

# WAPH-Web Application Programming and Hacking

**Instructor:** Dr. Phu Phung

**Student**

**Name:** Amit Gaddi

**Email:** gaddiat@mail.uc.edu

**Short-bio:** Amit has keen interests in IT.



## Repository Information

Repository's URL: <https://github.com/gaddiat-uc/waph.git>

This is a private repository for Amit Gaddi to store all code from the course. The organization of this repository is as follows.

## Lab 3 - Secure Web Application Development in PHP/MySQL

Lab3 Link

### Overview

I became fully involved in the practical elements of creating, administering, and protecting online applications using PHP and MySQL in Lab 3 of the Secure online Application Development course. The lab is set up to walk me through the process of creating a basic login system that is purposefully weak to expose typical online vulnerabilities like SQL Injection and Cross-Site Scripting (XSS) attacks. Students first learn the fundamentals of database construction and maintenance before moving on to create an unsafe login system, which I then use in practical hacking activities. At the end of the lab, prepared statements for SQL injection prevention and output sanitization to reduce XSS threats are implemented, strengthening the application's security. By taking such a thorough approach, the lab not only made me aware of the challenges that online apps face, but it also gave me the basic security knowledge I needed to address these flaws and create a strong basis for developing safe web applications.

### a. Database Setup and Management (10 pts)

**Create a New Database, Database User and Permission** In the file `database-account.sql`, I have written the code to create a new database named `waph` & user with my school's 6+2, moreover I had granted all permissions to the user. To run this command in terminal I used the command `'sudo mysql -u gaddiat -p < database-account.sql'`

Included file `database-account.sql`

```
create database waph;  
CREATE USER 'gaddiat'@'localhost' IDENTIFIED BY '1234';  
GRANT ALL ON waph.* TO 'gaddiat'@'localhost';
```

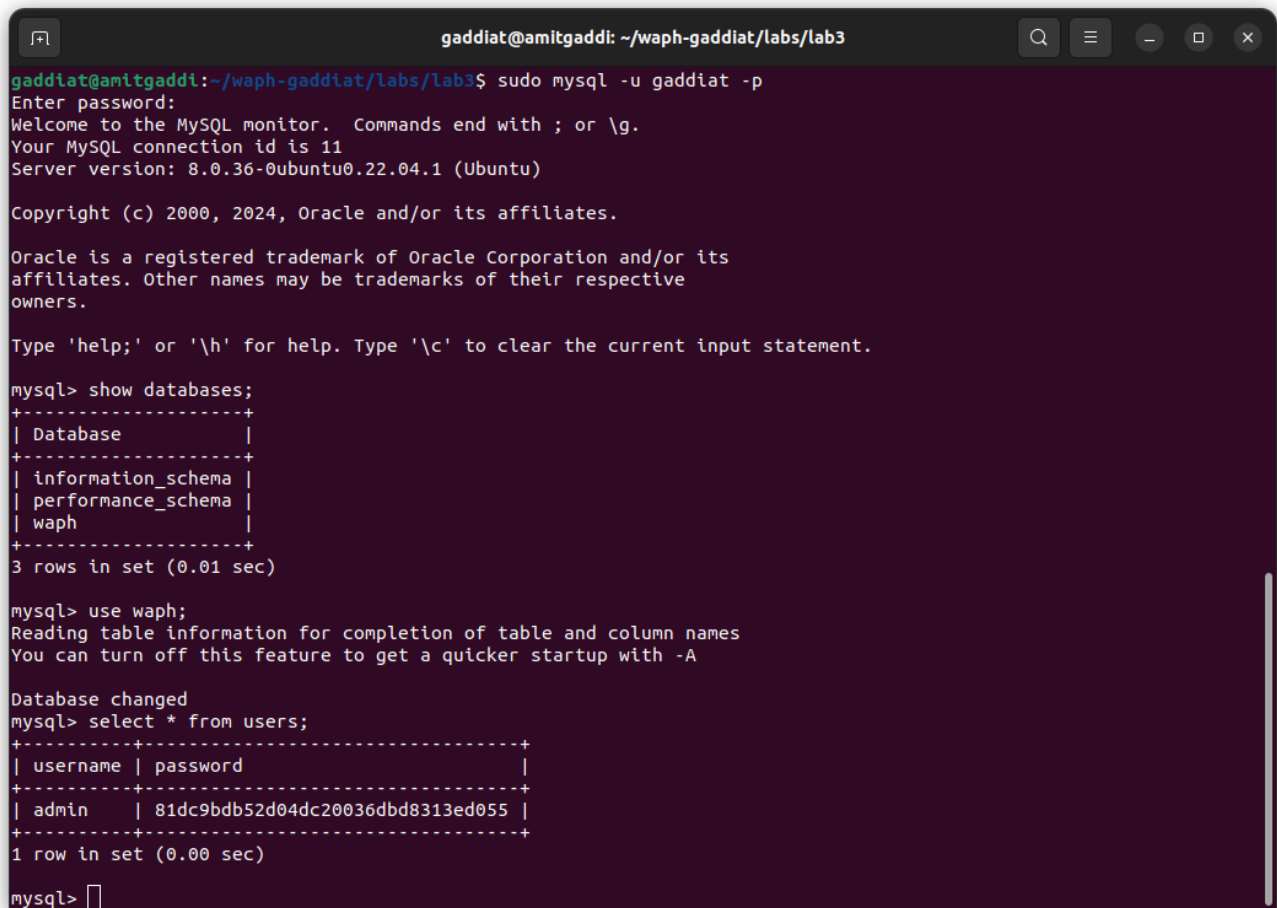
Then I logged in with the above user into the db.

