Project:4

Build Own Vulnerable Web Application

PHP SQL Injection Demonstration

This project focuses on building a simple vulnerable PHP web application to understand how SQL Injection occurs, how it can be exploited, and how it can be prevented using secure coding practices.

Objective

The objectives of this project were:

- To design a simple PHP-based login page
- To intentionally introduce an SQL Injection vulnerability
- To demonstrate how the vulnerability can be exploited
- To fix the vulnerability using secure coding techniques
- To compare vulnerable and secure implementations

Technology Used

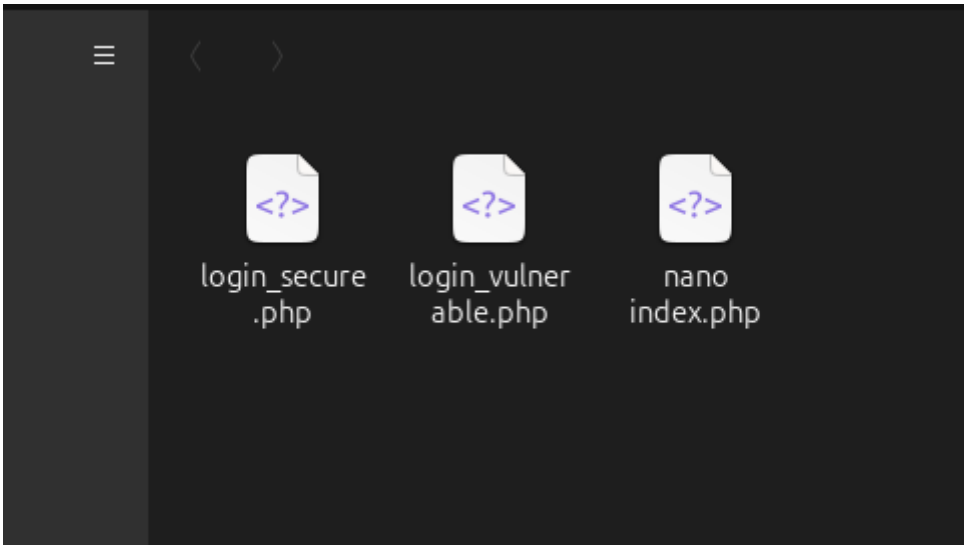
- PHP
- MySQL
- PHP Built-in Development Server
- Kali Linux
- Web Browser

Application Design Overview

A basic login system was developed using PHP and MySQL. The application accepts a username and password from the user and validates them against records stored in a database.

Two versions of the application were created:

- A vulnerable version without input validation
- A secure version using prepared statements



