International Cybersecurity and Digital Forensic Academy (ICDFA)

Course:

Web Application Security Essentials (OWASP Top 10) Vulnerabilities and Exploitation

Course code:

BVWS103

By

Chibunna Joe

Reg No.: 2025/FWSD/11522

Lab Title:

HTML INJECTION

Instructor:

Mr. Aminu Idris

Date Submitted:

January 25, 2026.

Chibunna Joe

Lab 2: HTML Injection

Objective:

By the end of this lab, it is expected to from students:

1. Understand HTML injection vulnerabilities and how they work.
2. Learn how an attacker can exploit HTML injection.
3. Apply mitigation techniques to prevent HTML injection vulnerabilities in your web applications.

Task 1: Exploit HTML Injection Vulnerability

Step 1: Set up Your Vulnerable Web Application

1. **Create a new folder** for your assignment. Name it Lab2_HTMLInjection.
2. Inside your folder, create two files:
    - index.html
    - server.php

Step 2: Review and Implement the Code

- Open your text editor (e.g., VS Code, Sublime Text) and create the following files with the given code.
- **index.php**: This file will contain a simple form where users can input their name.

html.php File

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>HTML Injection Example</title>
  </head>
  <body>
    <h1>HTML Injection Test</h1>
    <form action="server.php" method="post">
      <label for="name">Name:</label>
```

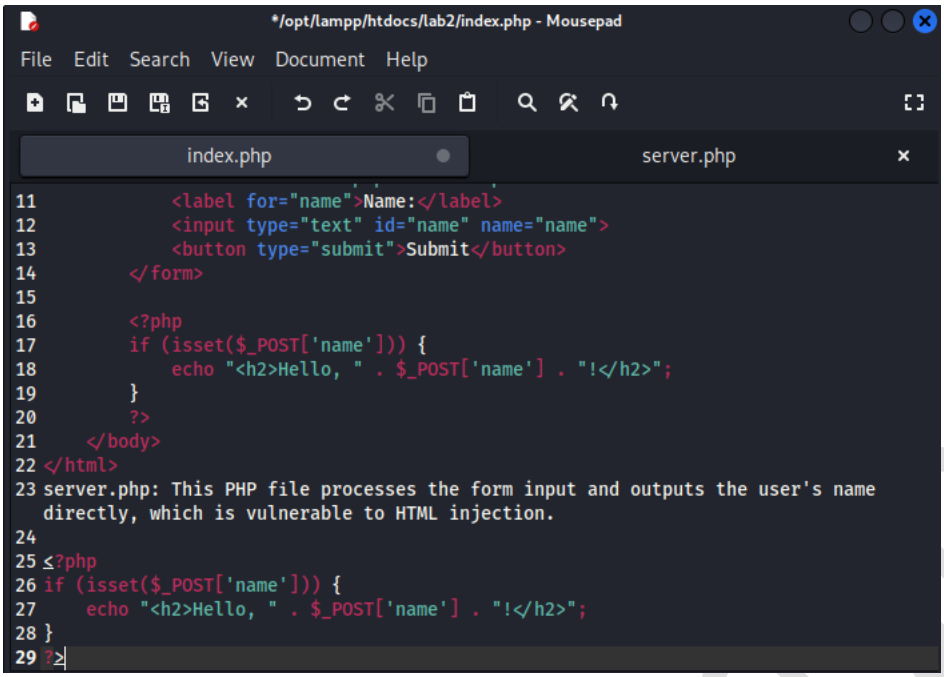
Chibuma Joe

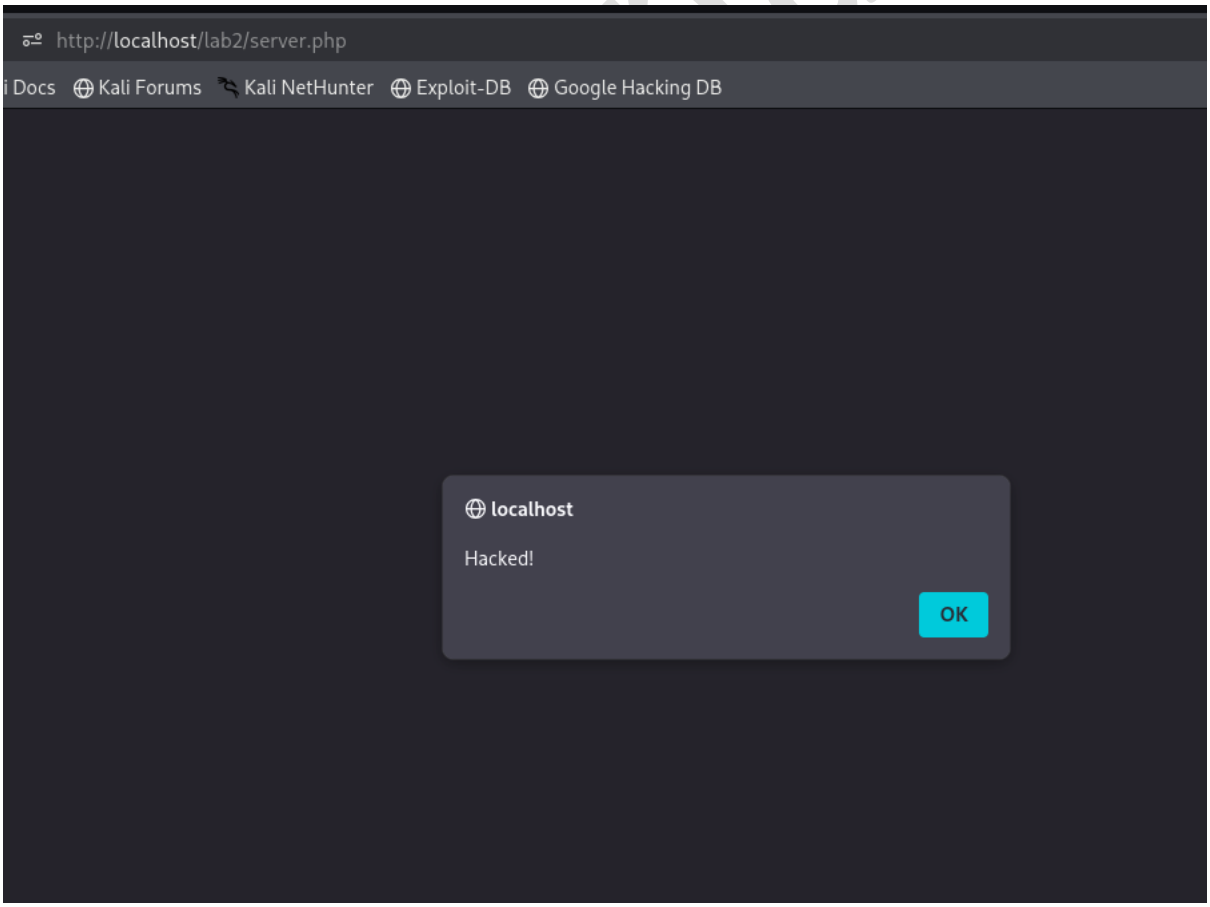
```
        <input type="text" id="name" name="name">
        <button type="submit">Submit</button>
    </form>
    <?php
    if (isset($_POST['name'])) {
        echo "<h2>Hello, " . $_POST['name'] . "!</h2>";
    }
    ?>
  </body>
</html>
```

Server.php file

