WELCOME TO THE WORLD OF WEB DEVELOPMENT

Week 7 – Day 3

Web Development Techniques using MVC architecture

Objective

- DAY 3
- - o SQL Injection and How to Prevent

SQL injection

SQL injection is a type of security vulnerability that occurs when an attacker manipulates a web application's input fields to inject malicious SQL code into a database query. The attacker exploits the application's failure to properly validate or sanitize user input, allowing them to execute unauthorized SQL queries.

SQL injection can have serious consequences, including unauthorized access to sensitive data, data corruption, or even complete database compromise.

SQL injection

To prevent SQL injection in PHP, follow these best practices:

Use Prepared Statements: Prepared statements are a powerful feature of PHP's PDO and MySQLi extensions that allow you to separate SQL code from user input. They automatically handle escaping and parameter binding, preventing SQL injection. Example using PDO:

```
$stmt = $pdo->prepare('SELECT * FROM users WHERE username = :username');
$stmt->execute(['username' => $username]);
```

SQL injection

To prevent SQL injection in PHP, follow these best practices:

Use Prepared Statements: Prepared statements are a powerful feature of PHP's PDO and MySQLi extensions that allow you to separate SQL code from user input. They automatically handle escaping and parameter binding, preventing SQL injection. Example using PDO:

```
$stmt = $pdo->prepare('SELECT * FROM users WHERE username = :username');
$stmt->execute(['username' => $username]);
```

SQL injection

Use Parameterized Queries: If prepared statements are not available, use parameterized queries with proper escaping functions (e.g., mysqli_real_escape_string()) to sanitize user input before using it in SQL queries. Example using MySQLi:

```
$username = mysqli_real_escape_string($conn, $username);
$sql = "SELECT * FROM users WHERE username = '$username'";
$result = mysqli_query($conn, $sql);
```

SQL injection

Limit Database Privileges: Ensure that the database user used by your application has only the necessary permissions to perform the required operations. Avoid using a superuser account for application tasks.

Validate and Sanitize Input: Validate and sanitize all user input before using it in database queries. Use functions like filter_var() or custom validation to ensure that the input adheres to the expected format.

Use ORM and Frameworks: Consider using Object-Relational Mapping (ORM) libraries or PHP frameworks that automatically handle SQL queries and data validation, reducing the risk of SQL injection.

SQL injection

Disable Error Reporting: Avoid displaying detailed error messages to users, as they might leak sensitive information. Use custom error handling to log errors securely.

Regularly Update Libraries: Keep PHP and database libraries up-to-date to take advantage of security patches and improvements.

Use Content Security Policies (CSP): Implement CSP to restrict the sources from which your application can load content, reducing the risk of executing injected scripts.

SQL injection

Disable Error Reporting: Avoid displaying detailed error messages to users, as they might leak sensitive information. Use custom error handling to log errors securely.

Regularly Update Libraries: Keep PHP and database libraries up-to-date to take advantage of security patches and improvements.

Use Content Security Policies (CSP): Implement CSP to restrict the sources from which your application can load content, reducing the risk of executing injected scripts.

SQL injection

```
<!DOCTYPE html>
<html>
<head>
  <title>Login Form</title>
</head>
<body>
  <h2>Login Form</h2>
  <form action="login.php" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" required><br>

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

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

```
<?php
// Check if the form was submitted
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  // Get the input data
  $username = $_POST["username"];
  $password = $_POST["password"];

  // Connect to the database (Replace with your database credentials)
  $conn = new mysqli("localhost", "username", "password",
"database_name");

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

Continue…!

SQL injection

```
// Prepare the SQL statement using a prepared statement
    $stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
    $stmt->bind_param("ss", $username, $password);

    // Execute the prepared statement
    $stmt->execute();

    // Get the result
    $result = $stmt->get_result();

    // Check if a matching user was found
    if ($result->num_rows == 1) {
        echo "Login successful!";
        // You can perform further actions for a successful login here
    } else {
        echo "Invalid username or password!";
    }

    // Close the statement and the database connection
    $stmt->close();
    $conn->close();
}
?>
```

Continue…!

Q & A

Thank you