
Project 4: Build Your Own Vulnerable Web Application Module

Objective

The objective of this project is to understand insecure coding practices in web applications and demonstrate how improper input handling can lead to serious security vulnerabilities such as **SQL Injection (SQLi)**. This project also illustrates how the identified vulnerability can be mitigated using **secure coding techniques**.

Tools & Environment Used

- **Operating System:** Windows 10
- **Web Server:** XAMPP (Apache + MySQL)
- **Programming Language:** PHP
- **Database:** MySQL (MariaDB)
- **Database Tool:** phpMyAdmin
- **Browser:** Google Chrome

Application Overview

This application consists of a simple PHP-based login system connected to a MySQL database. The login functionality is intentionally implemented without input validation to demonstrate a SQL Injection vulnerability. A secure version of the same functionality is later implemented using prepared statements.

Vulnerability Chosen

SQL Injection (SQLi)

SQL Injection occurs when user-supplied input is directly embedded into SQL queries without proper validation or sanitization. This allows attackers to manipulate SQL queries, potentially bypassing authentication and gaining unauthorized access to the application.

Project Setup

1. Database Creation

- Database name: *project4*
- Table name: *users*

Table Structure:

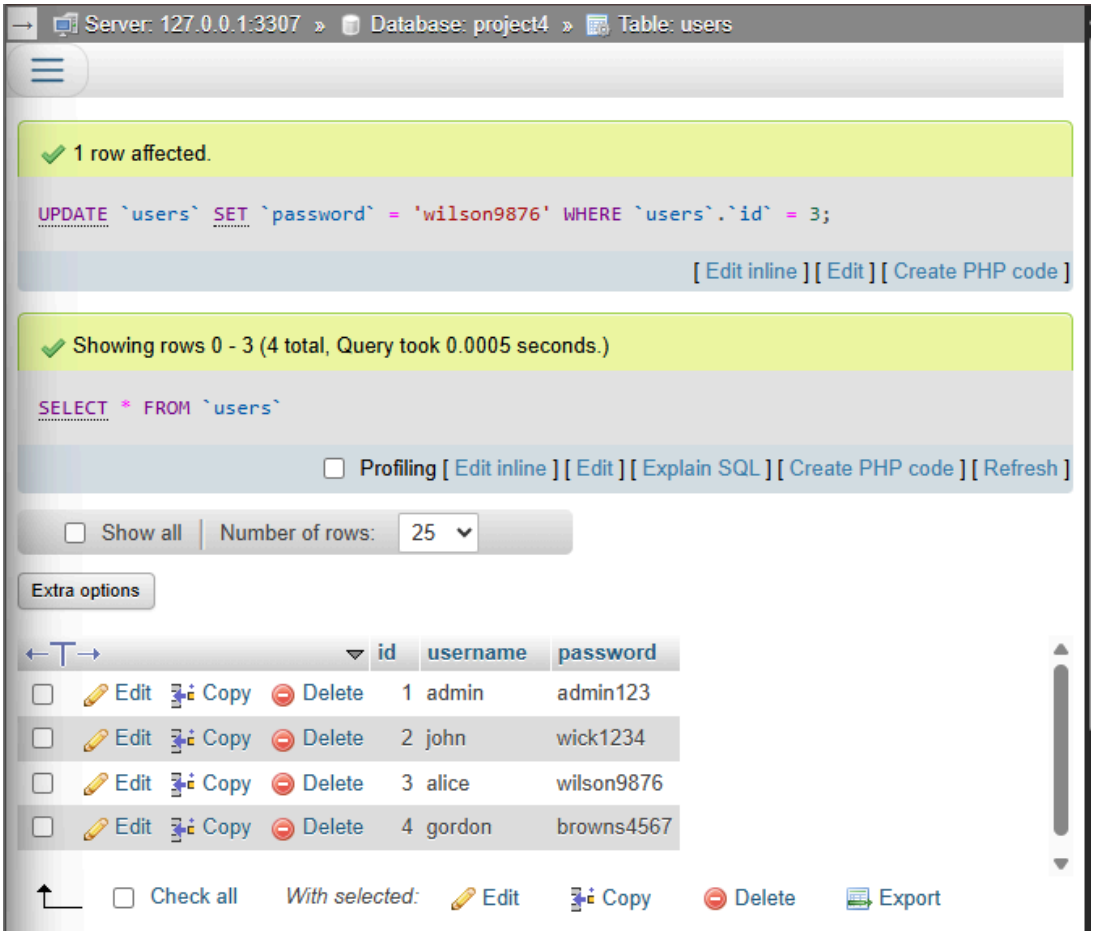
Column Name	Data Type
id	INT (Primary Key, Auto Increment)
username	VARCHAR(50)
password	VARCHAR(50)

---

2. Sample Users Added

To simulate a real-world application scenario, multiple user records were added to the database:

Username	Password
admin	admin123
john	wick1234
alice	wilson9876
gordon	browns4567



Users table in phpMyAdmin

Part 1: Vulnerable Login Implementation

Vulnerable Code (login_vulnerable.php)

```
<?php
$conn = new mysqli("localhost", "root", "", "project4", 3307);

$username = $_POST['username'];
$password = $_POST['password'];

$sql = "SELECT * FROM users WHERE username='$username' AND
password='$password'";
$result = $conn->query($sql);

if ($result->num_rows > 0) {
    echo "Login Successful!";
} else {
```

```
    echo "Login Failed!";
}
?>
```

Issue

- User input is directly concatenated into the SQL query.
- No input validation or prepared statements are used.
- This allows attackers to inject malicious SQL code.

