EC7020: COMPUTER AND NETWORK SECURITY

LABORATORY EXPERIMENT: 02

SECURITY VULNERABILITIES

Reg No: 2020/E/013 11/10/2024, from 13:30 to 16:30

AIM: Student will learn the fundamental principles of computer and network security by studying attacks on computer systems, network, and the Web. Students will learn how those attacks work and how to prevent and detect them.

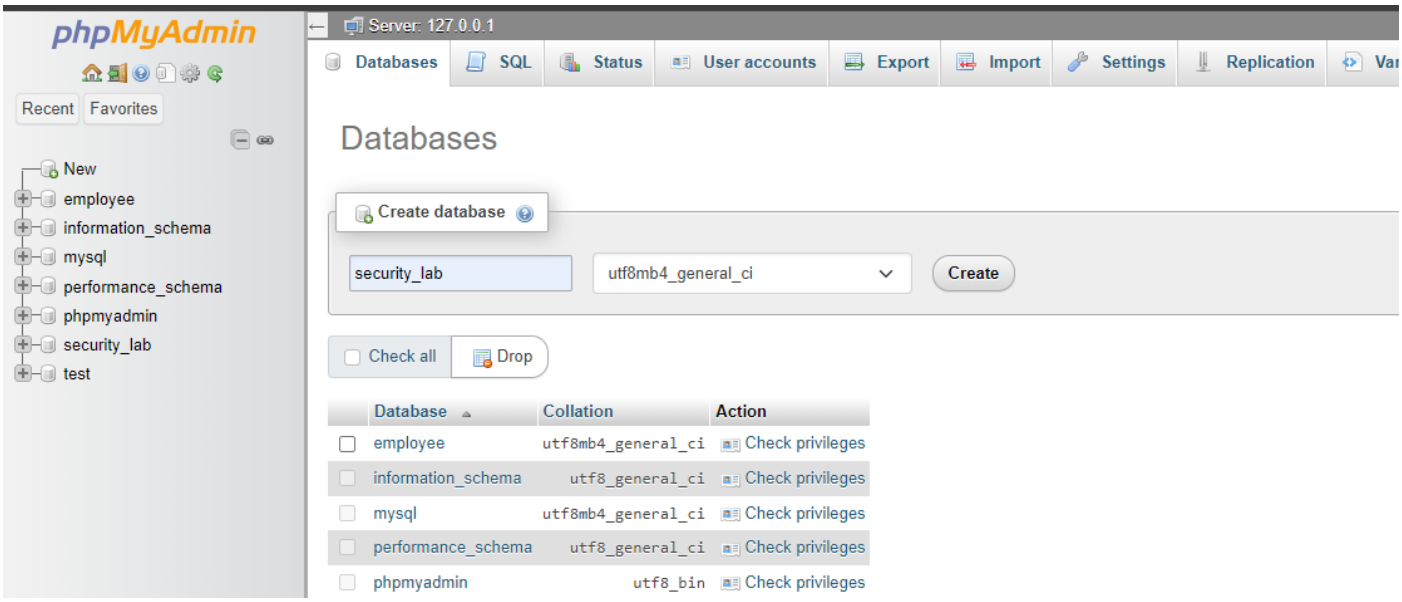
OBJECTIVES:

- To understand the fundamentals of Internet Security.
- To understand the Fundamentals about Internet Security Vulnerabilities.
- Ability to simulate basic Security vulnerabilities.

Following are the tasks for this lab session.

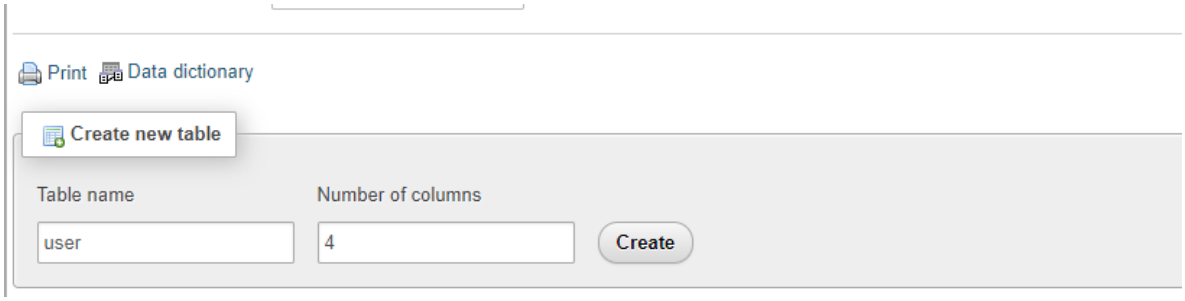
1. Implement SQL injection in your own. Preferred to use java and MySQL.

