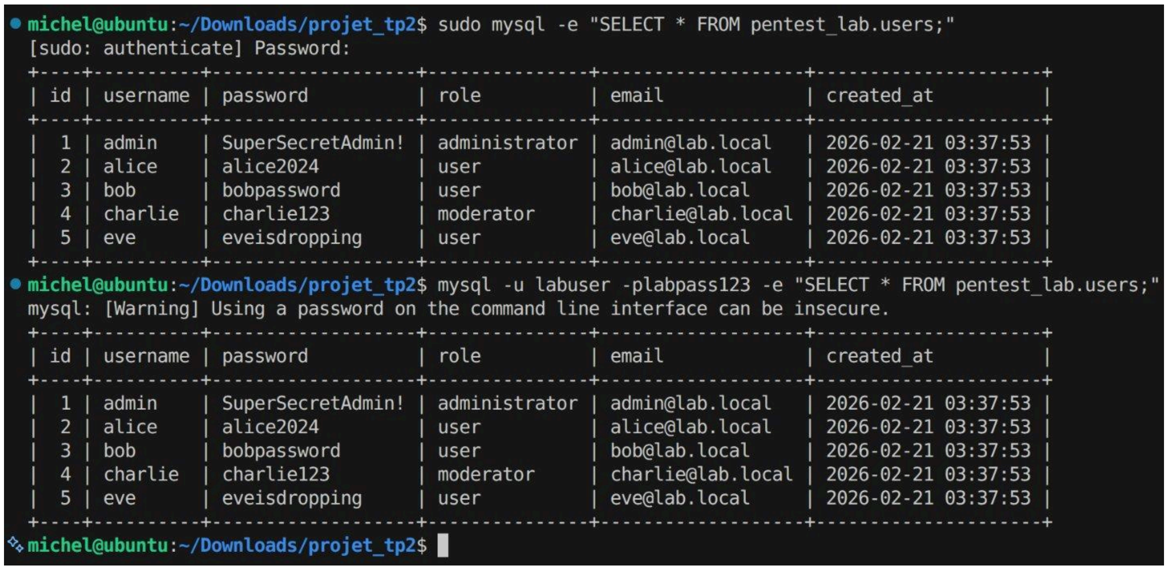
The database.sql script creates the pentest_lab database, a users table with 5 test accounts, and a dedicated low-privilege MySQL user (labuser).

```
cd ~/Downloads/projet_tp2
sudo mysql < database.sql
```

1.3 Database Verification

Both root and the application user (labuser) were tested to confirm the database was populated correctly and accessible:

```
sudo mysql -e "SELECT * FROM pentest_lab.users;"
mysql -u labuser -plabpass123 -e "SELECT * FROM pentest_lab.users;"
```



```
michel@ubuntu:~/Downloads/projet_tp2$ sudo mysql -e "SELECT * FROM pentest_lab.users;"
[sudo: authenticate] Password:
+----+----------+-------------------+---------------+-------------------+---------------------+
| id | username | password          | role          | email             | created_at          |
+----+----------+-------------------+---------------+-------------------+---------------------+
|  1 | admin    | SuperSecretAdmin! | administrator | admin@lab.local   | 2026-02-21 03:37:53 |
|  2 | alice    | alice2024         | user          | alice@lab.local   | 2026-02-21 03:37:53 |
|  3 | bob      | bobpassword       | user          | bob@lab.local     | 2026-02-21 03:37:53 |
|  4 | charlie  | charlie123        | moderator     | charlie@lab.local | 2026-02-21 03:37:53 |
|  5 | eve      | eveisdropping     | user          | eve@lab.local     | 2026-02-21 03:37:53 |
+----+----------+-------------------+---------------+-------------------+---------------------+
michel@ubuntu:~/Downloads/projet_tp2$ mysql -u labuser -plabpass123 -e "SELECT * FROM pentest_lab.users;"
mysql: [Warning] Using a password on the command line interface can be insecure.
+----+----------+-------------------+---------------+-------------------+---------------------+
| id | username | password          | role          | email             | created_at          |
+----+----------+-------------------+---------------+-------------------+---------------------+
|  1 | admin    | SuperSecretAdmin! | administrator | admin@lab.local   | 2026-02-21 03:37:53 |
|  2 | alice    | alice2024         | user          | alice@lab.local   | 2026-02-21 03:37:53 |
|  3 | bob      | bobpassword       | user          | bob@lab.local     | 2026-02-21 03:37:53 |
|  4 | charlie  | charlie123        | moderator     | charlie@lab.local | 2026-02-21 03:37:53 |
|  5 | eve      | eveisdropping     | user          | eve@lab.local     | 2026-02-21 03:37:53 |
+----+----------+-------------------+---------------+-------------------+---------------------+
michel@ubuntu:~/Downloads/projet_tp2$
```