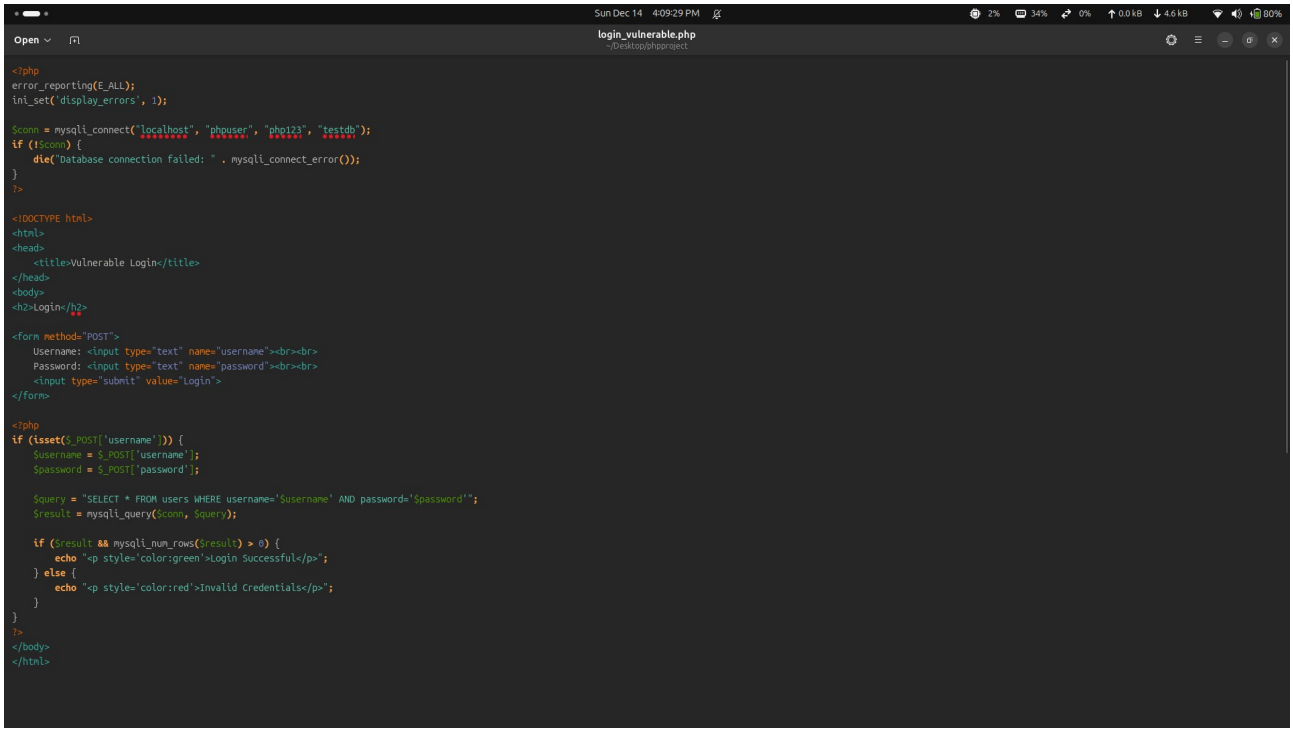
Vulnerable Application Implementation

The vulnerable login page directly concatenates user input into an SQL query. No validation or sanitization is performed before executing the query.

This insecure practice makes the application vulnerable to SQL Injection attacks.

Vulnerable PHP Code

File name: `login_vulnerable.php`



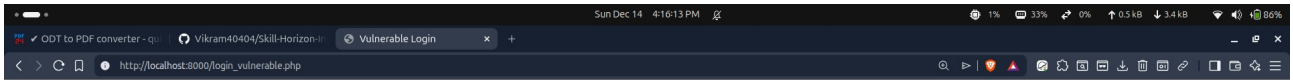
The vulnerable code constructs SQL queries using raw user input, allowing attackers to manipulate the query logic.

Exploitation of the Vulnerability

The SQL Injection vulnerability was exploited by entering a crafted payload in the username field.

Payload used:

