**Example: Broken Authentication (Weak Password Storage & Session Mismanagement)**

**Assignment requirements:**

1. Source code cmplementation:
   * Develop a website containing a Broken Access Control vulnerability (as followed).
   * Implement a fix to resolve the vulnerability by enhancing the source code to ensure secure access control.
2. Write a report:
   * Document all the steps in detail, from building the website, identifying the vulnerability, to implementing the fix.
   * The report should clearly explain the process, how the vulnerability was detected, and the solution you applied.
3. Submit the report:
   * Complete the report file and submit it via the Google Form provided by the instructor.

Note: Ensure that all steps for the vulnerability detection and the fix are thoroughly detailed and accurate in the report.

**Code example**

# File Structure

/broken-auth-demo

├── index.php

├── login.php

├── register.php

├── home.php

└── logout.php

# index.php (Home Page - Login Form)

<?php

session\_start();

if (isset($\_SESSION['username'])) {

header('Location: home.php');

}

?>

<!DOCTYPE html>

<html>

<head>

<title>Login</title>

</head>

<body>

<h1>Login</h1>

<form action="login.php" method="POST">

<input type="text" name="username" placeholder="Username" required><br><br>

<input type="password" name="password" placeholder="Password" required><br><br>

<input type="submit" value="Login">

</form>

<p>Don't have an account? <a href="register.php">Register here</a></p>

</body>

</html>

# login.php (Login Handling)

<?php

session\_start();

// Database connection

$conn = new mysqli('localhost', 'root', '', 'broken\_auth\_demo');

// Check for errors

if ($conn->connect\_error) {

die("Connection failed: " . $conn->connect\_error);

}

if ($\_SERVER['REQUEST\_METHOD'] == 'POST') {

$username = $\_POST['username'];

$password = $\_POST['password'];

// Weak password check (plain text password comparison)

$sql = "SELECT \* FROM users WHERE username = '$username' AND password = '$password'";

$result = $conn->query($sql);

if ($result->num\_rows > 0) {

$\_SESSION['username'] = $username;

header('Location: home.php');

} else {

echo "Invalid username or password";

}

}

$conn->close();

?>

# register.php (Registration Handling)

<?php

session\_start();

// Database connection

$conn = new mysqli('localhost', 'root', '', 'broken\_auth\_demo');

// Check for errors

if ($conn->connect\_error) {

die("Connection failed: " . $conn->connect\_error);

}

if ($\_SERVER['REQUEST\_METHOD'] == 'POST') {

$username = $\_POST['username'];

$password = $\_POST['password'];

// Vulnerable to SQL Injection and stores password in plain text

$sql = "INSERT INTO users (username, password) VALUES ('$username', '$password')";

if ($conn->query($sql) === TRUE) {

echo "Registration successful!";

header('Location: index.php');

} else {

echo "Error: " . $sql . "<br>" . $conn->error;

}

}

$conn->close();

?>

<!DOCTYPE html>

<html>

<head>

<title>Register</title>

</head>

<body>

<h1>Register</h1>

<form action="register.php" method="POST">

<input type="text" name="username" placeholder="Username" required><br><br>

<input type="password" name="password" placeholder="Password" required><br><br>

<input type="submit" value="Register">

</form>

<p>Already have an account? <a href="index.php">Login here</a></p>

</body>

</html>

# home.php (Protected Page)

<?php

session\_start();

if (!isset($\_SESSION['username'])) {

header('Location: index.php');

exit();

}

?>

<!DOCTYPE html>

<html>

<head>

<title>Home</title>

</head>

<body>

<h1>Welcome, <?php echo $\_SESSION['username']; ?>!</h1>

<p>This is a protected page.</p>

<a href="logout.php">Logout</a>

</body>

</html>

# logout.php (Logout Handler)

<?php

session\_start();

session\_destroy();

header('Location: index.php');

exit();

?>

# Database Structure

You can create a simple MySQL table for users with the following SQL command:

CREATE DATABASE broken\_auth\_demo;

USE broken\_auth\_demo;

CREATE TABLE users (

id INT(11) AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(50) NOT NULL,

password VARCHAR(255) NOT NULL

);

# Vulnerabilities in This Example:

* **Plain-text Password Storage:** Passwords are stored directly in the database without hashing, making them easy to steal if the database is compromised.
* **SQL Injection:** The code directly inserts user input ($username and $password) into SQL queries without sanitization, making it vulnerable to SQL injection.
* **Session Mismanagement:** The session is based only on the username, with no additional security checks like tokens or session expiration.

1. **How to Improve Security:**

* Use password hashing with password\_hash() and password\_verify() in PHP.
* Prevent SQL injection by using prepared statements (e.g., mysqli\_prepare()).
* Implement proper session management with session expiration and secure cookies.
* Use HTTPS to protect communication between the client and server.

## Improved login.php (Secure Login)

Uses password hashing with password\_hash() and password\_verify().

Implements prepared statements to prevent SQL Injection.

<?php

session\_start();

// Database connection

$conn = new mysqli('localhost', 'root', '', 'secure\_auth\_demo');

// Check connection

if ($conn->connect\_error) {

die("Connection failed: " . $conn->connect\_error);

}

if ($\_SERVER['REQUEST\_METHOD'] == 'POST') {

$username = $\_POST['username'];

$password = $\_POST['password'];

// Use prepared statement to avoid SQL Injection

$stmt = $conn->prepare("SELECT password FROM users WHERE username = ?");

$stmt->bind\_param("s", $username);

$stmt->execute();

$stmt->store\_result();

// Check if the username exists

if ($stmt->num\_rows > 0) {

$stmt->bind\_result($hashed\_password);

$stmt->fetch();

// Verify the password using password\_verify()

if (password\_verify($password, $hashed\_password)) {

$\_SESSION['username'] = $username;

header('Location: home.php');

} else {

echo "Incorrect password.";

}

} else {

echo "Username not found.";

}

$stmt->close();

}

$conn->close();

?>

## Improved register.php (Secure Registration)

* Passwords are hashed using password\_hash() for secure storage.
* Uses prepared statements to prevent SQL Injection.

