AJEENKYA DY PATIL UNIVERSITY

School of Engineering

TOPIC - PHP Secure Login Code and HTML Login Form

Name - Yash Thakkar

Subject- ISA Report

URN – 2022-B-14112004

Branch - B.Tech MAIS

Name – Saad Ansari

Subject- ISA Report

URN – 2022-B-30052005

Branch - B.Tech MAIS

1 PHP Secure Login Code:

```
<?php
// Database connection
$host = 'localhost';
$dbname = 'secure_login'; // Replace with your database name
$username = 'root'; // Default username for XAMPP
$password = ''; // Default password for XAMPP

try {
    $pdo = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
    die("Database connection failed: " . $e->getMessage());
}

// Handle login request
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $user = $_POST['username'];
    $pass = $_POST['password'];

    // Prevent SQL Injection using prepared statements
    $stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username AND password =
:password");
    $stmt->bindParam(':username', $user);
    $stmt->bindParam(':password', $pass); // Using plain-text password
    $stmt->execute();

    // Check if a user record is found
    if ($stmt->rowCount() > 0) {
```

```
        echo "Login successful!";
    } else {
        echo "Invalid username or password.";
    }
}
?>
```

Report on PHP Secure Login Code:

Overview:

The provided PHP code implements a basic login system with a MySQL database connection. It demonstrates how to authenticate a user by checking the provided username and password against records stored in a database. The code uses **PDO** (PHP Data Objects) for database interaction and prepares the SQL query to mitigate SQL injection risks.

Code Breakdown

1. Database Connection

- The connection is established using the PDO class.
- Database connection parameters are specified, including the host (localhost), database name (secure_login), and credentials (username = root, password = '').
- The connection is enclosed within a try-catch block to handle any potential errors. If the connection fails, a PDOException is thrown, and an error message is displayed to the user.

2. Login Request Handling

- The script checks if the request method is POST, indicating that the user has submitted a login form.
- The user's input is captured using $_POST for both the username and password

3. SQL Query and Prepared Statements

- A **prepared statement** is used to execute the SQL query. Prepared statements ensure that user inputs are safely handled and help prevent SQL injection attacks.
- The code binds the input parameters (username and password) to the prepared statement using bindParam.

4. **Executing the Query**

- The statement is executed using $stmt->execute(), which fetches the matching user from the database.

5. **Login Success/Failure**

- After executing the query, the script checks whether the query returned any rows using $stmt->rowCount().
- If at least one row is returned (i.e., a matching user is found), the user is authenticated, and a success message ("Login successful!") is displayed.
- If no records are found, the user is notified with an error message ("Invalid username or password.").

Security Issues and Recommendations:

1. **Plain-text Password Storage**:
    - The biggest security flaw in this code is storing passwords in plain text in the database. It's critical to hash passwords using a secure hashing algorithm such as bcrypt (provided by PHP's password_hash() function).
    - During login, the password should be verified using password_verify().

Suggested Fix:

- Hash the password before storing it in the database:

php

Copy code

```
$hashed_password = password_hash($pass, PASSWORD_BCRYPT);
```

- In the login check, use password_verify() to check the password:

php

Copy code

```
if ($stmt->rowCount() > 0) {
   $userRecord = $stmt->fetch(PDO::FETCH_ASSOC);
   if (password_verify($pass, $userRecord['password'])) {
      echo "Login successful!";
   } else {
      echo "Invalid username or password.";
```

```
    }
}
```

2. **Lack of Session Management**:
    - After successful authentication, the user is not provided with a session or authentication token. In a real-world application, sessions should be created using PHP's session_start() and $_SESSION[] to manage user login status.

Suggested Improvement:

php

Copy code

```
session_start();
$_SESSION['user_id'] = $userRecord['id'];
```

3. **Error Handling**:
    - The script currently outputs error messages directly, which could expose sensitive information. It's better to show a generic message (e.g., "Invalid login credentials") to the user and log detailed errors on the server side for debugging purposes.

Suggested Fix:

- Use a logging system or custom error handling to capture errors while providing generic messages to the user.

4. **SQL Injection Mitigation**:
    - While prepared statements are used, which significantly reduce the risk of SQL injection, it's essential to remember that this technique does not mitigate other security issues (such as weak passwords or session hijacking).

Conclusion:

The provided code demonstrates the use of PDO to securely query the database and authenticate users. However, the implementation is insecure because it stores passwords as plain text. To improve the security of the application, the passwords should be hashed, and proper session management should be implemented. Additionally, error handling should be refined, and user input should be validated and sanitized where applicable.

2 HTML Login Form code :

```
<!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>
  <form action="login.php" method="POST">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" required><br><br>

    <label for="password">Password:</label>
    <input type="password" id="password" name="password" required><br><br>

    <button type="submit">Login</button>
  </form>
</body>
</html>
```

Report on HTML Login Form:

Overview:

The provided HTML code creates a simple login form that collects a username and password from a user. The form submits the entered data to a server-side PHP script (login.php) using the POST method for authentication.

Code Breakdown

1. **HTML Structure**
   - The document begins with the standard HTML5 doctype declaration (<!DOCTYPE html>), followed by the opening <html> tag with the language set to English (lang="en").

- The <head> section contains meta tags for character encoding (UTF-8) and viewport settings to ensure the page is responsive on mobile devices.
- The page title is set to "Login".

2. **Form Structure**

- The form is defined using the <form> tag, with the action="login.php" attribute specifying the target PHP script for handling form submission. The method="POST" attribute ensures that the data is sent via the POST HTTP method, keeping it hidden from the URL.
- Inside the form, two input fields are defined: one for the username and one for the password.

3. **User Interface and Accessibility**
   - The form uses <label> tags to describe each input field, making it accessible for screen readers. The for attribute links each label to its respective input field, improving usability for users with disabilities.
   - The required attribute on both input fields ensures that the form cannot be submitted without entering values for both the username and password, providing a basic level of validation on the client side.
4. **Form Submission**
   - Upon clicking the "Login" button, the form submits the data (username and password) to login.php via the POST method.
   - The server-side PHP script (login.php) is responsible for processing the form data, validating the user, and managing login functionality (as discussed in the earlier PHP code).

Security Considerations:

While the HTML form itself does not directly introduce security vulnerabilities, it is important to note that sensitive data, such as passwords, should always be handled securely on the server side. In particular:

- Ensure that the password is **hashed** before being stored in the database (using functions like password_hash() in PHP).
- Consider using **HTTPS** for the form submission to encrypt the data in transit, protecting it from man-in-the-middle attacks.
- Implement input validation and sanitization on both the client and server sides to protect against injection attacks.

Potential Improvements

1. **CSS Styling**:
   - The form lacks any CSS styling, which may affect the user experience. Adding some CSS would help improve the appearance and layout of the form, making it more user-friendly.

2. **Client-side Validation**:
    - Although the required attribute ensures that both fields are filled, additional client-side validation (e.g., checking the format of the username or adding strength requirements for the password) could enhance user experience.
3. **Error Handling**:
    - Consider displaying error messages to the user if the login attempt fails. This would improve the user experience and guide them to correct any mistakes.
4. **Security Features**:
    - For enhanced security, add a CAPTCHA or other bot-detection mechanism to prevent automated login attempts.
    - Use https:// for the form submission URL to ensure encrypted communication between the client and server.

Conclusion:

This HTML form provides a basic user interface for logging into a system by submitting a username and password. While functional, it can be improved in terms of design, client-side validation, and security. The data submission to the server-side PHP script (login.php) should ensure proper password hashing, HTTPS usage, and protection against common web vulnerabilities like SQL injection and cross-site scripting (XSS).