**Create a new table Users and insert data into the table** I wrote the below code in database-data.sql and ran the file with the non root user in waph database, this created a new table users and stored the username admin and hashed password in the table.

Included file database-data.sql

```
create table users(  
    username varchar(50) PRIMARY KEY,  
    password varchar(100) NOT NULL);  
INSERT INTO users(username,password) VALUES ('admin',md5('1234'));
```

Below is the screenshot of the table and the user data.



```
gaddiat@amitgaddi: ~/waph-gaddiat/labs/lab3  
gaddiat@amitgaddi:~/waph-gaddiat/labs/lab3$ sudo mysql -u gaddiat -p  
Enter password:  
Welcome to the MySQL monitor.  Commands end with ; or \g.  
Your MySQL connection id is 11  
Server version: 8.0.36-0ubuntu0.22.04.1 (Ubuntu)  
  
Copyright (c) 2000, 2024, Oracle and/or its affiliates.  
  
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
  
mysql> show databases;  
+-----+  
| Database |  
+-----+  
| information_schema |  
| performance_schema |  
| waph |  
+-----+  
3 rows in set (0.01 sec)  
  
mysql> use waph;  
Reading table information for completion of table and column names  
You can turn off this feature to get a quicker startup with -A  
  
Database changed  
mysql> select * from users;  
+-----+  
| username | password |  
+-----+  
| admin | 81dc9bdb52d04dc20036dbd8313ed055 |  
+-----+  
1 row in set (0.00 sec)  
  
mysql> 
```

## b. A Simple (Insecure) Login System with PHP/MySQL

For my page I have used the below login system that checks the password and username that we can created in the above database, we enter the details in form.php page and from the index.php it checks the matches the details and displays the welcome message if the details are valid.