<?php

session\_start();

// Database connection

$conn = new mysqli('localhost', 'root', '', 'secure\_auth\_demo');

// Check connection

if ($conn->connect\_error) {

die("Connection failed: " . $conn->connect\_error);

}

if ($\_SERVER['REQUEST\_METHOD'] == 'POST') {

$username = $\_POST['username'];

$password = $\_POST['password'];

// Hash the password using password\_hash() before storing it

$hashed\_password = password\_hash($password, PASSWORD\_BCRYPT);

// Use prepared statement to avoid SQL Injection

$stmt = $conn->prepare("INSERT INTO users (username, password) VALUES (?, ?)");

$stmt->bind\_param("ss", $username, $hashed\_password);

if ($stmt->execute()) {

echo "Registration successful!";

header('Location: index.php');

} else {

echo "Error: " . $stmt->error;

}

$stmt->close();

}

$conn->close();

?>

<!DOCTYPE html>

<html>

<head>

<title>Register</title>

</head>

<body>

<h1>Register</h1>

<form action="register.php" method="POST">

<input type="text" name="username" placeholder="Username" required><br><br>

<input type="password" name="password" placeholder="Password" required><br><br>

<input type="submit" value="Register">

</form>

<p>Already have an account? <a href="index.php">Login here</a></p>

</body>

</html>

## Improved home.php (Protected Page)

* Added session timeout and secure session management.

<?php

session\_start();

// Check if the user is not logged in

if (!isset($\_SESSION['username'])) {

header('Location: index.php');

exit();

}

// Check session timeout (e.g., 30 minutes)

if (isset($\_SESSION['last\_activity']) && (time() - $\_SESSION['last\_activity'] > 1800)) {

session\_unset(); // Unset all session variables

session\_destroy(); // Destroy the session

header('Location: index.php');

exit();

}

$\_SESSION['last\_activity'] = time(); // Update last activity time

?>

<!DOCTYPE html>

<html>

<head>

<title>Home</title>

</head>

<body>

<h1>Welcome, <?php echo htmlspecialchars($\_SESSION['username']); ?>!</h1>

<p>This is a protected page.</p>

<a href="logout.php">Logout</a>

</body>

</html>

## Secure Database Structure

In this case, the database will store hashed passwords using password\_hash() instead of storing them in plain text.

CREATE DATABASE secure\_auth\_demo;

USE secure\_auth\_demo;

CREATE TABLE users (

id INT(11) AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(50) NOT NULL,

password VARCHAR(255) NOT NULL

);

# Security Improvements Implemented:

1. **Password Hashing:** Passwords are securely stored using password\_hash() and verified using password\_verify() to ensure they are not stored in plain text.
2. **SQL Injection Prevention:** Prepared statements (mysqli\_prepare() and bind\_param()) are used to safely handle user input and prevent SQL Injection.
3. **Session Management:**
   * Implemented session timeout (session expires after 30 minutes of inactivity).
   * session\_unset() and session\_destroy() are used to securely destroy sessions.
4. **Output Encoding:** htmlspecialchars() is used to prevent Cross-Site Scripting (XSS) attacks by encoding special characters in user input.

**How to test**

In the original insecure version of the website, the application does not properly sanitize or validate user inputs before directly inserting them into SQL queries. This opens the door to **SQL Injection**, a technique where an attacker manipulates the SQL query by injecting malicious SQL code into input fields, which the server mistakenly treats as valid instructions.

**How SQL Injection Works:**

In the original example, the login form in login.php directly concatenates user inputs into the SQL query, like this:

$sql = "SELECT \* FROM users WHERE username = '$username' AND password = '$password'";

Here, both $username and $password are directly passed into the query without any validation or protection. This means that an attacker can submit specially crafted SQL code in place of a normal username or password to manipulate the query.

* **Steps to Exploit SQL Injection Vulnerability:**
* **Step 1: Identify a Vulnerable Form**

In this case, the login form on the website (login.php) is vulnerable because it accepts user input directly in the query without sanitizing it.

* **Step 2: Submit Malicious Input**

Instead of providing a legitimate username and password, the attacker can submit SQL code. For example:

* **Username:** **' or 1=1-- !**
* **Password:** anything (can be left empty or irrelevant)

The SQL query now becomes:

SELECT \* FROM users WHERE username = '' OR '1' = '1' AND password = '';

The query essentially says:

* Select users where the username is an empty string ('') OR 1 = 1 (which is always true), and the password can be anything.

Since 1 = 1 is always true, this query returns the first user in the database, often the **administrator**, regardless of the password.

* **Step 3: Gain Unauthorized Access**

Once the query returns a user (usually the first row), the attacker is granted access as that user without knowing the actual password. They now have unauthorized access to the system, potentially as an admin or privileged user.

* **SQL Injection Exploitation Example in Detail:**

1. **Input:**
   * **Username:** **' or 1=1-- !**
   * **Password:** anything
2. **Query Generated:**

SELECT \* FROM users WHERE username = '' OR '1' = '1' AND password = 'anything';

**Result:**

* The condition 1 = 1 is true, so the query returns all rows where this condition holds.
* The first matching user (which could be the admin) is logged in without any need to match the correct password.