Build Your Own Vulnerable Web App Module

Course: Secure Programming / Web Security

Student: Isha Bachhav

Title

Build Your Own Vulnerable Web App Module — Demonstration of SQL Injection (vulnerable) and its Fix (prepared statements)

Abstract

This report describes the creation of a deliberately vulnerable PHP login page that demonstrates a classic SQL Injection (SQLi) vulnerability, the steps used to exploit it (with payloads and proof), and the secure, fixed version using parameterized queries (PDO prepared statements). The project includes both the vulnerable and the fixed source code, instructions to reproduce the setup locally, screenshots to prove exploitation, and a short discussion of secure coding practices and defenses. The goal is to show understanding of insecure coding and to demonstrate remediation.

Objectives

- Create a simple PHP web page intentionally vulnerable to SQL Injection.
- Demonstrate exploitation with clear step-by-step proof (screenshots).
- Implement a secure fixed version using parameterized queries.
- Document the process and provide a short security analysis and mitigation recommendations.

Tools & Environment

- OS: Any (Windows with XAMPP or Linux with LAMP tested)

- Web server: Apache (bundled with XAMPP / LAMP)
- PHP: 7.x or 8.x
- Database: MySQL / MariaDB
- Browser: Chrome / Firefox
- Optional tools: Burp Suite (free edition), sqlmap (for automation)

Database schema and sample data

Run the following SQL on your MySQL server (phpMyAdmin or CLI):

```
CREATE DATABASE sqli_demo;
USE sqli_demo;

CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(50) UNIQUE NOT NULL,
  password VARCHAR(255) NOT NULL
);

-- sample user (password stored in plaintext for demo only)
INSERT INTO users (username, password) VALUES ('alice', 'alice123');
```

> **Note:** For the vulnerable demo, the password is stored in plaintext to keep the example simple. For production code always store salted hashes (e.g., password_hash() / password_verify() in PHP).

File structure (for XAMPP htdocs or /var/www/html)

```
/htdocs/sqli_demo/
  ├─ vulnerable/
  │   ├─ index.php      (login form + vulnerable processing)
  │   └─ db.php         (database connection)
  └─ fixed/
      ├─ index.php      (login form + secure processing)
      └─ db.php         (database connection using PDO)
```

Vulnerable version (vulnerable/index.php)

This intentionally vulnerable login takes username and password from POST and embeds them directly into an SQL query without validation or parameterization.

```
<?php
// vulnerable/db.php
$host = 'localhost';
$db   = 'sqli_demo';
$user = 'root';
$pass = ''; // your local root password

$mysqli = new mysqli($host, $user, $pass, $db);
if ($mysqli->connect_error) {
    die('Connect Error ('. $mysqli->connect_errno .') '. $mysqli->connect_error);
}
?>

<!-- vulnerable/index.php -->
<?php
require 'db.php';

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $username = $_POST['username'];
    $password = $_POST['password'];

    // INSECURE: directly concatenate user inputs into query
    $sql = "SELECT * FROM users WHERE username = '$username' AND password = '
$password'";
    $result = $mysqli->query($sql);

    if ($result && $result->num_rows > 0) {
        echo "<h3>Login successful — welcome " . htmlspecialchars($username) . "</h3>";
    } else {
        echo "<h3>Login failed</h3>";
    }
}
?>

<form method="post">
  <label>Username: <input name="username"></label><br>
  <label>Password: <input name="password" type="password"></label><br>
  <button type="submit">Login</button>
</form>
```

Why this is vulnerable

The values of $_POST['username'] and $_POST['password'] are embedded into the SQL query string without any escaping or parameterization. An attacker can craft a payload to change the SQL logic.

Exploitation (how to demonstrate)

1. Start XAMPP / Apache + MySQL and place the vulnerable folder in htdocs.
2. Open browser to http://localhost/sqli_demo/vulnerable/index.php.
3. For username field, enter the payload:

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

Leave the password blank or enter anything. Submit.

Explanation: The SQL becomes:

```
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = ''
```

Because '1'='1' is always true, the WHERE clause matches rows and login bypasses authentication.

4. Capture screenshot(s):
    - Browser showing successful login message.
    - Browser address bar showing URL.
    - Optional: Show query in server logs or use echo $sql; in code (temporarily) to display the final SQL string for proof.

Suggested screenshot filenames: vuln_login_success.png, vuln_sql_echo.png.

