Ethan Conner Aden Tessman
Phase 1

Selected vulnerability: **SQL injection**

General Information

SQL injection (SQLi) is one of the most common and extremely dangerous web application vulnerabilities. It occurs when an attacker manipulates a web application's SQL query by inserting (injecting) malicious SQL code into an input field. These malicious inputs can alter or change the behavior of the intended SQL behavior, allowing attackers to potentially bypass authentication, modify or delete sensitive data from the database, and gather information, among other things.

The nature of the vulnerability arises from user inputs being improperly filtered or sanitized (parameterized) before passing to direct SQL queries into the database. There are also different types of SQLi including: in-band SQLi (direct extraction), inferential SQLi (inferring information and observation), and out-of-band SQLi (different channels used to retrieve data or trigger behaviors, such as HTTP, DNS, etc.).

Exploitation of SQLi vulnerabilities by attackers can include: injection into queries, error-based exploitation; this involves purposefully using database errors to gather information about the database structure, blind SQLi; using delays or changes response time to infer information, exfiltrating data; this involves injecting SQL queries that combine data from multiple different tables to extract sensitive information, and privilege escalation; this could be using injections to manipulate the database's privilege system to gain restricted access.

Potential impacts of an SQLi injection attack can include: unauthorized data access, data manipulation, authentication bypassing, privilege escalation, denial of service (DoS), database compromise, reputation damage, and legal and regulatory consequences.

Mitigation methods to avoid SQLi vulnerabilities in web applications include: using parameterized queries (input values are bound to parameters rather than directly embedded in SQL queries), stored procedures; this limits dynamic query construction and is more secure with proper input validation, input validation and sanitization; this removes dangerous characters like ', ", –, etc., escaping special characters, this ensures user input does not conflict with SQL syntax and their input is treated as data rather than code, practicing principle of least privilege; this limits damage in an attack and can help separate attackers from important accounts, proper error handling; this ensures avoiding exposure of details about database through errors users can see on their screen, using web application firewalls; this is pattern recognition and attack pattern blocking implementations, have regular security audits; this helps identity and find vulnerabilities before they can be exploited, and limit data exposure; this involves only returning necessary data in query results, and avoiding overexposure of data or exposure of sensitive information in query responses.

Phase 2

Controlled Environment:

To ensure safety and integrity of our testing, we established a controlled environment running our Mysql database on a live Linux server hosted through Amazon Web Services (AWS). This setup allows us to closely monitor, manage and isolate database operations and tests to ensure issues or vulnerabilities exploited can stay contained and be reset to previous versions. To simplify this process we used phpmyadmin to create the database.

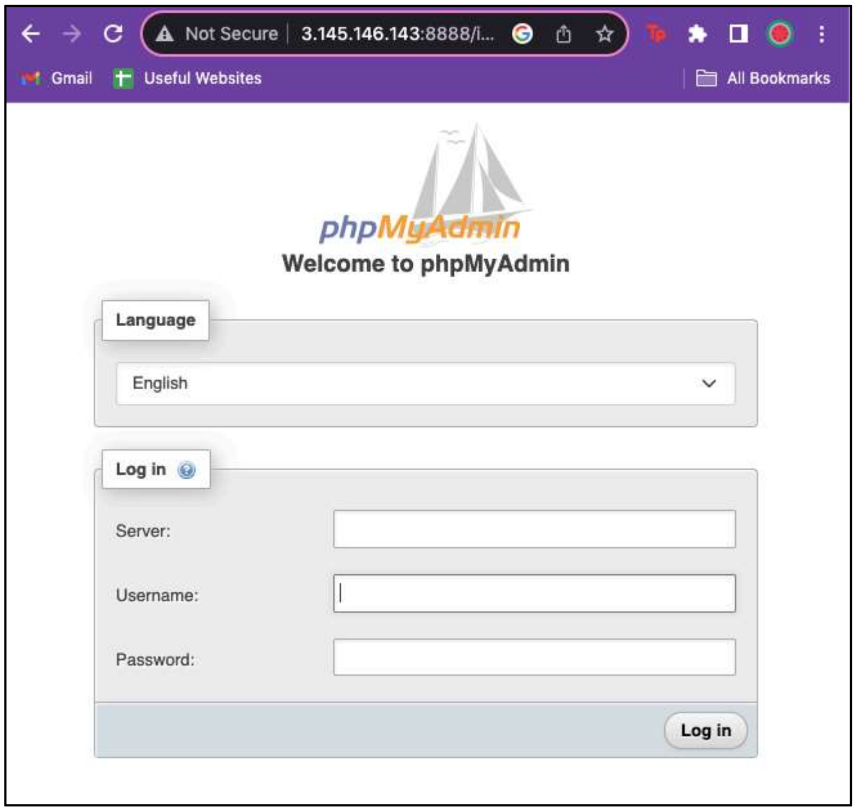


Figure 2.1 phpMyAdmin Login Page

About the Database

For our implementation we created a simple University Database. Our database consists of the two tables: Student (contains student information), users (contains usernames and passwords) . We then filled the database with various related columns to the table. Finally we filled our tables with random information for this demonstration. (Seen Below)