Included snippet of index.php

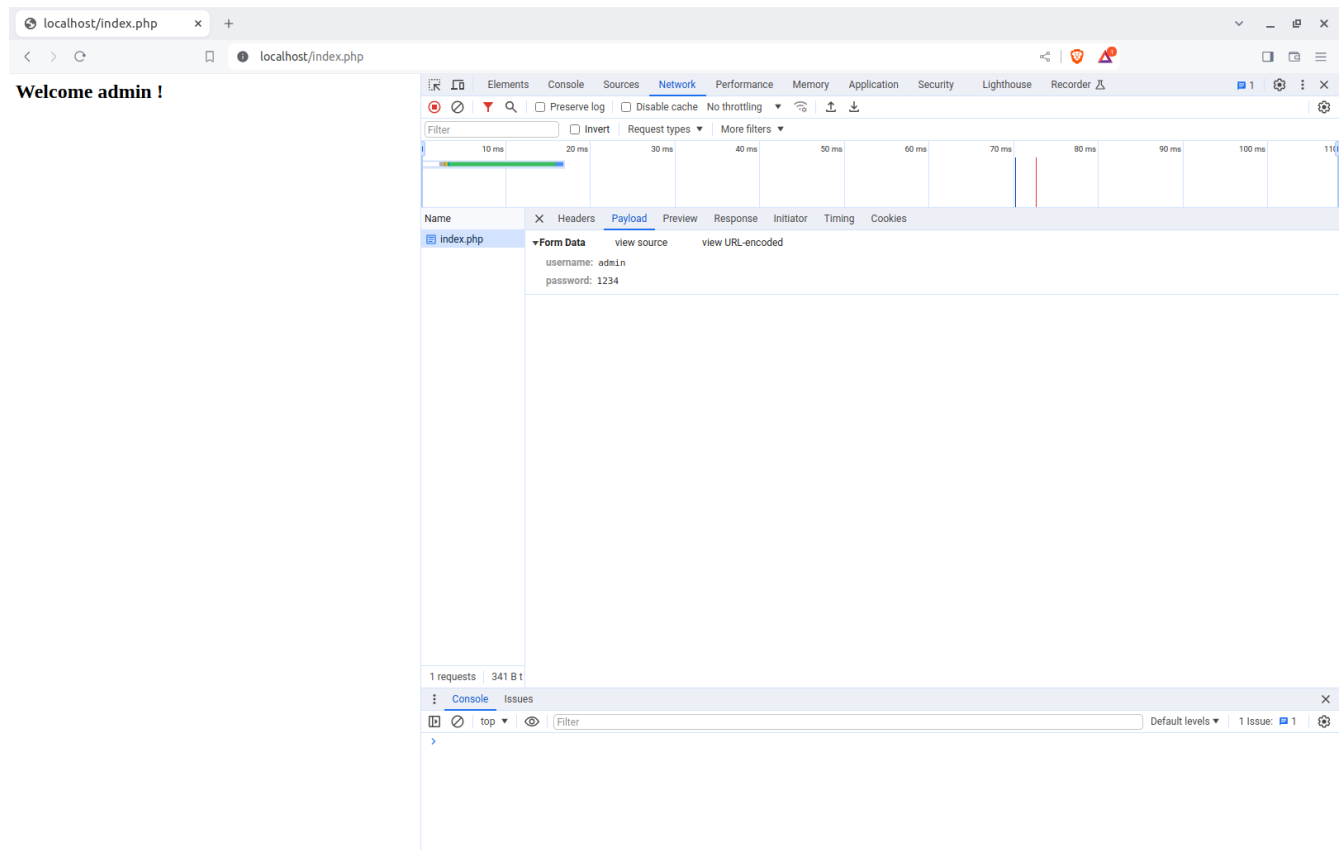
```
function checklogin_mysql($username, $password) {  
    $mysqli = new mysqli('localhost','gaddiat','1234','waph');  
    if($mysqli->connect_errno){  
        printf("Database connection failed: %s\n", $mysqli->connect_errno);  