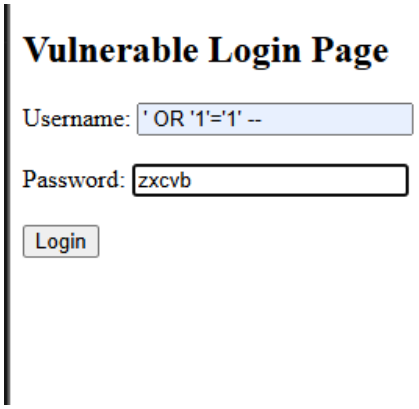
Exploitation of Vulnerability

SQL Injection Payload Used

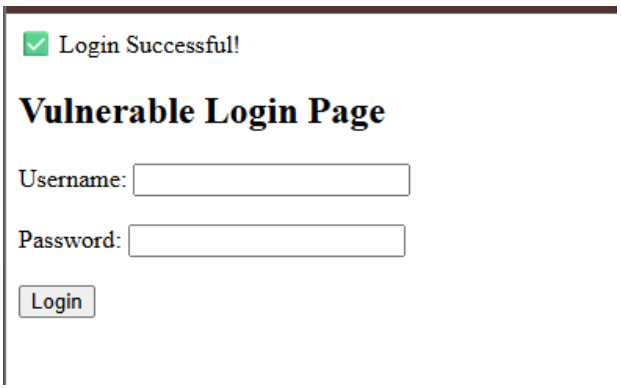
`' OR '1'='1' --`

Result

- Login is successful without valid credentials.
- Authentication is completely bypassed.



SQL Injection payload entered



Successful login message after exploitation

Part 2: Fixed (Secure) Implementation

Secure Code (login_secure.php)

```
<?php
$conn = new mysqli("localhost", "root", "", "project4", 3307);

if ($_SERVER["REQUEST_METHOD"] == "POST") {

    $username = $_POST['username'];
    $password = $_POST['password'];

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

    $stmt->execute();
    $result = $stmt->get_result();

    if ($result->num_rows > 0) {
        echo "Login Successful!";
    } else {
        echo "Login Failed!";
    }
} else {
    echo "Please submit the login form.";
}
?>
```

Fix Applied

- Implemented **prepared statements**
- User input is treated strictly as data, not executable SQL
- SQL Injection attempts are effectively blocked

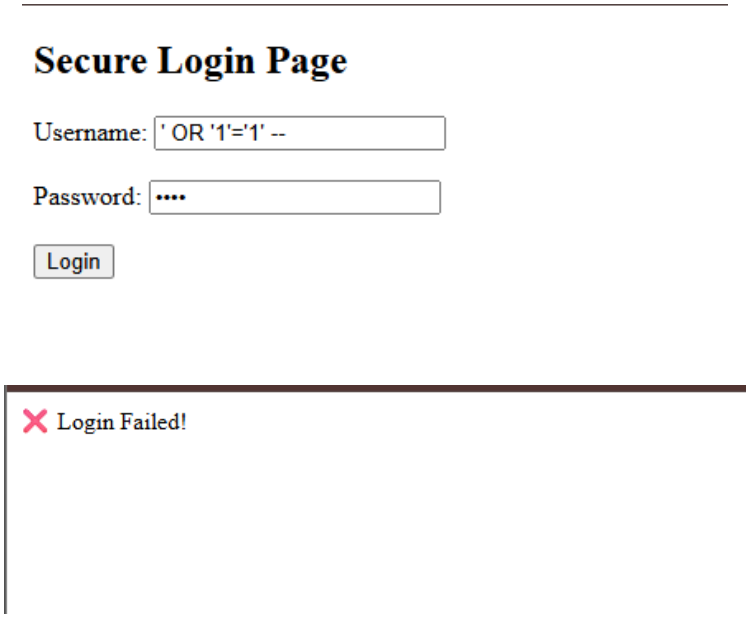
Verification After Fix

Injection Attempt After Fix

Username: ' OR '1'='1' --
Password: test

Result

Login Failed
(SQL Injection no longer works)



Injection attempt failed on secure page

Project Outcome

- Successfully demonstrated a real-world SQL Injection vulnerability.
- Showed how attackers can bypass authentication in insecure applications.

- Implemented secure coding practices to prevent SQL Injection.
- Gained practical understanding of input handling and parameterized queries.

Key Learnings

- Never trust user input.
- Avoid dynamic SQL queries.
- Always use prepared statements for database interactions.
- Secure coding practices are essential to protect web applications.

Conclusion

This project demonstrates how insecure coding practices can expose web applications to critical vulnerabilities such as SQL Injection. By implementing prepared statements and secure input handling, the application becomes resistant to such attacks. The project highlights the importance of adopting secure development practices in real-world web applications.