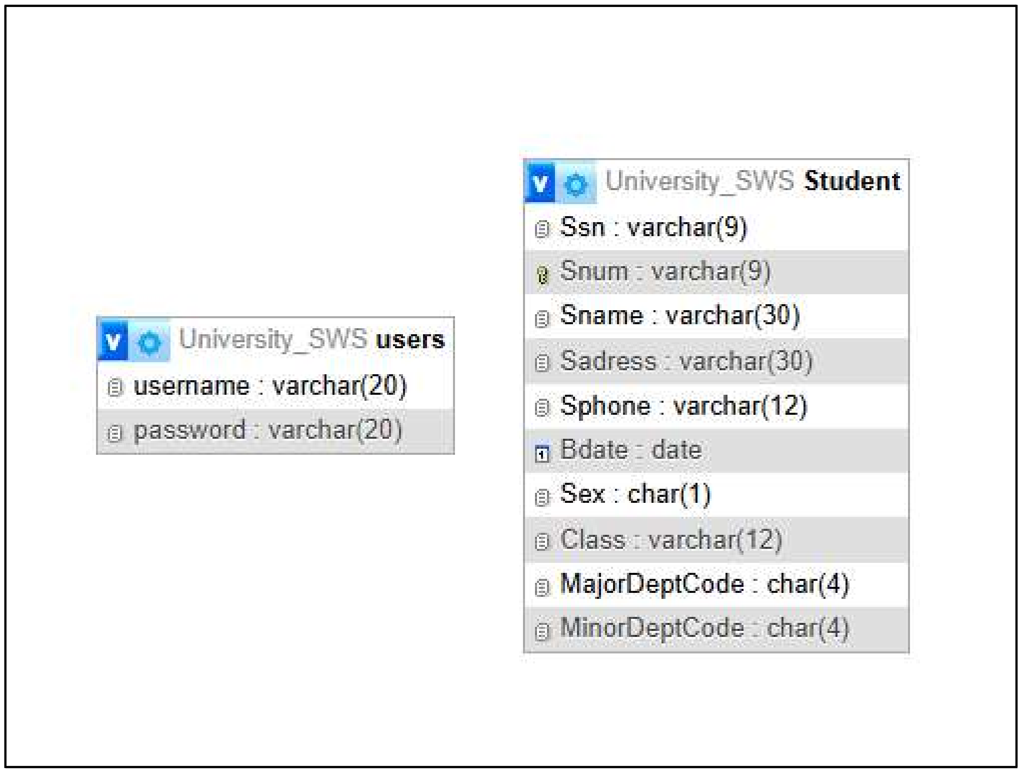


Figure 2.2 University Database Schema

Some of the tuples within the database consist of sensitive information such as students' SSN. Therefore precautions must be made when granting access to tables.

Injection Demonstration Environment

For this demonstration we created a simple html login page that communicates to a separate php file. The php file is then able to connect to the remote database and verify user login credentials.

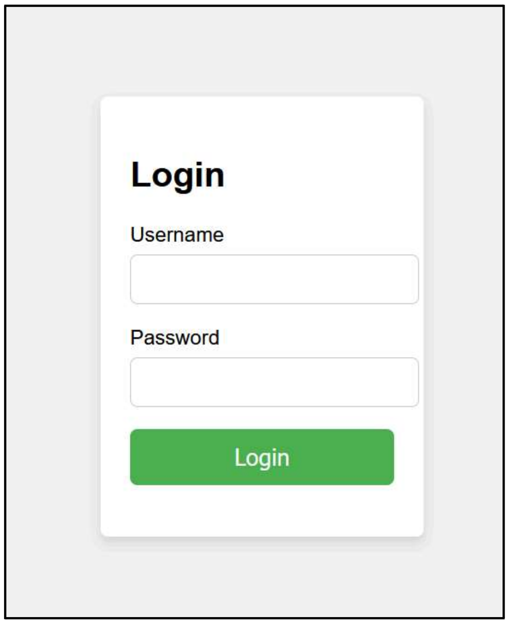
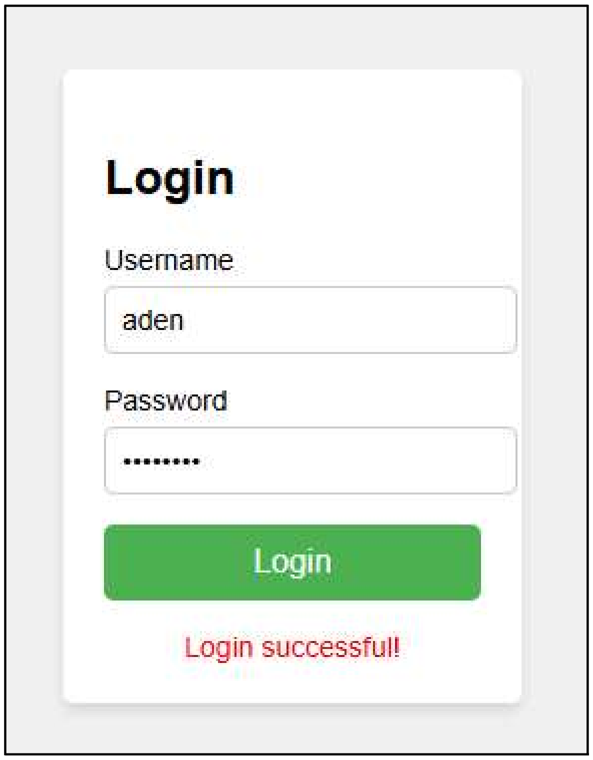
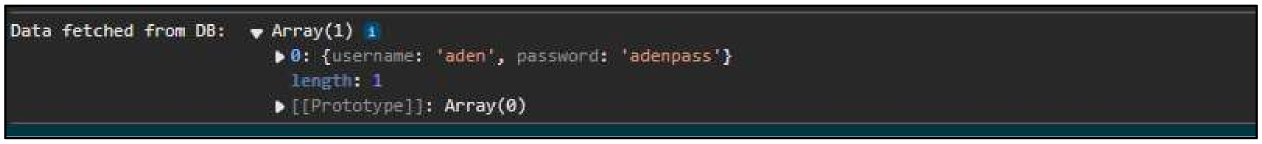


Figure 2.3 login page for University database

To keep track of the information being accessed we have the html log the JSON received from the php file in the console. When the program is utilized as intended, access is granted and the data received is just the credentials passed to the php file.





```
Data fetched from DB:  ▼ Array(1) i
                         ▶ 0: {username: 'aden', password: 'adenpass'}
                           length: 1
                         ▶ [[Prototype]]: Array(0)
```

Figure 2.4 successful login

This is done to show that this user's tuple exists within the database and that the user is allowed to login. However due to the limited amount of protections done in this example system, it can very easily be exploited.