Figure 1: Database verification — 5 users seeded, labuser connectivity confirmed

1.4 Start the Servers

Two PHP development servers were launched: the vulnerable version on port 8080 and the secure version on port 9090.

```
cd ~/Downloads/projet_tp2/vulnerable
php -S localhost:8080
```

In a second terminal:

```
cd ~/Downloads/projet_tp2/secure
php -S localhost:9090
```

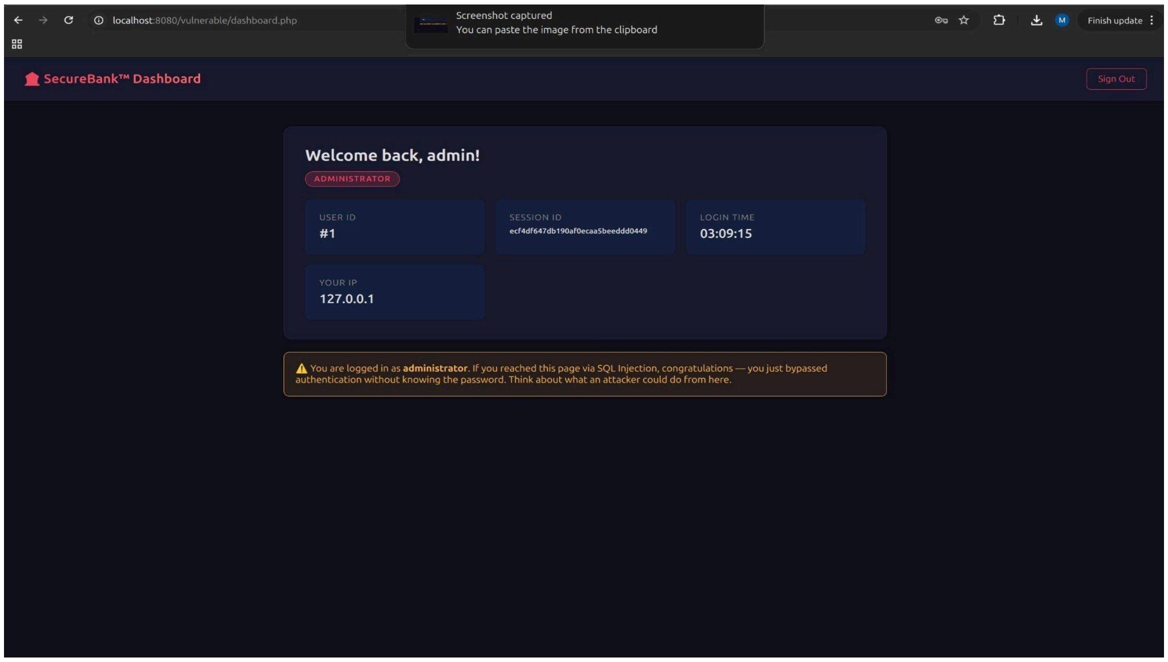
2. Baseline: Normal Authentication

Before testing any attacks, a normal login was performed to verify the application works correctly with valid credentials. The admin account uses the password SuperSecretAdmin! which is stored in the database in plaintext (a deliberate weakness for this lab).