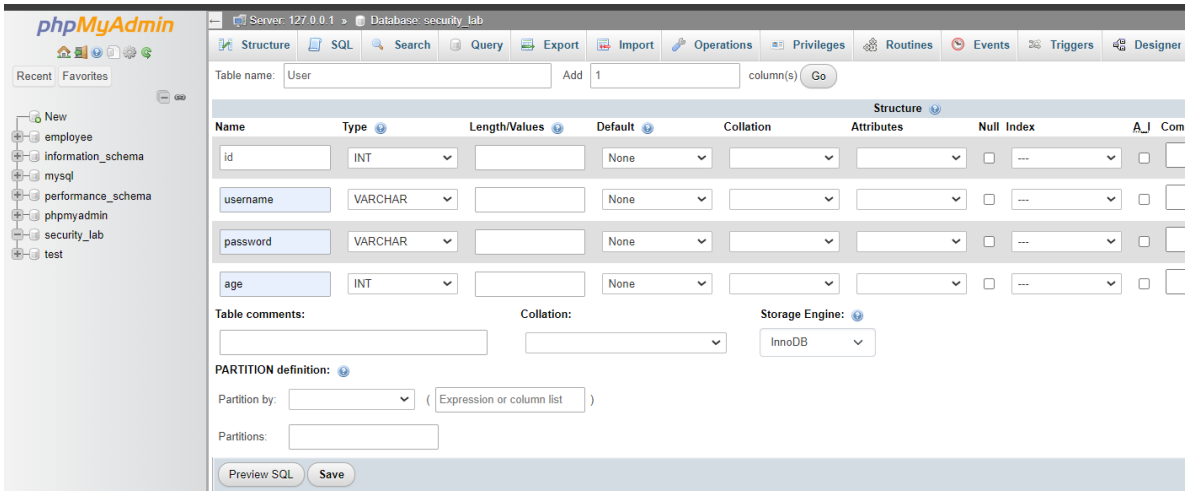
a. Create a SQL Database Schema (5 Marks)

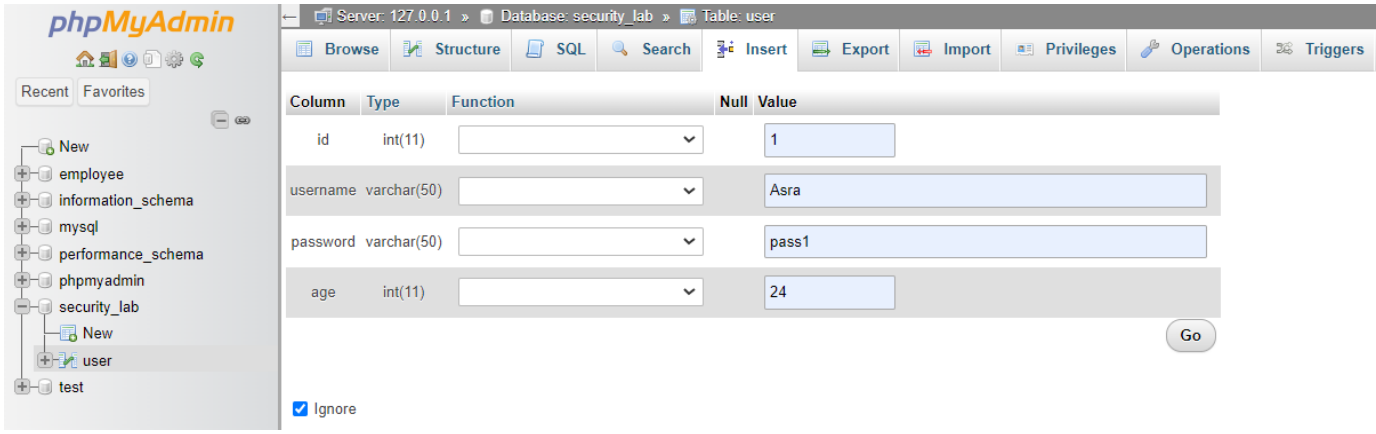


b. Create a Table named User Table and add some data. (5 Marks)