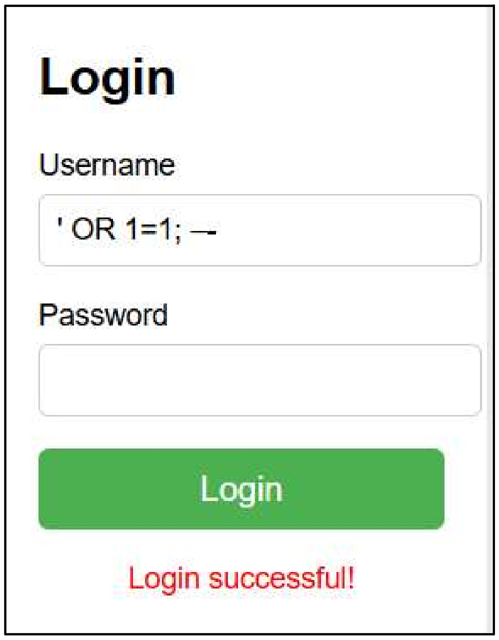
Exploitation:

We were able to create two injections. One to receive all usernames and passwords stored within the system, and a second one to receive all information from another unrelated table in the system with confidential information (SSNs).

Injection 1:

Input → Username: `' OR 1=1; --`

Password: (blank)



Login

Username

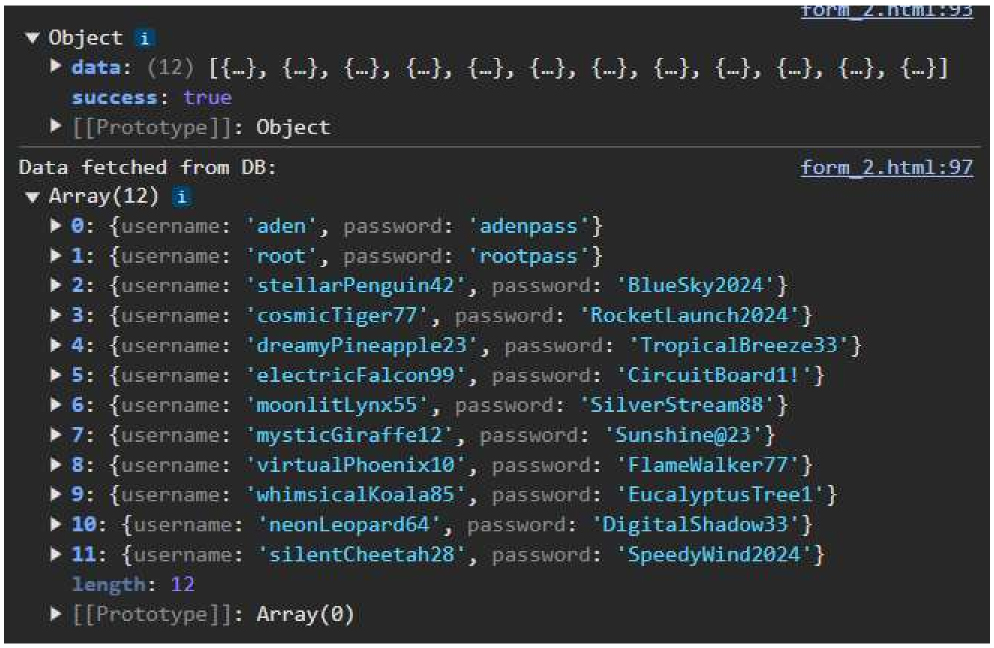
' OR 1=1; –-

Password

Login successful!

Figure 2.5 successful SQL injection

Output → table of usernames and passwords



```
                                                              form_2.html:93
▼ Object i
  ▶ data: (12) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
    success: true
  ▶ [[Prototype]]: Object
Data fetched from DB:                                         form_2.html:97
▼ Array(12) i
  ▶ 0: {username: 'aden', password: 'adenpass'}
  ▶ 1: {username: 'root', password: 'rootpass'}
  ▶ 2: {username: 'stellarPenguin42', password: 'BlueSky2024'}
  ▶ 3: {username: 'cosmicTiger77', password: 'RocketLaunch2024'}
  ▶ 4: {username: 'dreamyPineapple23', password: 'TropicalBreeze33'}
  ▶ 5: {username: 'electricFalcon99', password: 'CircuitBoard1!'}
  ▶ 6: {username: 'moonlitLynx55', password: 'SilverStream88'}
  ▶ 7: {username: 'mysticGiraffe12', password: 'Sunshine@23'}
  ▶ 8: {username: 'virtualPhoenix10', password: 'FlameWalker77'}
  ▶ 9: {username: 'whimsicalKoala85', password: 'EucalyptusTree1'}
  ▶ 10: {username: 'neonLeopard64', password: 'DigitalShadow33'}
  ▶ 11: {username: 'silentCheetah28', password: 'SpeedyWind2024'}
    length: 12
  ▶ [[Prototype]]: Array(0)
```

Figure 2.6 stolen table of usernames and passwords

Injection 2:

Input → Username: `' UNION SELECT Ssn, Sname FROM Student; --`

Password: (Blank)