Username: admin

Password: SuperSecretAdmin!

The application successfully authenticated the user and displayed the admin dashboard with User ID #1, the administrator role badge, the PHP session ID, login time, and client IP. This confirms the database connection, query logic, and session handling all function correctly — establishing our baseline before exploitation.


localhost:8080/vulnerable/dashboard.php
Screenshot captured
You can paste the image from the clipboard
Finish update
SecureBank™ Dashboard
Sign Out

Welcome back, admin!

ADMINISTRATOR

USER ID
#1

SESSION ID
ecf4df647db190af0ecaa5beeddd0449

LOGIN TIME
03:09:15

YOUR IP
127.0.0.1

⚠ You are logged in as **administrator**. If you reached this page via SQL Injection, congratulations — you just bypassed authentication without knowing the password. Think about what an attacker could do from here.

Figure 2: Successful login with valid credentials — admin dashboard displayed

3. Attack #1: SQL Injection (Authentication Bypass)

3.1 The Vulnerability

In the vulnerable version, authenticate.php concatenates user input directly into a SQL query:

```
$query = "SELECT * FROM users WHERE username = '$username' AND password =
'$password'";
```

Because the input is not sanitized, an attacker can inject SQL syntax to alter the query logic. The single quote (') closes the string context, and the hash (#) comments out the rest of the query, including the password check.

3.2 The Payload

Username: admin'#

Password: anything (irrelevant)

This transforms the SQL query into:

```
SELECT * FROM users WHERE username = 'admin'#' AND password = 'anything'
```

Everything after # is treated as a comment by MySQL. The password clause is completely ignored, and the database returns the admin row unconditionally.

