**Laboratory Report: SQL Injection Attack and Mitigation**

**Course:** *IAS101*  
**Lecturer:** *Tomas Paolo A. Echaluce*  
**Date Conducted:** February 26, 2025  
**Student Name:** *Jerome A. Brenzuela*  
**Student ID:** *223166*

Introduction

SQL Injection (SQLi) is a type of attack where an attacker inserts harmful SQL commands into a web application's input fields. If the application does not properly check or clean user input, the attacker can gain unauthorized access, change database records, or even take full control of the system

Purpose of the lab: to manually exploit SQL Injection vulnerabilities and implement basic input sanitization as a fix.

Objectives

* Identify SQL Injection vulnerabilities in the given project.
* Manually craft and enter SQL Injection payloads to exploit the system.
* Fix the vulnerabilities without using prepared statements.
* Understand the importance of input sanitization.

Materials and Tools

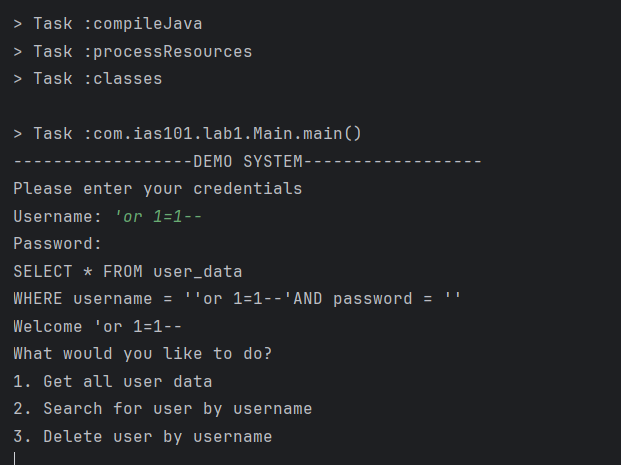
* **Application Used:** *IntelliJ IDEA*
* **Database:** SQLite
* **Software Used:** IDE (IntelliJ IDEA), SQLite, Java Language, Build Tool (Gradle)
* **Attack Method:** Manually entering SQL Injection payloads

Procedure

**A. Performing the SQL Injection Attack**

1. **Identifying Vulnerable Input Fields**
   * User input fields were tested, during testing I used Authentication Bypass which is the (‘ or 1=1--), and it bypass the username and password.
   * The feedback I observed was only the Bad credentials. As I input wrong username and password.
2. **During and after the attack**
   * The system respond well, and the Tasks are completely executed, there is no errors encountered.

Screenshots of Successful SQL injection:



**B. Fixing the Vulnerabilities**

1. **Code Review: Identifying Vulnerable Queries**
   * Show the original SQL query and explain why it is vulnerable. Example:
2. String query = "SELECT \* FROM users WHERE username = '" + userInput
3. + "' AND password = '" + passwordInput + "'";
   * + Vulnerability: User input is directly concatenated into the query. Which can lead to easily comment what is in the database.
4. **Applying Input Sanitization (Without Prepared Statements)**
   * InputSanitization: this will replace the inputted (‘, “, =, --)
   * Provide the updated code with comments. Example:
5. String safeUsername = InputSanitizer.*escapeInput*(username);  
   String safePassword = InputSanitizer.*escapeInput*(password);  
     
   String query = "SELECT \* FROM user\_data WHERE username = '" + safeUsername + "' AND password = '" + safePassword + "'";  
     
     
   System.*out*.println(query);  
   ResultSet rs = statement.executeQuery(query);  
     
   return rs.next();

Explain how this solution works and its limitations.

1. **Testing the Fix**

This method is not truly or fully secured. As it is only a temporary for rush.

**5. Results and Analysis**

The system can now prevent or secure from SQL injection.

**6. Conclusion**

It highlighted the dangers of SQL Injection and demonstrated how improper handling of user input can leave applications vulnerable to attacks. The key takeaway is that directly inserting user input into SQL queries without proper validation can lead to authentication bypass, data breaches, or even database destruction.

To prevent SQL Injection, it is crucial to use secure coding practices, such as prepared statements, input validation, and escaping special characters. However, attackers can still find ways to bypass basic sanitization, which is why a defense-in-depth approach is necessary. This includes limiting database privileges, implementing web application firewalls, and regularly testing for vulnerabilities.