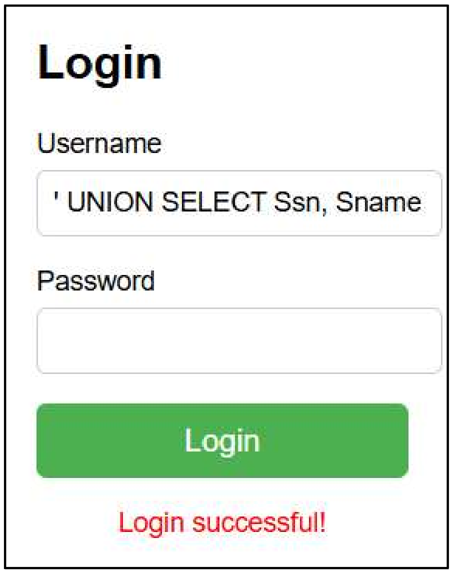
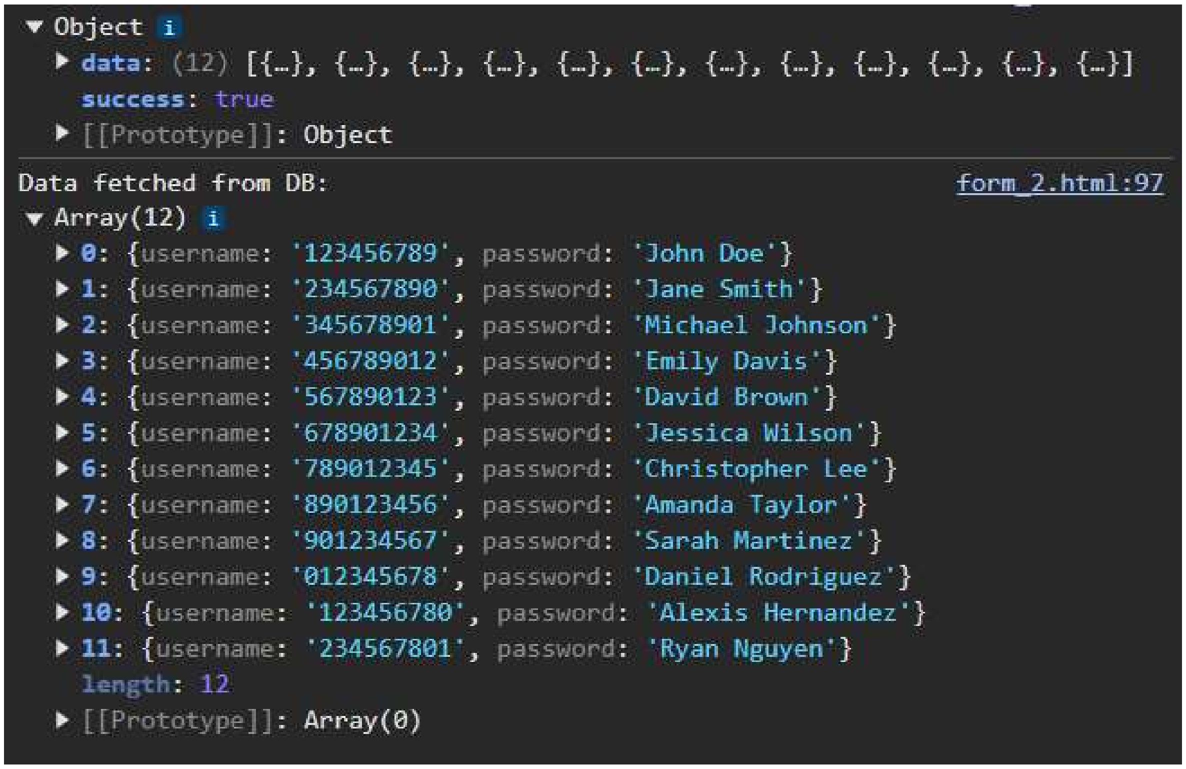