        exit();  
    }  
    $sql = "SELECT * FROM users WHERE username='" . $username . "'";
```

```

$sql = $sql . "AND password =md5('" . $password . "')";
$result = $mysqli->query($sql);
if($result->num_rows ==1)
    return TRUE;
return FALSE;
}

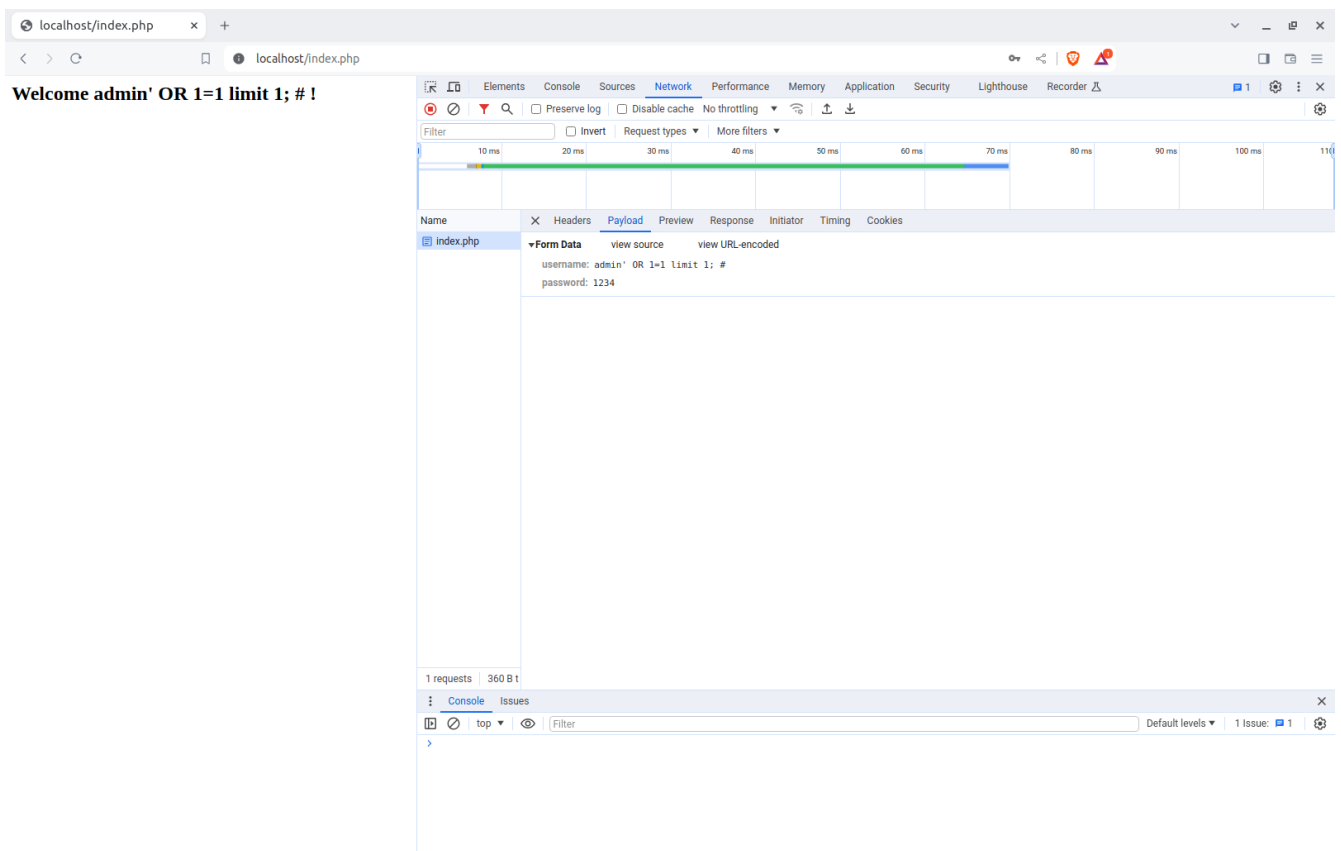
```