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



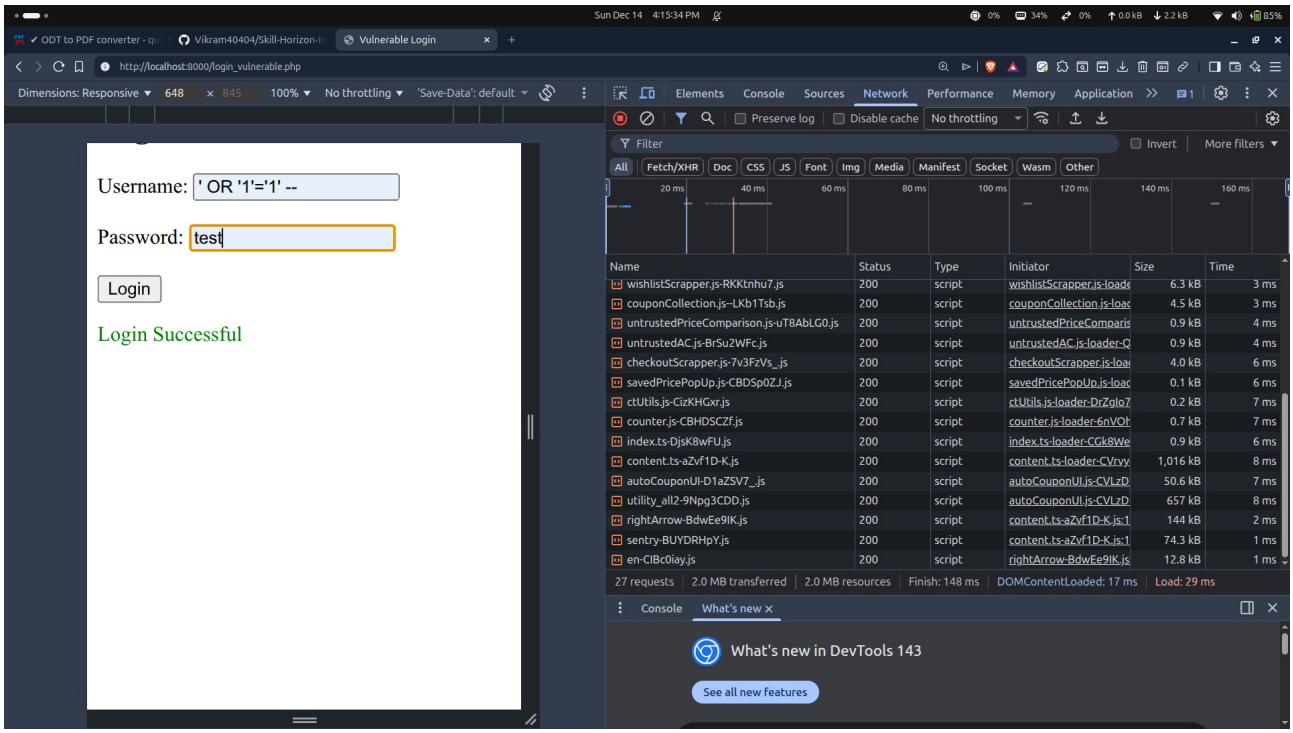
Login

Username: ' OR '1'='1' --

Password: test

Login

Login Successful



Observation:

- The SQL comment bypassed the password condition
- Authentication was successfully bypassed
- Login was granted without valid credentials

This confirms the presence of an SQL Injection vulnerability.