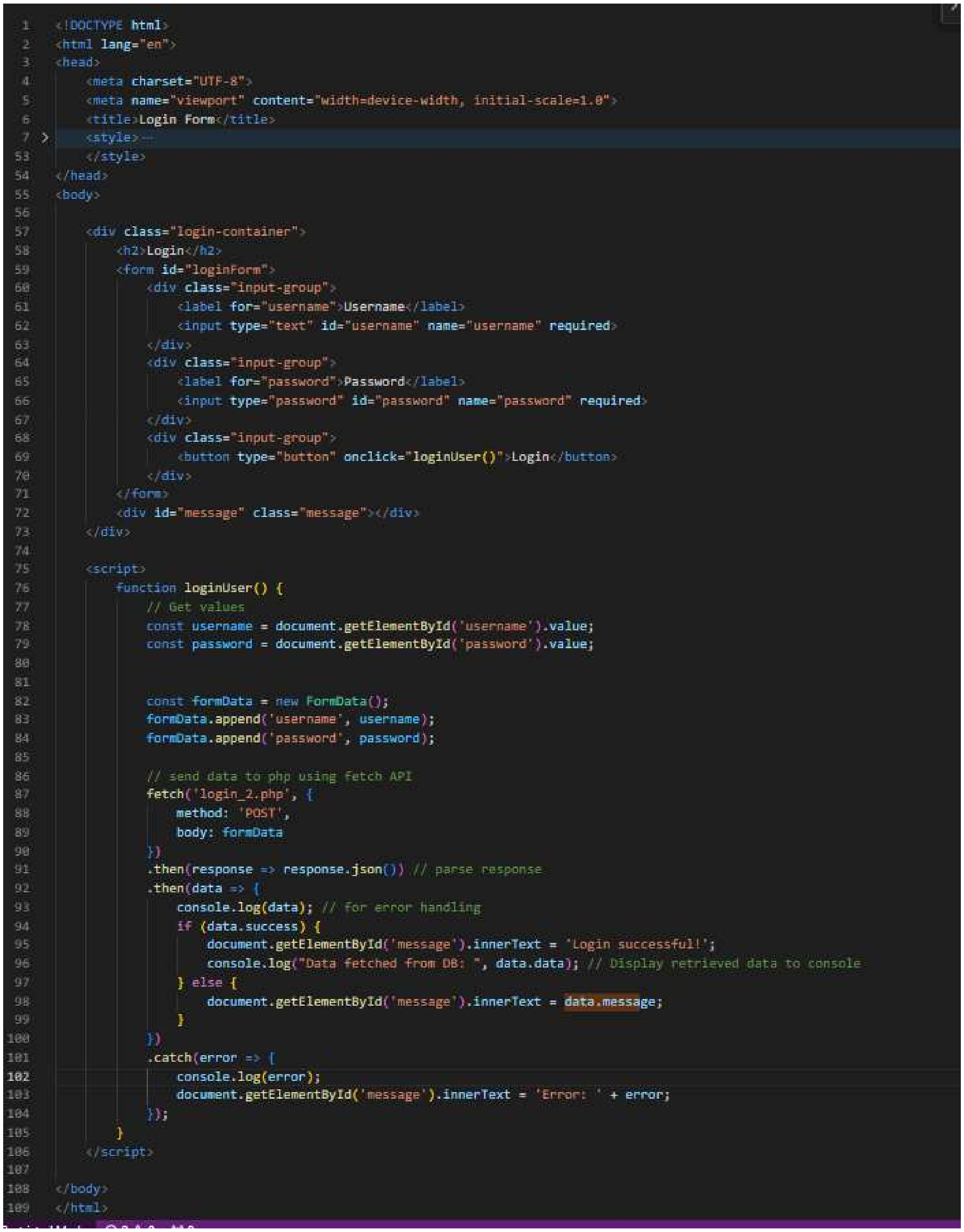
Figure 2.7 successful SQL injection

Output → Table of confidential student information, and authentication bypassing



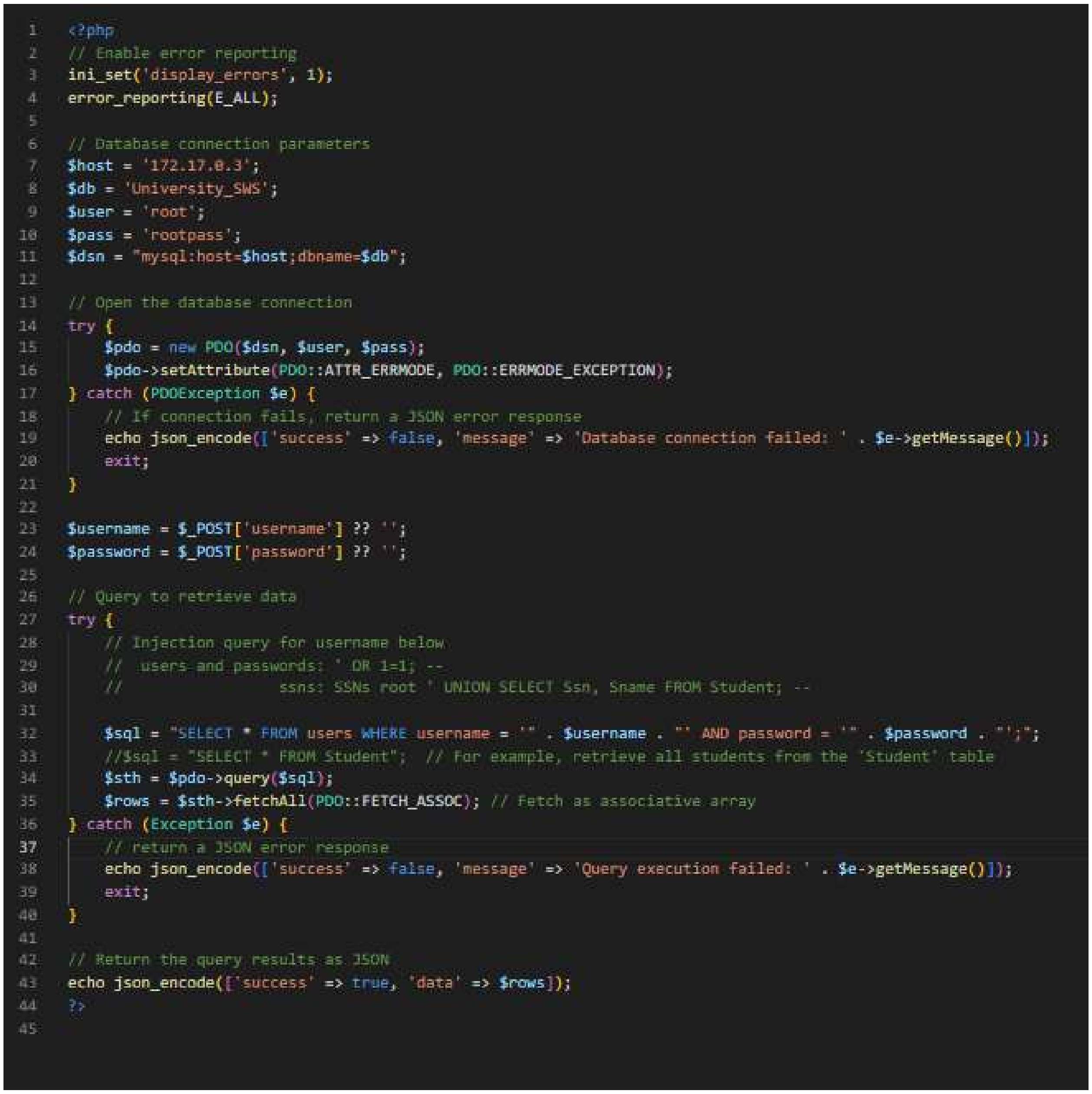
```
▼ Object i
  ▶ data: (12) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
    success: true
  ▶ [[Prototype]]: Object
Data fetched from DB:                                                    form_2.html:97
▼ Array(12) i
  ▶ 0: {username: '123456789', password: 'John Doe'}
  ▶ 1: {username: '234567890', password: 'Jane Smith'}
  ▶ 2: {username: '345678901', password: 'Michael Johnson'}
  ▶ 3: {username: '456789012', password: 'Emily Davis'}
  ▶ 4: {username: '567890123', password: 'David Brown'}
  ▶ 5: {username: '678901234', password: 'Jessica Wilson'}
  ▶ 6: {username: '789012345', password: 'Christopher Lee'}
  ▶ 7: {username: '890123456', password: 'Amanda Taylor'}
  ▶ 8: {username: '901234567', password: 'Sarah Martinez'}
  ▶ 9: {username: '012345678', password: 'Daniel Rodriguez'}
  ▶ 10: {username: '123456780', password: 'Alexis Hernandez'}
  ▶ 11: {username: '234567801', password: 'Ryan Nguyen'}
    length: 12
  ▶ [[Prototype]]: Array(0)
```

Figure 2.8 stolen student information (SSNs and names)



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Form</title>
    <style>···
    </style>
</head>
<body>

    <div class="login-container">
        <h2>Login</h2>
        <form id="loginForm">
            <div class="input-group">
                <label for="username">Username</label>
                <input type="text" id="username" name="username" required>
            </div>
            <div class="input-group">
                <label for="password">Password</label>
                <input type="password" id="password" name="password" required>
            </div>
            <div class="input-group">
                <button type="button" onclick="loginUser()">Login</button>
            </div>
        </form>
        <div id="message" class="message"></div>
    </div>