```
<?php
if (isset($_POST['name'])) {
    echo "<h2>Hello, " . $_POST['name'] . "!</h2>";
}
?>
```

Chibuma Joe





1. Why does this code allow the JavaScript to execute?

The JavaScript executes because the application does not validate or sanitize user input before rendering it in the browser. To buttress the point, the value entered into the **"Name"** input field is inserted directly into the HTML page. So, the browser interprets the input as **HTML/JavaScript code**, not plain text. Since <script> tags are valid HTML elements, the browser executes the embedded JavaScript when the page loads or refreshes. In secure applications, user input should be:

- **Sanitized** (dangerous characters removed or escaped), or
- **Encoded** (e.g., converting < to <)

Because this application fails to do either, it becomes vulnerable to **HTML/JavaScript injection**, allowing arbitrary scripts to run in the user's browser.

2. What potential risks does this vulnerability pose in a real-world application?

In a real-world scenario, this vulnerability can have serious security and privacy implications, the potential risk of this vulnerability include:

1. Cross-Site Scripting (XSS) Attacks

Attackers can inject malicious scripts that:

- Steal session cookies or authentication tokens
- Hijack user sessions
- Impersonate legitimate users

2. Data Theft

Malicious scripts can:

- Capture keystrokes (keylogging)
- Extract sensitive user data such as login credentials or personal information
- Send stolen data to an attacker-controlled server

3. Website Defacement

An attacker can:

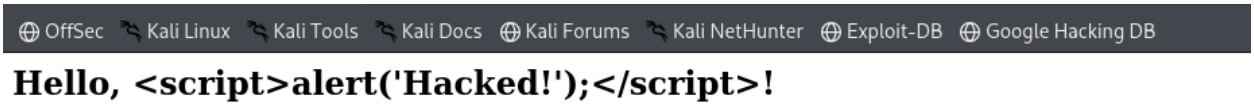
- Modify page content
- Display fake messages or phishing forms
- Damage the organization's credibility and trust

4. Malware Distribution

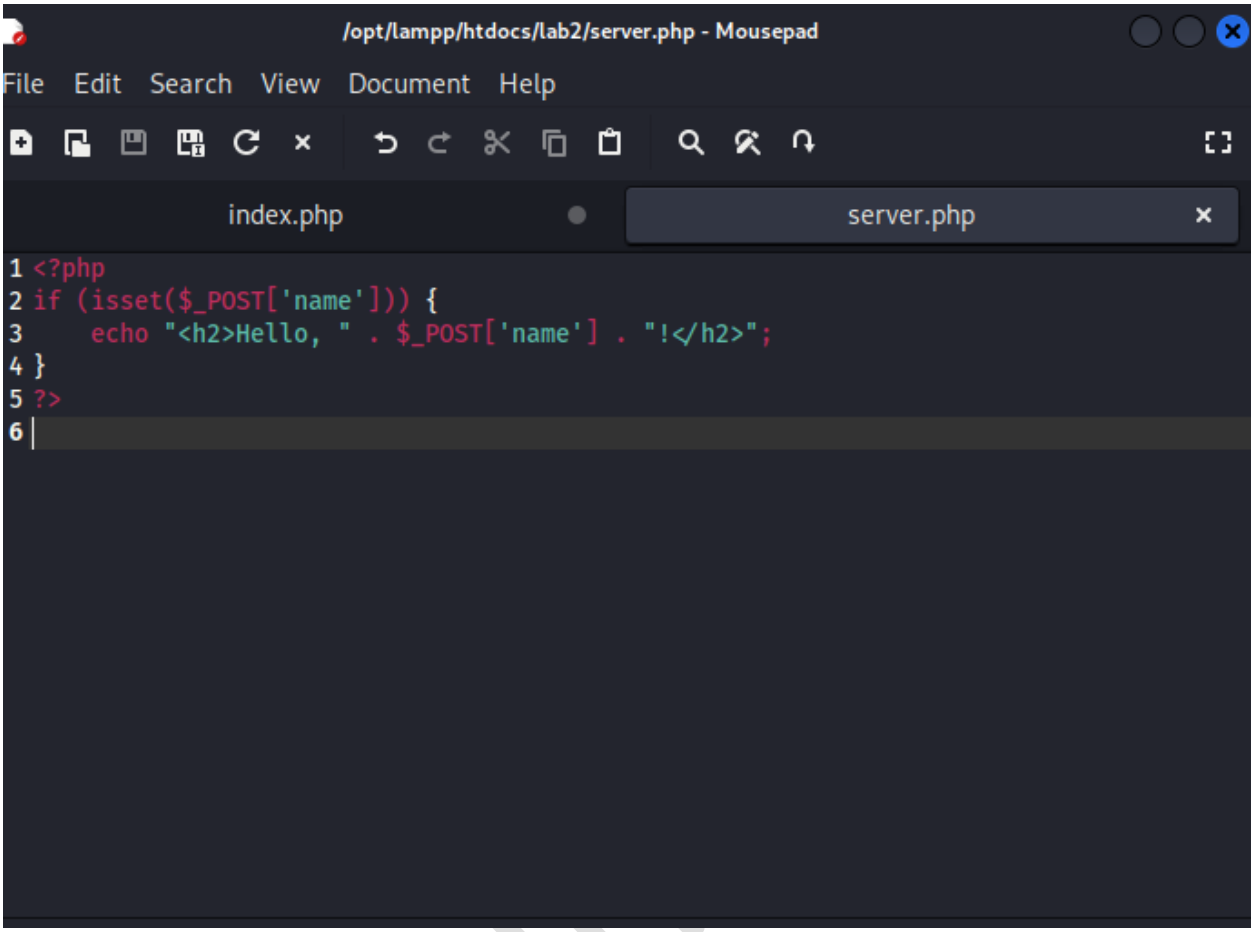
Chibuma Joe

Injected scripts can:

- Redirect users to malicious websites
- Force-drive downloads of malware without user consent



Chibunna



Reflections

1. What happens to the malicious input after applying htmlspecialchars()?

After applying htmlspecialchars(), the malicious input is converted from executable code into harmless text.

Specifically:

- Special HTML characters such as <, >, ', and " are escaped.
- For example:
    - < becomes <
    - > becomes >

So the input: <script>alert('Hacked!');</script>

is transformed into: <script>alert('Hacked!');</script>

Instead of being treated as HTML or JavaScript, the browser now treats the input as **plain text** and simply displays it on the page.

2. How does this prevent the code from being executed?

Browsers execute JavaScript only when it is interpreted as valid HTML/JavaScript code.

By escaping special characters:

- The <script> tag is no longer recognized as an HTML element.
- The browser does not parse it as JavaScript.
- The code is rendered literally on the page instead of being executed.

In a nutshell, htmlspecialchars() breaks the browser's ability to interpret user input as executable code, effectively neutralizing the attack. This is a core defense against: HTML Injection and Cross-Site Scripting (XSS)

3. How do you feel about the importance of input sanitization in web security?