Below is screenshot of successful login.

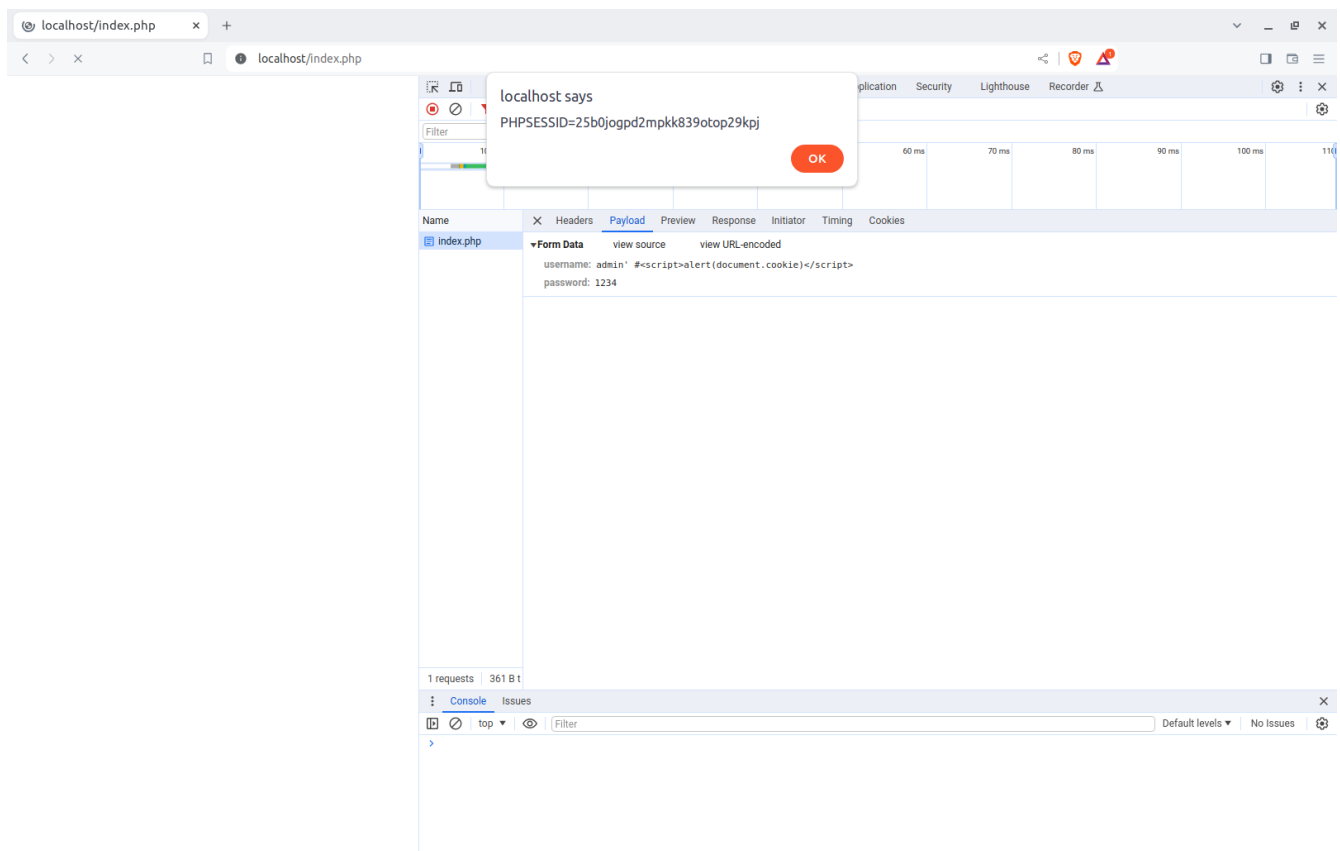


### c. Performing XSS and SQL Injection Attacks

**SQL Injection Attacks** Below is the screenshot of the attack where sql query was ran and displayed in the browser by the attacker, due to the failure of validating the code and sanitizing and verifying the details of the user input before the sql query is ran.



**Cross-site Scripting (XSS)** Below is the screenshot of the attack due to the failure of sanitizing the input which javascript code that displays the cookie details, here the attacker used the script tag and then passed it as an input to get the details, here the attackers insert malicious code into the system such as the above mentioned. this occurs when the input received is not cleaned.