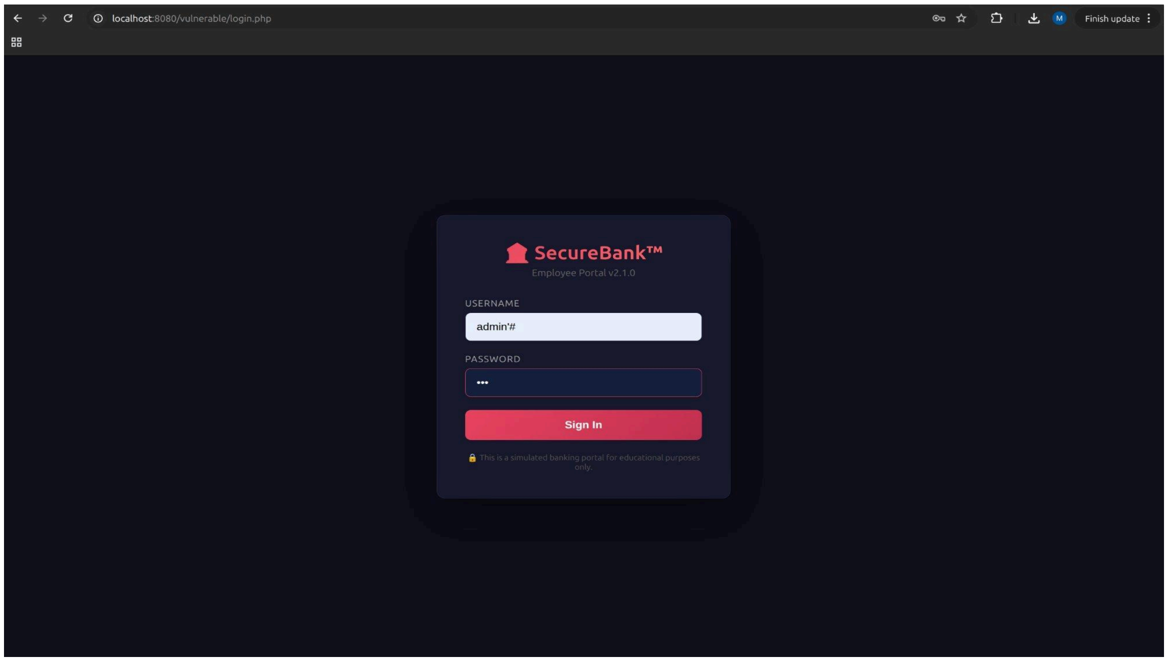
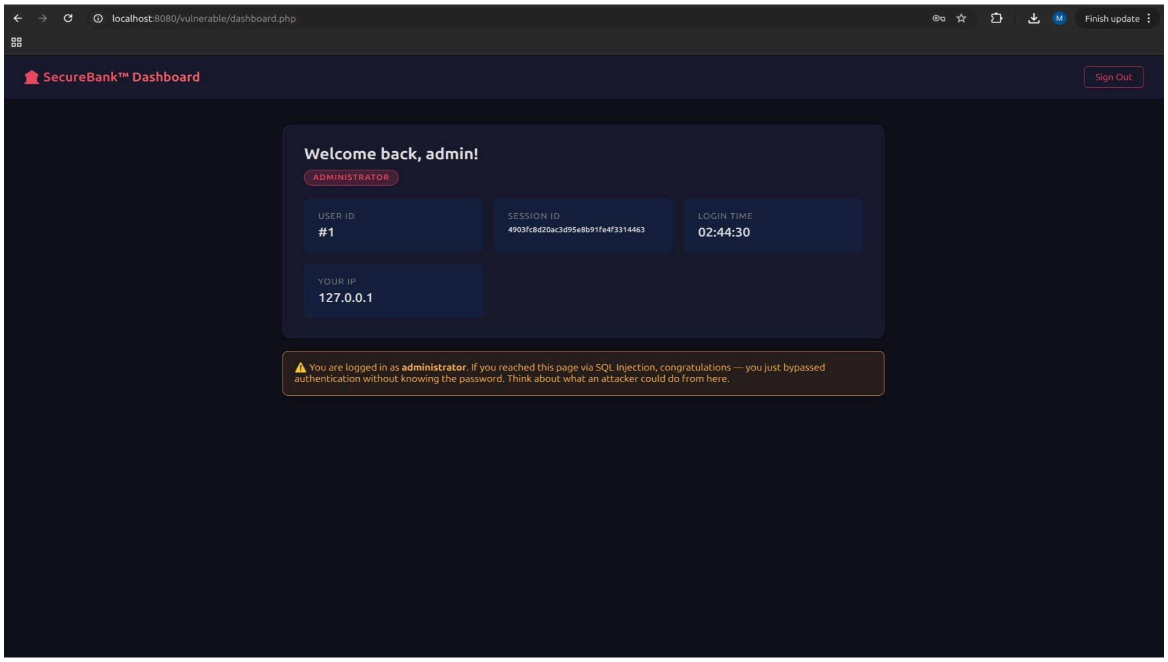

localhost:8080/vulnerable/login.php
Finish update
SecureBank™
Employee Portal v2.1.0
USERNAME
admin'#
PASSWORD
•••
Sign In
🔒 This is a simulated banking portal for educational purposes only.

Figure 3: SQL injection payload entered in the login form

3.3 Result: Full Admin Access

The injection succeeded. The application authenticated us as admin (User ID #1, role: administrator) without knowing the password. The warning banner on the dashboard confirms the bypass.



Welcome back, admin!

ADMINISTRATOR

USER ID
#1

SESSION ID
4903fc8d20ac3d95e8b91fe4f3314463

LOGIN TIME
02:44:30

YOUR IP
127.0.0.1

⚠ You are logged in as **administrator**. If you reached this page via SQL Injection, congratulations — you just bypassed authentication without knowing the password. Think about what an attacker could do from here.