    <script>
        function loginUser() {
            // Get values
            const username = document.getElementById('username').value;
            const password = document.getElementById('password').value;


            const formData = new FormData();
            formData.append('username', username);
            formData.append('password', password);

            // send data to php using fetch API
            fetch('login_2.php', {
                method: 'POST',
                body: formData
            })
            .then(response => response.json()) // parse response
            .then(data => {
                console.log(data); // for error handling
                if (data.success) {
                    document.getElementById('message').innerText = 'Login successful!';
                    console.log("Data fetched from DB: ", data.data); // Display retrieved data to console
                } else {
                    document.getElementById('message').innerText = data.message;
                }
            })
            .catch(error => {
                console.log(error);
                document.getElementById('message').innerText = 'Error: ' + error;
            });
        }
    </script>

</body>
</html>
```

Figure 2.9 Original HTML file before SQLi mitigation



```php
<?php
// Enable error reporting
ini_set('display_errors', 1);
error_reporting(E_ALL);

// Database connection parameters
$host = '172.17.0.3';
$db = 'University_SWS';
$user = 'root';
$pass = 'rootpass';
$dsn = "mysql:host=$host;dbname=$db";

// Open the database connection
try {
    $pdo = new PDO($dsn, $user, $pass);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
    // If connection fails, return a JSON error response
    echo json_encode(['success' => false, 'message' => 'Database connection failed: ' . $e->getMessage()]);
    exit;
}

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

// Query to retrieve data
try {
    // Injection query for username below
    //  users and passwords: ' OR 1=1; --
    //                 ssns: SSNs root ' UNION SELECT Ssn, Sname FROM Student; --

    $sql = "SELECT * FROM users WHERE username = '" . $username . "' AND password = '" . $password . "';";
    //$sql = "SELECT * FROM Student";  // For example, retrieve all students from the 'Student' table
    $sth = $pdo->query($sql);
    $rows = $sth->fetchAll(PDO::FETCH_ASSOC); // Fetch as associative array
} catch (Exception $e) {
    // return a JSON error response
    echo json_encode(['success' => false, 'message' => 'Query execution failed: ' . $e->getMessage()]);
    exit;
}

// Return the query results as JSON
echo json_encode(['success' => true, 'data' => $rows]);
?>
```

Figure 2.10 Original php file before SQLi mitigation

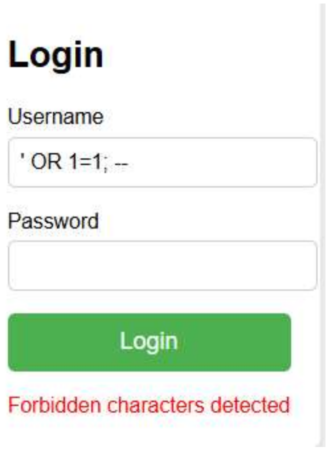
Mitigation:

To securely implement our program and mitigate SQL injection risks, we utilized parameterized queries. Parameterized queries ensure that user inputs are treated as data rather than executable SQL code, preventing malicious injection attempts. Instead of directly incorporating user input into the query string, we replaced it with placeholders and bound variables that are safely handled by the database engine and prepares the input.

Additionally, we implemented a function called suspicious to validate user input by scanning the provided username and password for potentially malicious characters commonly used in SQL injection attacks, such as **;** , **--** , and **'**. The function returned a flag variable to indicate the presence of suspicious characters.

In our HTML file, we incorporated logic to check this flag alongside the retrieved data. If suspicious characters were detected, the program displayed an injection error to the user, preventing further processing of the input.

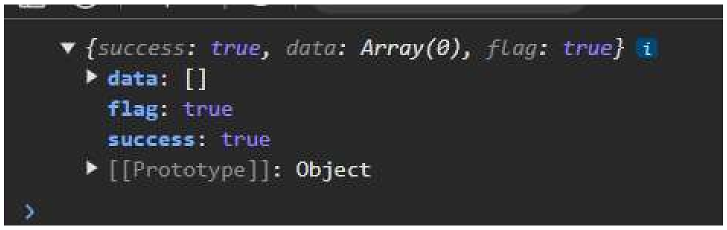
While our implementation focused on a specific set of suspicious characters, additional validation could be introduced to detect SQL keywords like **UNION, OR, AND**, and **DROP**. However, we intentionally left this out to keep our program lightweight and focused on core functionality.



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

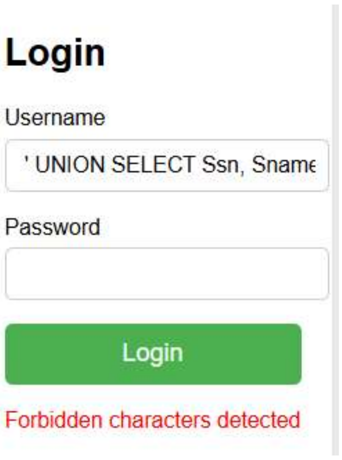
Forbidden characters detected

Figure 2.11 SQL injection fails, gets flagged



```
▼ {success: true, data: Array(0), flag: true}
  ▶ data: []
    flag: true
    success: true
  ▶ [[Prototype]]: Object
```

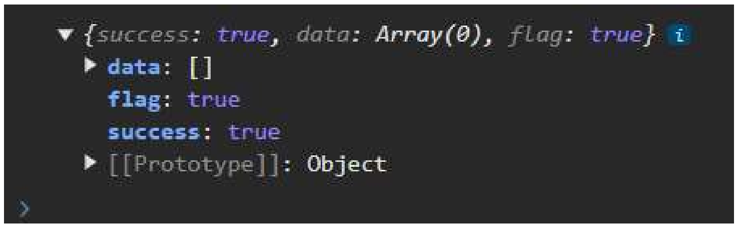
Figure 2.12 No data returned