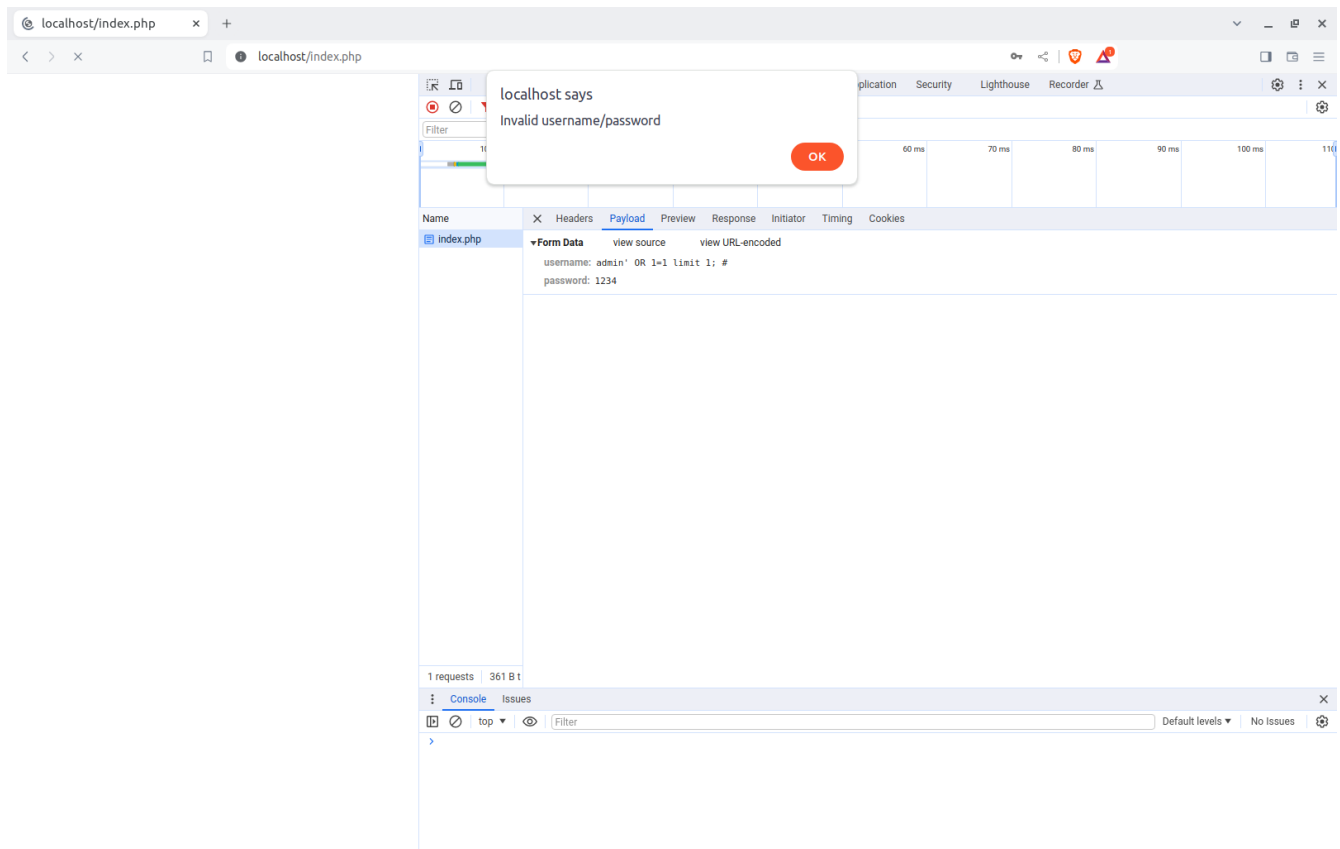
#### d. Prepared Statement Implementation

**Prepared Statement for SQL Injection Prevention** SQL Injection prevention can be done as i did for my page, here the code is not directly taken but rather it implements prepared statements to form the query, below is the code snippet of the function that prevents the sql injection.