Figure 4: Authentication bypassed — admin dashboard reached via SQL injection

4. Attack #2: Reflected Cross-Site Scripting (XSS)

4.1 The Vulnerability

In the vulnerable login.php, the error GET parameter is echoed into the HTML without any sanitization:

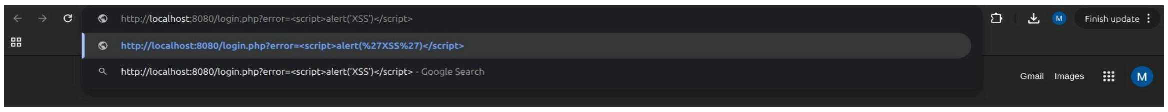
```
echo '<div class="error-box">' . $_GET['error'] . '</div>';
```

An attacker can craft a URL containing JavaScript in the error parameter. When a victim clicks this link, the browser executes the injected script in the context of the application.

4.2 The Payload

The following URL was entered directly in the browser address bar:

```
http://localhost:8080/login.php?error=<script>alert('XSS')</script>
```

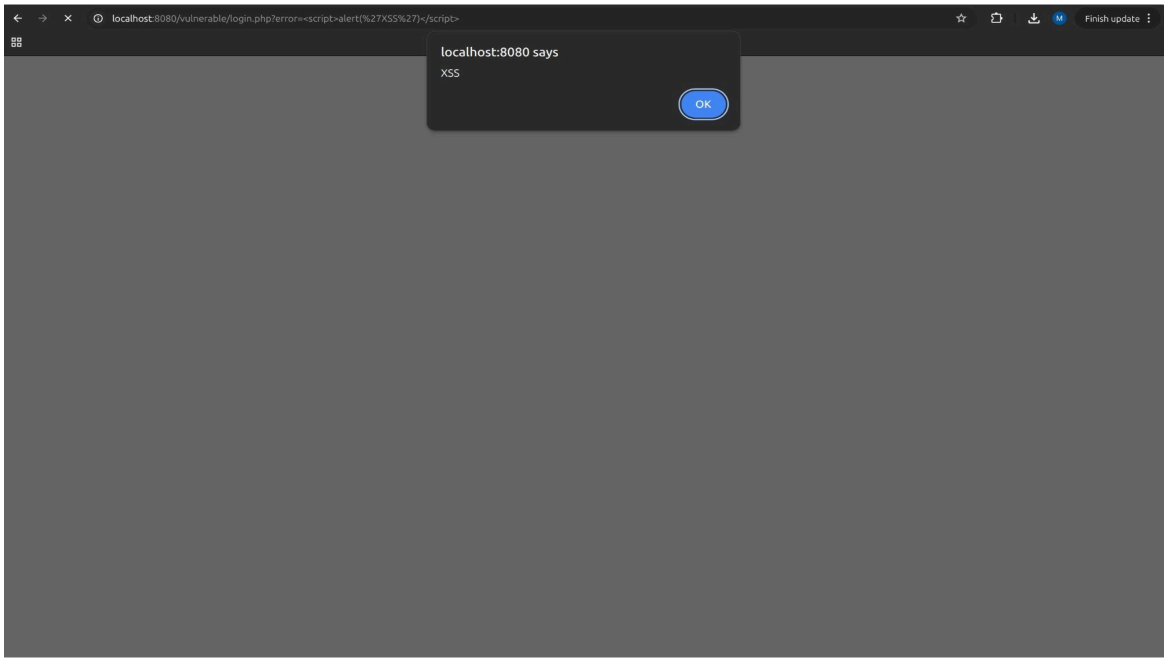


```
http://localhost:8080/login.php?error=<script>alert('XSS')</script>
http://localhost:8080/login.php?error=<script>alert(%27XSS%27)</script>
http://localhost:8080/login.php?error=<script>alert('XSS')</script> - Google Search
```

Figure 5: Crafting the XSS payload URL in the browser address bar

4.3 Result: JavaScript Execution

The browser executed the injected JavaScript. A popup alert appeared displaying "XSS", proving that arbitrary code runs in the victim's browser session. In a real attack, this could steal session cookies, redirect the user, or inject a phishing form.



```
localhost:8080/vulnerable/login.php?error=<script>alert(%27XSS%27)</script>
```

localhost:8080 says

XSS

Figure 6: JavaScript alert popup — XSS successfully executed on vulnerable version