```
' UNION SELECT Ssn, Sname
```

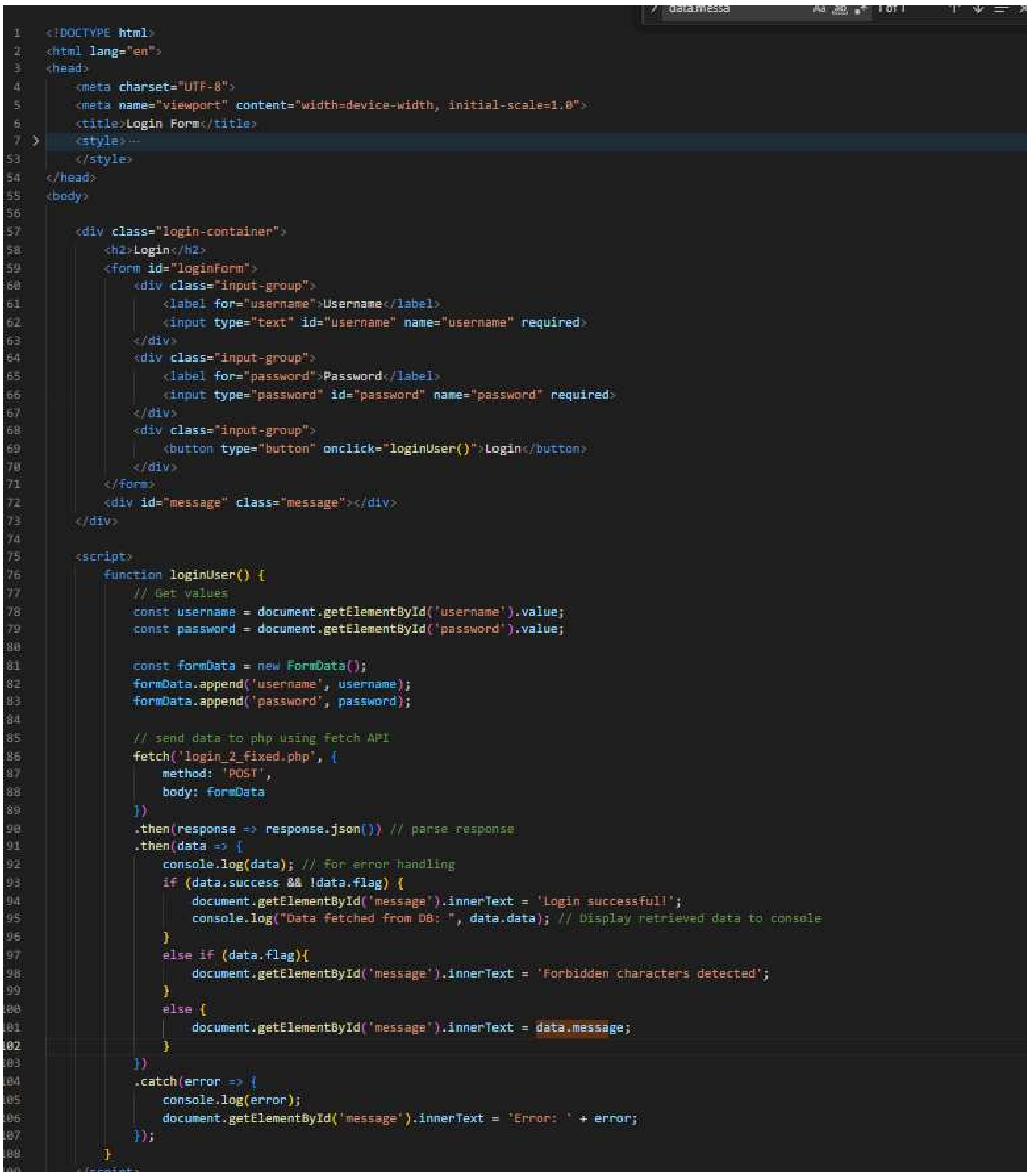
Forbidden characters detected

Figure 2.13 SQL injection fails, gets flagged



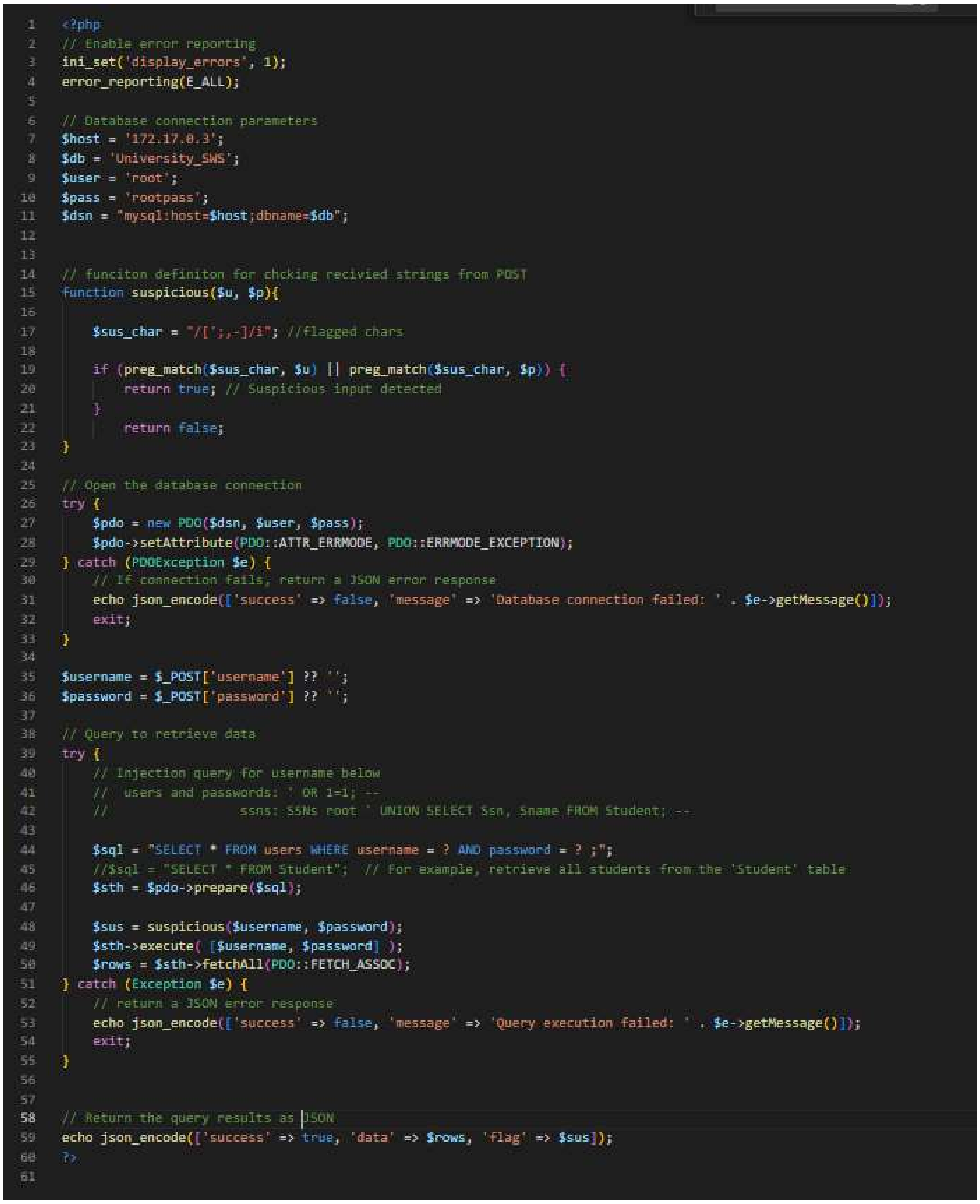
```
▼ {success: true, data: Array(0), flag: true}
  ▶ data: []
    flag: true
    success: true
  ▶ [[Prototype]]: Object
```

Figure 2.14 No data returned



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Form</title>
    <style>···
    </style>
</head>
<body>

    <div class="login-container">
        <h2>Login</h2>
        <form id="loginForm">
            <div class="input-group">
                <label for="username">Username</label>
                <input type="text" id="username" name="username" required>
            </div>
            <div class="input-group">
                <label for="password">Password</label>
                <input type="password" id="password" name="password" required>
            </div>
            <div class="input-group">
                <button type="button" onclick="loginUser()">Login</button>
            </div>
        </form>
        <div id="message" class="message"></div>
    </div>

    <script>
        function loginUser() {
            // Get values
            const username = document.getElementById('username').value;
            const password = document.getElementById('password').value;

            const formData = new FormData();
            formData.append('username', username);
            formData.append('password', password);

            // send data to php using fetch API
            fetch('login_2_fixed.php', {
                method: 'POST',
                body: formData
            })
            .then(response => response.json()) // parse response
            .then(data => {
                console.log(data); // for error handling
                if (data.success && !data.flag) {
                    document.getElementById('message').innerText = 'Login successful!';
                    console.log("Data fetched from DB: ", data.data); // Display retrieved data to console
                }
                else if (data.flag){
                    document.getElementById('message').innerText = 'Forbidden characters detected';
                }
                else {
                    document.getElementById('message').innerText = data.message;
                }
            })
            .catch(error => {
                console.log(error);
                document.getElementById('message').innerText = 'Error: ' + error;
            });
        }
```