Using sqlmap (optional automated proof)

If you want to show automated proof, run sqlmap against the local page (replace --data payload accordingly). Example:

```
sqlmap -u "http://localhost/sqli_demo/vulnerable/index.php" --data="username=alice&password=anything" --risk=2 --level=2 --batch
```

This should identify injectable parameters and (optionally) dump the users table. Capture sqlmap output as proof.

Fixed version (fixed/index.php) — using PDO prepared statements

Use PDO and parameterized queries to prevent SQL Injection.

```
<?php
// fixed/db.php
$host = 'localhost';
$db   = 'sqli_demo';
$user = 'root';
$pass = '';
$dsn  = "mysql:host=$host;dbname=$db;charset=utf8mb4";

try {
    $pdo = new PDO($dsn, $user, $pass, [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    ]);
} catch (PDOException $e) {
    die('DB Connection failed: ' . $e->getMessage());
}
?>

<!-- fixed/index.php -->
<?php
require 'db.php';

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $username = $_POST['username'];
    $password = $_POST['password'];

    // SECURE: use prepared statements
    $stmt = $pdo->prepare('SELECT * FROM users WHERE username = :username AND
 password = :password');
    $stmt->execute([':username' => $username, ':password' => $password]);
    $user = $stmt->fetch();

    if ($user) {
        echo "<h3>Login successful — welcome " . htmlspecialchars($username) . "</h3>";
    } else {
        echo "<h3>Login failed</h3>";
    }
}
```

```
?>

<form method="post">
  <label>Username: <input name="username"></label><br>
  <label>Password: <input name="password" type="password"></label><br>
  <button type="submit">Login</button>
</form>
```

> **Notes:** - Parameterized queries separate code and data. Even if the user supplies `' OR '1'='1`, it will be treated as a string, not SQL. - For real systems, do not store plaintext passwords: use `password_hash()` when creating users and `password_verify()` on login.

Reproducing the fix and testing

1. Place the `fixed` folder in `htdocs`.
2. Open `http://localhost/sqli_demo/fixed/index.php`.
3. Try same payload `' OR '1'='1` — it must fail.
4. Login with valid credentials (e.g., `alice` / `alice123`) to show correct authentication.
5. Capture screenshots: `fixed_login_fail.png`, `fixed_login_success.png`.

Report of Changes (vulnerable -> fixed)

Item	Vulnerable implementation	Fixed implementation
Query building	String concatenation with raw inputs	PDO prepared statements (parameterized)
Input handling	None	Parameters bound, HTML output encoded with `htmlspecialchars()`
Password storage	Plaintext (for demo)	Recommendation: use `password_hash()` / `password_verify()`

Security Analysis & Recommendations

- **Use parameterized queries** for all database access. Never build SQL by concatenating user input.
- **Validate and sanitize input** at boundaries (both server- and client-side). But validation is not a substitute for parameterized queries.
- **Use proper password storage**: password_hash() and password_verify() in PHP.
- **Least privilege DB user**: Create a database user with only required privileges (SELECT, INSERT) instead of root.
- **Error handling**: Do not show detailed DB errors to users. Log them instead.
- **Use a web application firewall (WAF)** as an additional layer (optional for this assignment).

Screenshots & Proof (placeholders)

Include the following screenshots in your submission (insert actual images into the report file): 1. vuln_login_success.png — browser showing successful login with SQLi payload. 2. vuln_sql_echo.png — (optional) server output showing constructed SQL string (if you temporarily echo it during demo). 3. sqlmap_output.txt — (optional) output from sqlmap if used. 4. fixed_login_fail.png — browser showing failed login when using the SQLi payload against the fixed app. 5. fixed_login_success.png — successful login with correct credentials on fixed app.

How to take screenshots: Use PrtSc or Snipping Tool (Windows) or gnome-screenshot / gnome-screenshot -a (Linux) and save into project folder.

Appendix A — Full Source Code Files

(Include files from the vulnerable and fixed folders. The full code was presented earlier; include them verbatim in the appendix of your submitted report.)

Appendix B — Assessment Checklist (for instructor)

- ☐ Project objective clearly stated
- ☐ Vulnerable version provided and described
- ☐ Exploitation demonstrated (payloads and screenshots)
- ☐ Fixed version provided and explained
- ☐ Secure coding best practices noted
- ☐ Reproducible steps included (environment, SQL schema)
- ☐ All relevant files included in submission (source code + screenshots)

References

- OWASP SQL Injection: https://owasp.org/www-community/attacks/SQL_Injection
- PHP PDO documentation: https://www.php.net/manual/en/book.pdo.php