After dismissing the popup, the login page renders with an empty error box where the script tag was injected (the script executed invisibly rather than displaying as text):

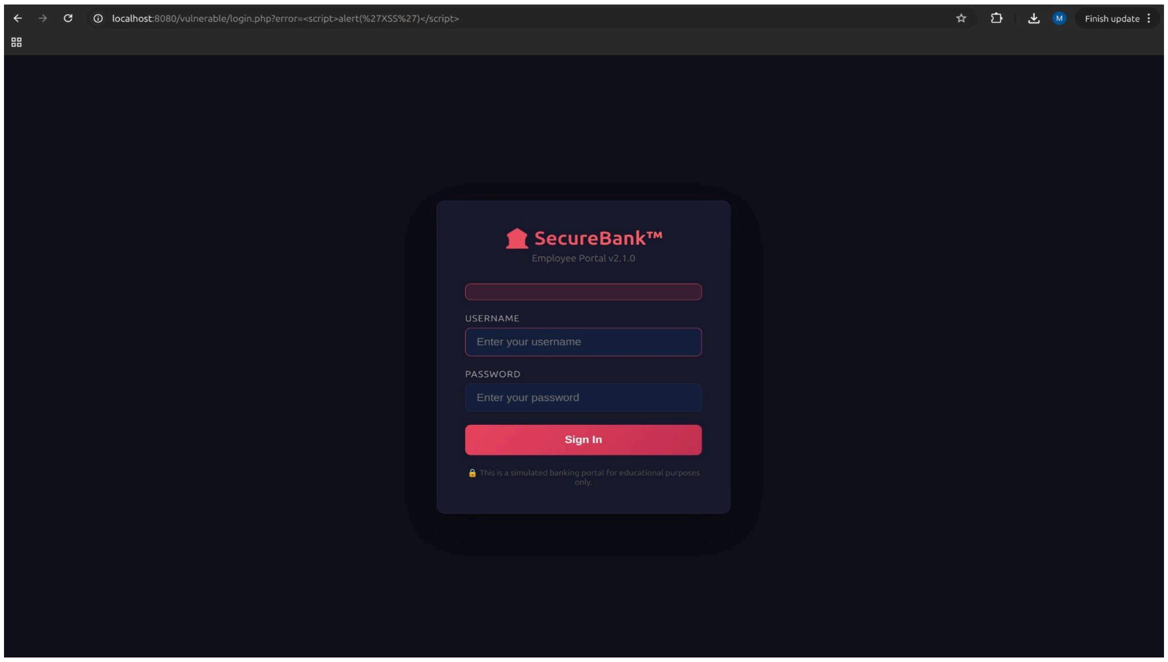

localhost:8080/vulnerable/login.php?error=<script>alert(%27XSS%27)</script>
Finish update
SecureBank™
Employee Portal v2.1.0
USERNAME
Enter your username
PASSWORD
Enter your password
Sign In
This is a simulated banking portal for educational purposes only.

Figure 7: Login page after XSS — empty error box where the script was injected

5. Secure Version: Verifying the Defenses

The same attacks were repeated against the secure version (port 9090) to prove that the implemented defenses effectively neutralize both vulnerabilities.