Included sinppet index.php

```
function checklogin_mysql($username, $password) {
    $mysqli = new mysqli('localhost', 'gaddiat', '1234', 'waph');
    if($mysqli->connect_errno){
        printf("Database connection failed: %s\n", $mysqli->connect_errno);
        exit();
    }
    $sql = "SELECT * FROM users WHERE username=? AND password = md5(?)";
    $stmt = $mysqli->prepare($sql);
    $stmt->bind_param("ss", $username, $password);
    $stmt->execute();
    $result = $stmt->get_result();
    if($result->num_rows ==1)
        return TRUE;
    return FALSE;
}
```

Below is the screenshot of the failed SQL Injection attack.



## Security Analysis

- **Prepared Statement Explanation:**

Instead of immediately incorporating user inputs into the query string, developers may create SQL queries with placeholders for parameters by utilizing prepared statements. By using this method, you can be guaranteed that user inputs are handled as data and not like components of the SQL operation. This means that parameters must be tied to these placeholders in a different stage. This strategy is essential for preventing injection attacks and preserving the integrity of the SQL query.

When parameters are tied to placeholders in prepared statements, the underlying database driver handles the automated escaping of special characters in the incoming data. In order to stop harmful inputs from being interpreted as part of the SQL syntax, this feature is essential. This successfully neutralizes SQL injection attacks, which take advantage of security holes by introducing malicious SQL into a database query. This security precaution is essential for maintaining the integrity of the database and shielding private data from illegal access.

Moreover, the database server precompiles prepared statements, generating a query execution plan that is not yet dependent on parameter values. By separating the data from the SQL query's structure, this procedure not only increases security but also boosts efficiency. Database operations are sped up because reusing the execution plan for many queries with distinct parameters eliminates the need to recompile the query. Prepared statements are an essential tool in the creation of safe and effective database-driven systems because of their double advantages.

- **Implement Sanitization:**

Without any kind of sanitization, the username was echoed back to the user in the original code, which might have led to XSS vulnerabilities, particularly if the username contained HTML or JavaScript code. In order to mitigate this danger, special characters in the username are encoded using the `htmlspecialchars()` method before it is echoed back. Characters such as `{`, `>`, `&`, `"`, and `'` are transformed into their respective HTML entities using this method. By doing this, it protects against malicious code execution and thwarts XSS attacks by guaranteeing that these characters appear in the browser as plain text.

Below is revised code that sanitizes the outputs and mitigating the XSS risks.

```

<?php
    session_start();
    if (checklogin_mysql($_POST["username"],$_POST["password"])) {
?>
        <h2> Welcome <?php echo htmlentities($_POST['username']); ?> !</h2>
<?php
    }else{
        echo "<script>alert('Invalid username/password');window.location='form.php';</script>";
        die();
    }
    function checklogin_mysql($username, $password) {
        $mysqli = new mysqli('localhost','gaddiat','1234','waph');
        if($mysqli->connect_errno){
            printf("Database connection failed: %s\n", $mysqli->connect_errno);
            exit();
        }
        $sql = "SELECT * FROM users WHERE username=? AND password = md5(?)";
        $stmt = $mysqli->prepare($sql);
        $stmt->bind_param("ss", $username, $password);
        $stmt->execute();
        $result = $stmt->get_result();
        if($result->num_rows ==1)
            return TRUE;
        return FALSE;
    }
?>

```

- **Discussions:**

Indeed, there are a number of possible vulnerabilities and programming errors in the current code that need to be fixed to improve security and robustness:

- There is no validation in place for inputs that contain null usernames or passwords. In the absence of this check, the authentication procedure could continue with insufficient data, which could result in unexpected behavior or mistakes. Verifying user inputs to make sure they are not empty is essential before moving forward with authentication.
- The `checklogin_mysql()` function as it is currently implemented requires that the password and username entered exactly match those in the database. But if the username is entered differently—for example, with extra spaces or different case sensitivity—authentication will fail. Using case-insensitive or fuzzy matching for usernames, or sending out notifications when an entered username nearly matches one in the database, could enhance user experience and flexibility.
- There are no defenses against brute force attacks in the code, including rate limitation or account lockout. Implementing such measures can improve overall security posture and lessen the chance of unwanted entry attempts.