This exercise highlights that **input sanitization is critical to web security**, not optional.

From this lab:

- A single line of unsanitized input allowed JavaScript execution.
- One security function (htmlspecialchars()) completely prevented the attack.

This shows that:

- User input can **never be trusted**, regardless of its source.
- Even simple forms can become attack vectors.
- Proper sanitization protects both **users and the application owner** from serious threats such as data theft, session hijacking, and website defacement.

In summary, this reinforces my understanding that **secure coding practices are a fundamental responsibility of developers**, and that input sanitization is one of the simplest yet most effective ways to prevent client-side attacks.

Executive Summary

Description of the HTML Injection Vulnerability

HTML injection is a web application vulnerability that occurs when user input is improperly handled and directly rendered in a web page without validation or sanitization. In this lab, the application accepted user input from a form field and displayed it back to the browser without escaping special characters. As a result, the browser interpreted the input as executable HTML and JavaScript code rather than plain text. This weakness allows attackers to inject malicious scripts that can be executed in a user's browser, potentially leading to Cross-Site Scripting (XSS) attacks.

chibuma joe

How I Exploited the Vulnerability in Task 1

In Task 1, I exploited the vulnerability by entering the payload <script>alert('Hacked!');</script> into the input field and submitting the form. After submission, the browser executed the script and displayed a pop-up message stating “Hacked!”. This confirmed that the application was vulnerable and that client-side code could be injected and executed. Performing this step helped me clearly understand how attackers can manipulate unsanitized inputs to compromise web applications.

How I Fixed the Vulnerability in Task 2 Using htmlspecialchars()

In Task 2, I mitigated the vulnerability by applying the htmlspecialchars() function to the user input before it was rendered on the page. When I submitted the same malicious payload again, the script did not execute. Instead, the input was displayed as encoded text (<script>alert('Hacked!');</script>). This showed that the browser could no longer interpret the input as executable code. Implementing this fix demonstrated how proper output encoding can effectively prevent HTML injection and XSS attacks.

Lessons Learned

From completing this assignment, I learned that user input should never be trusted and must always be validated and sanitized before being processed or displayed. I also realized that small coding oversights can lead to serious security vulnerabilities. This lab strengthened my understanding of secure coding practices and highlighted the importance of implementing preventive controls such as input validation and output encoding. Overall, the exercise improved my awareness of how easily web applications can be exploited and how essential it is to integrate security measures during development rather than after deployment.

chiuma joe