5.1 SQL Injection Blocked

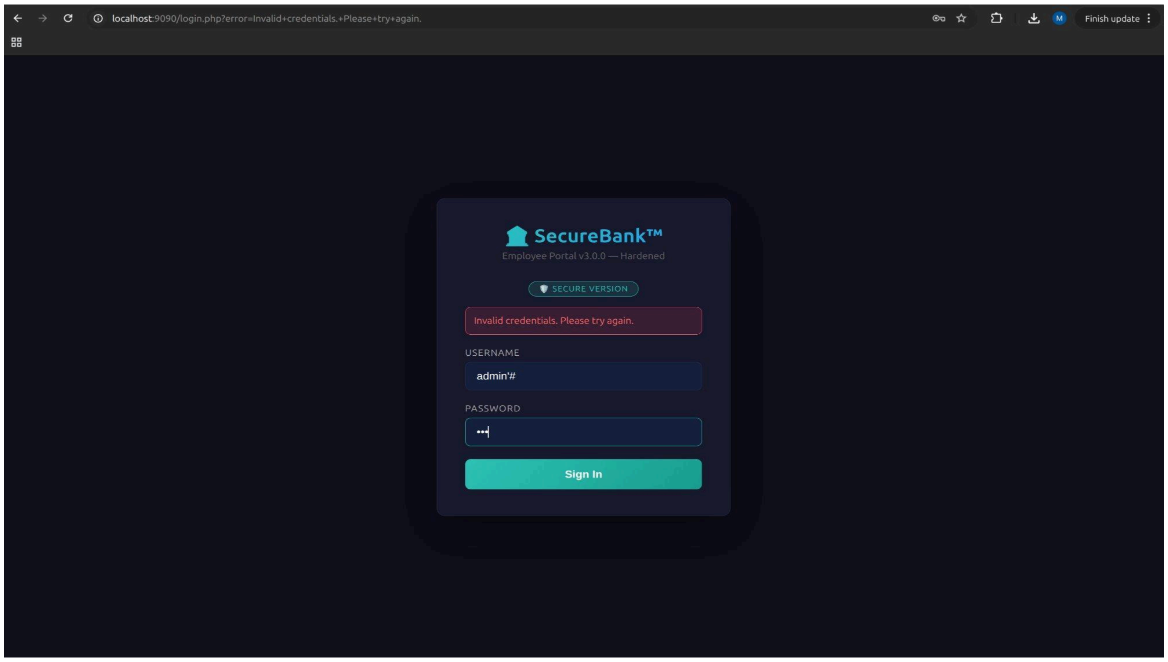
The secure version uses prepared statements with parameter binding:

```
$stmt = $conn->prepare("SELECT ... WHERE username = ? AND password = ?");
$stmt->bind_param("ss", $username, $password);
```

The database engine receives the query structure and data through separate channels. The input admin'# is treated as a literal string — the database searches for a user whose username is literally "admin'#" (with the quote and hash as part of the name). No such user exists, so authentication fails.

Payload used: admin'# / anything

Result: "Invalid credentials. Please try again."



localhost:9090/login.php?error=Invalid+credentials.+Please+try+again.

Invalid credentials. Please try again.

USERNAME
admin'#

PASSWORD
•••

Figure 8: SQL injection blocked — prepared statements treat input as data, not SQL code