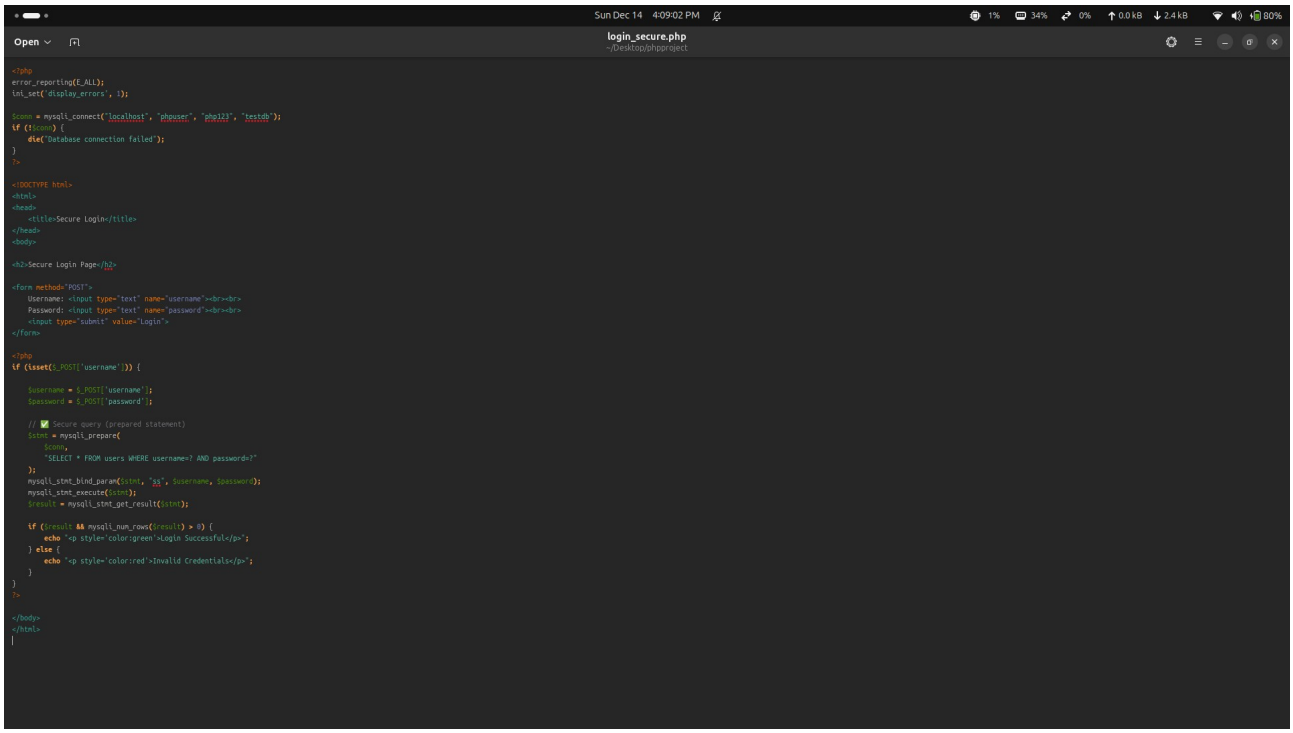
Secure Application Implementation

To fix the vulnerability, prepared statements were implemented. Prepared statements separate SQL code from user-supplied data, ensuring that input cannot alter query structure.

This approach follows secure coding best practices.

Secure PHP Code

File name: `login_secure.php`



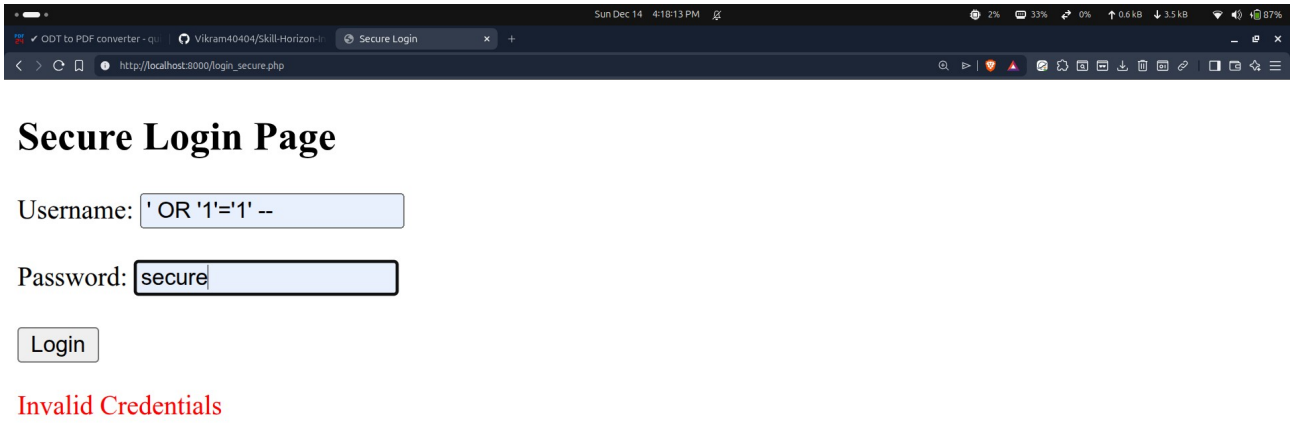
The secure version uses parameterized queries to safely handle user input and prevent SQL Injection attacks.