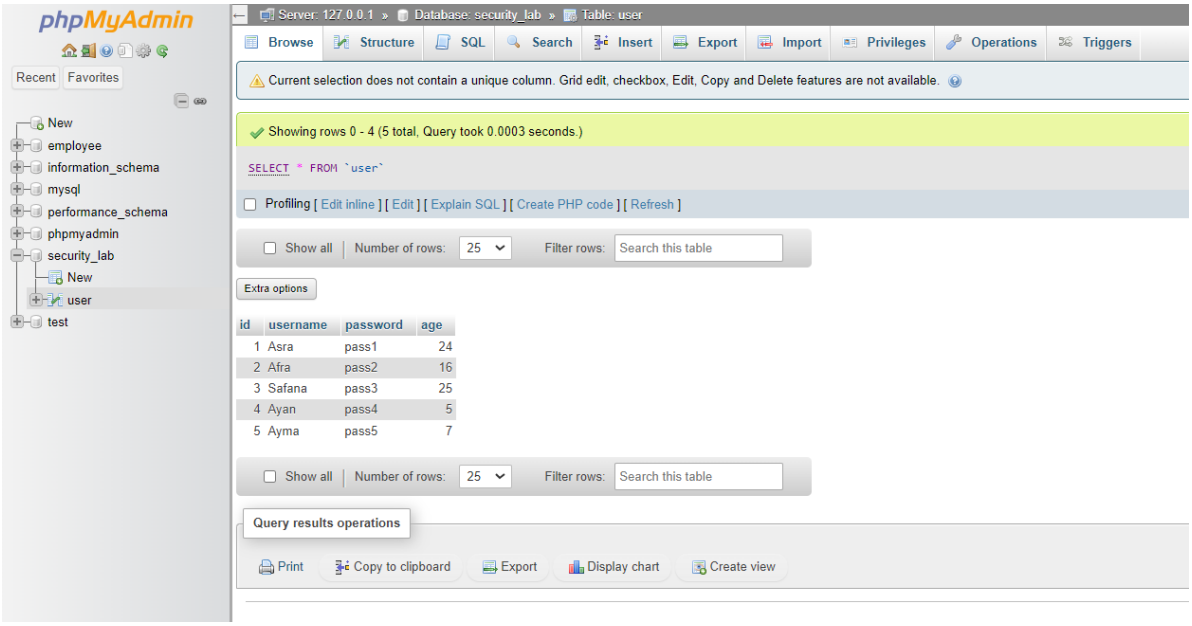
Creating a database user with a table name user. And create columns id, username, password, and age







Like this, I inserted 5 rows



c. Create an application to retrieve particular User data. (40 Marks)

Establishing a database connection is a fundamental step in developing applications that interact with a database.

Set up JDBC Connection:

Include MySQL JDBC Driver (mysql-connector-java.jar)

UserRetrieval.java

Move this file to a named package. SonarLint: Show rule description 'java:S1220' SonarLint: Disable rule 'java:S1220' SonarLint: Mark issue as...

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Scanner;

public class UserRetrieval {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost/security_lab";
        String user = "root"; // Default MySQL user
        String password = ""; // Default MySQL password, usually empty for XAMPP

        try {
            Connection conn = DriverManager.getConnection(url, user, password);
            Scanner scanner = new Scanner(System.in);

            System.out.print("Enter username to retrieve: ");
            String username = scanner.nextLine();

            String query = "SELECT * FROM User WHERE username = '" + username + "'"; // Vulnerable to SQL Injection
            PreparedStatement stmt = conn.prepareStatement(query);
            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                System.out.println("ID: " + rs.getInt( columnLabel: "id") + ", Username: "
                        + rs.getString( columnLabel: "username") + ", Password: " + rs.getString( columnLabel: "password"));
            }

            conn.close();
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C
Enter username to retrieve: Ayan
ID: 4, Username: Ayan, Password: pass4

Process finished with exit code 0
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Fil
Enter username to retrieve: Ayan' OR '1'='1
ID: 1, Username: Asra, Password: pass1
ID: 2, Username: Afra, Password: pass2
ID: 3, Username: Safana, Password: pass3
ID: 4, Username: Ayan, Password: pass4
ID: 5, Username: Ayma, Password: pass5

Process finished with exit code 0
```

d. Then initiate any two SQL Injection technique without modifying the source code.(10 Marks)

Blind sql injection

In blind sql injection, the attacker cannot directly view the result query, but they can infer information based on how the system behaves

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Progra
Enter username to retrieve: Ayan' AND '1'='1
ID: 4, Username: Ayan, Password: pass4

Process finished with exit code 0
```

This input would be sanitized by the preparedStatement, so it's safe.