Figure 2.15 Fixed HTML file preventing SQLi



```php
<?php
// Enable error reporting
ini_set('display_errors', 1);
error_reporting(E_ALL);

// Database connection parameters
$host = '172.17.0.3';
$db = 'University_SWS';
$user = 'root';
$pass = 'rootpass';
$dsn = "mysql:host=$host;dbname=$db";


// funciton definiton for chcking recivied strings from POST
function suspicious($u, $p){

    $sus_char = "/[';,-]/i"; //flagged chars

    if (preg_match($sus_char, $u) || preg_match($sus_char, $p)) {
        return true; // Suspicious input detected
    }
        return false;
}

// Open the database connection
try {
    $pdo = new PDO($dsn, $user, $pass);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
    // If connection fails, return a JSON error response
    echo json_encode(['success' => false, 'message' => 'Database connection failed: ' . $e->getMessage()]);
    exit;
}

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

// Query to retrieve data
try {
    // Injection query for username below
    //  users and passwords: ' OR 1=1; --
    //                 ssns: SSNs root ' UNION SELECT Ssn, Sname FROM Student; --

    $sql = "SELECT * FROM users WHERE username = ? AND password = ? ;";
    //$sql = "SELECT * FROM Student";  // For example, retrieve all students from the 'Student' table
    $sth = $pdo->prepare($sql);

    $sus = suspicious($username, $password);
    $sth->execute( [$username, $password] );
    $rows = $sth->fetchAll(PDO::FETCH_ASSOC);
} catch (Exception $e) {
    // return a JSON error response
    echo json_encode(['success' => false, 'message' => 'Query execution failed: ' . $e->getMessage()]);
    exit;
}


// Return the query results as JSON
echo json_encode(['success' => true, 'data' => $rows, 'flag' => $sus]);
?>
```

Figure 2.16 Fixed php file preventing SQLi

Conclusion

This project aimed to explore the critical vulnerability of SQL injection (SQLi), demonstrating its potential impact on web applications and databases through a controlled experiment on our own application. SQL injection exploits improper handling of user inputs to manipulate SQL queries, enabling attackers to bypass authentication, extract sensitive data, or escalate privileges. Using a simulated university database, simplified for abstraction, hosted on a secure AWS Linux server, we demonstrated two successful SQL attacks: retrieving all usernames and passwords stored on the database, and accessing confidential student information. These attacks highlight the damaging consequences of failing to sanitize user inputs effectively. To address these vulnerabilities, we implemented a combined mitigation technique, focusing on parameterized queries to ensure user inputs are treated as data rather than executable queries. Additionally we developed a validation function that detects potentially malicious characters that someone inputs. These measures together reduced the attack surface significantly and prevented the same injection commands from being used for unauthorized access.

References:

https://www.acunetix.com/blog/articles/exploiting-sql-injection-example/

https://www.w3schools.com/sql/sql_injection.asp

https://www.w3schools.com