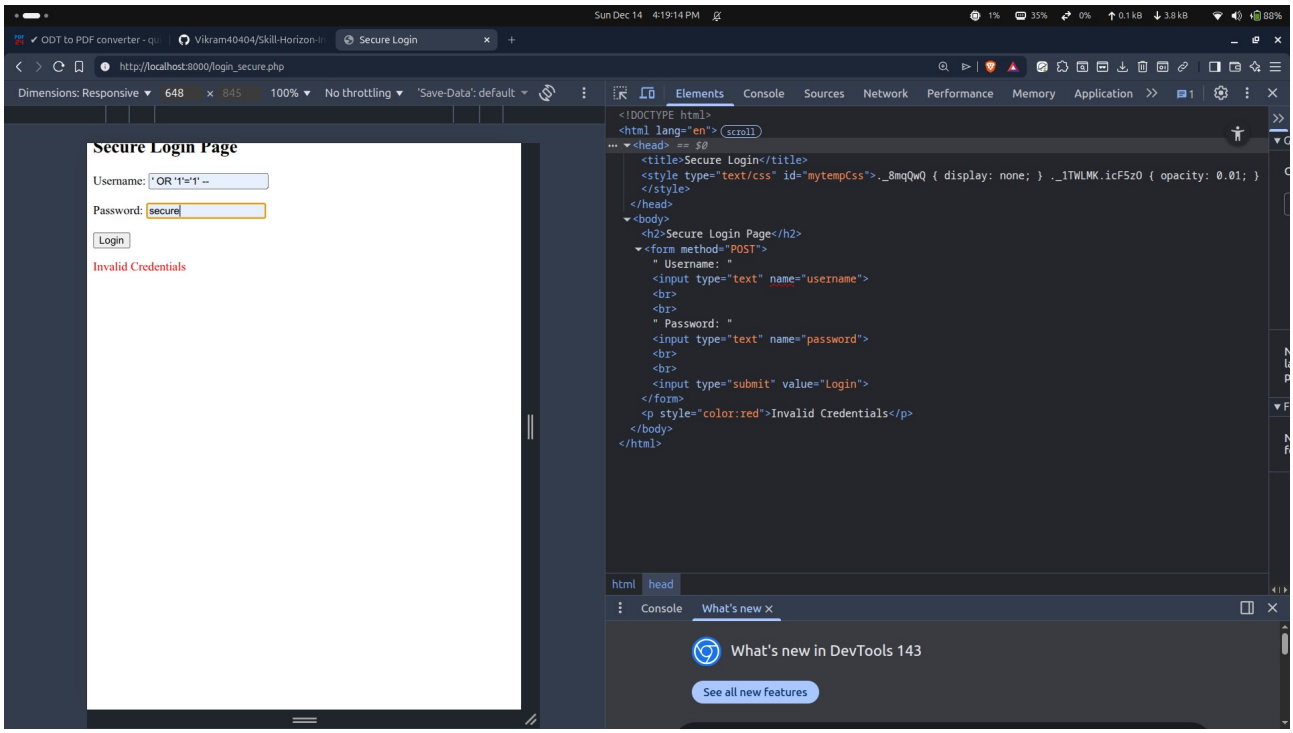
Verification of Fix

The same SQL Injection payload used on the vulnerable version was tested against the secure version.

Payload tested:

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





Observation:

- The payload was treated as plain text
- Authentication bypass was not possible
- Login failed as expected

This confirms that the vulnerability was successfully mitigated.

Comparison of Vulnerable and Secure Versions

- The vulnerable version allows direct manipulation of SQL queries
- The secure version strictly separates input data from SQL logic
- SQL Injection works in the vulnerable version but fails in the secure version
- Prepared statements effectively eliminate the attack vector

---

Conclusion

This project demonstrated how insecure coding practices can introduce critical vulnerabilities into web applications. A simple login page was shown to be vulnerable to SQL Injection due to improper handling of user input.

By implementing prepared statements, the same vulnerability was completely prevented. This highlights the importance of secure coding practices during application development rather than relying solely on post-deployment security testing.

Learning Outcomes

- Understanding how SQL Injection vulnerabilities occur
- Practical experience in exploiting and fixing vulnerabilities
- Knowledge of secure database access using prepared statements
- Awareness of common insecure coding practices
- Ability to demonstrate vulnerability remediation with proof