Second-order SQL injection

This occurs when malicious data is injected during one part of the application (like data entry) but is later executed in another query.

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\
Enter username to retrieve: anything' OR '1'='1
ID: 1, Username: Asra, Password: pass1
ID: 2, Username: Afra, Password: pass2
ID: 3, Username: Safana, Password: pass3
ID: 4, Username: Ayan, Password: pass4
ID: 5, Username: Ayma, Password: pass5
```

e. Then modify the code to secure the application from SQL injection. (20 Marks)

UserRetrieval.java

Move this file to a named package. SonarLint: Show rule description 'java:S1220' SonarLint: Disable rule 'java:S1220' SonarLint:

```java
import ...

public class UserRetrieval {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost/security_lab";
        String user = "root"; // Default MySQL user
        String password = ""; // Default MySQL password, usually empty for XAMPP

        try {
            Connection conn = DriverManager.getConnection(url, user, password);
            Scanner scanner = new Scanner(System.in);

            System.out.print("Enter username to retrieve: ");
            String username = scanner.nextLine();
            // prevention of sql injection
            String query = "SELECT * FROM User WHERE username = ?";
            PreparedStatement stmt = conn.prepareStatement(query);
            stmt.setString( parameterIndex: 1, username);
            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                System.out.println("ID: " + rs.getInt( columnLabel: "id") + ", Username: "
                        + rs.getString( columnLabel: "username") + ", Password: " + rs.getString( columnLabel: "password"));
            }

            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaag
Enter username to retrieve: Ayan' OR '1'='1

Process finished with exit code 0
```

Discussion (about the problem) (10 Marks)

In this lab session, we explored SQL injection, a critical security vulnerability when attackers manipulate user inputs to execute unintended SQL commands on the database. By simulating SQL injection attacks in a Java-based application connected to a MySQL database, we observed how improperly handled user input can lead to unauthorized data access. We demonstrated two SQL injection techniques: bypassing authentication

and using comments to manipulate SQL statements. These techniques exploited the application's vulnerability by injecting malicious inputs to modify the SQL query's logic. This lab emphasized the significance of input validation and query parameterization, as the original code directly embedded user inputs into the SQL query, making it susceptible to attack. By understanding these vulnerabilities, developers can recognize the dangers posed by SQL injection, such as data breaches, unauthorized access, and potential loss of database integrity. The exercise highlighted the need for secure coding practices and reinforced the importance of safeguarding applications by employing defenses like prepared statements, input validation, and escaping special characters. This experience underscored that security should be an integral part of the software development lifecycle to protect systems from malicious exploitation effectively.

Conclusion (Conclude this laboratory Session) (10 Marks)

The lab session on SQL injection provided valuable insights into the fundamentals of web application security and the techniques to safeguard against common attacks. Through practical implementation and experimentation, we identified the vulnerabilities associated with dynamic SQL query construction, which allowed malicious inputs to manipulate the application's behavior. By initially demonstrating how attackers could exploit these vulnerabilities to bypass authentication or access confidential data, we showcased the real-world risks associated with unsecured code. Afterward, we modified the code to implement prepared statements, effectively mitigating the risk of SQL injection by ensuring that user inputs were safely handled. This exercise underscored the importance of secure coding practices, such as using parameterized queries and input validation, to protect applications from SQL injection attacks. It also illustrated the need for developers to continuously be aware of potential security flaws and incorporate defensive programming techniques into their workflows. In conclusion, the lab reinforced the notion that proactive security measures are essential in developing robust software, and that even seemingly minor coding mistakes can lead to significant security breaches if left unaddressed

This is individual work. Write your answers in this Lab Instruction sheet with the file name EC7020_L1_YourRegNo. Submit it as a pdf document and archive all files and upload to the teams. Same name conversion applies for the Zip.