5.2 XSS Blocked

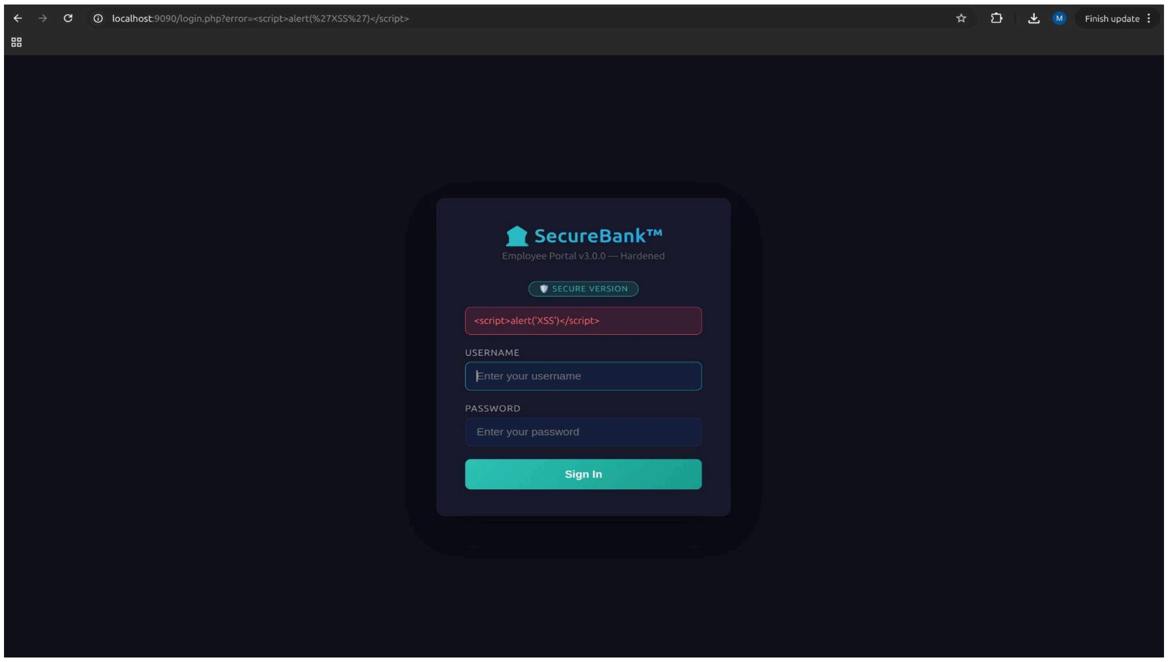
The secure version encodes all output with htmlspecialchars():

```
echo htmlspecialchars($_GET['error'], ENT_QUOTES, 'UTF-8');
```

This converts < into &lt; and > into &gt;, so the browser renders the script tag as visible text instead of executing it.

Payload used: <script>alert('XSS')</script> in the error parameter

Result: Script tags displayed as harmless text — no JavaScript executed


localhost:9090/login.php?error=<script>alert(%27XSS%27)</script>

<script>alert('XSS')</script>

Figure 9: XSS neutralized — script tags rendered as visible text, not executed

6. Technical Analysis: Why the Defenses Work

6.1 SQL Injection: The Root Cause

SQL injection occurs because of a trust boundary violation. When user input is concatenated into a SQL string, the database parser cannot distinguish between data and instructions. The attacker's input becomes part of the query's logic.

Prepared statements solve this architecturally. The query template (with ? placeholders) is sent to the database and compiled before the data arrives. By the time the user input is bound via bind_param(), the query plan is locked. The input can only fill data slots — it can never alter the query's structure. This makes injection impossible by design, not by filtering.

6.2 XSS: The Root Cause

Reflected XSS occurs when user-controlled data is inserted into HTML output without encoding. The browser interprets the injected content as part of the page's markup and executes any scripts it contains.

htmlspecialchars() neutralizes this by converting HTML-significant characters into harmless entities. The browser renders &lt;script&gt; as the visible text "<script>" instead of treating it as executable code. Combined with Content Security Policy headers (which restrict script sources), this creates a layered defense.

6.3 Defense in Depth

The secure version implements multiple layers beyond the primary fixes:

- Low-privilege database user (labuser) — cannot DROP tables even if injection occurred
- Security headers: Content-Security-Policy, X-Frame-Options, X-Content-Type-Options
- Session regeneration after login (prevents session fixation)
- Generic error messages (no internal details leaked)
- Input length validation as an additional safeguard
- Strict SQL mode enabled to prevent truncation attacks

7. Conclusion

This lab demonstrated the practical exploitation of two critical web vulnerabilities on a custom-built application. The SQL injection attack bypassed authentication entirely, granting full admin access without credentials. The reflected XSS attack achieved arbitrary JavaScript execution in the victim's browser context.

Both attacks were completely neutralized in the secure version through prepared statements (SQL injection) and output encoding (XSS), proving that these defenses are effective when correctly implemented. The additional defense-in-depth measures (low-privilege DB user, security headers, session management) provide fallback protection in case any single control is bypassed.

The fundamental principle underlying both vulnerabilities is the same: never trust user input. Any data crossing a trust boundary must be treated as potentially hostile and handled through appropriate mechanisms — parameterized queries for SQL, output encoding for HTML, and validation everywhere.