

TASK – 2(a)

CODEALPHA

OBJECTIVE

- In this task, we will review the code for security vulnerabilities and provide recommendations for secure coding practices.
- First we will use a code analyzer and then manual inspection to identify and fix potential issues.

CODING LANGUAGE USED FOR REVIEW

- The language used for coding review is **HTML**.
- HTML stands for HyperText Markup Language. It is the standard markup language used to create and design documents on the World Wide Web. HTML describes the structure of a web page and its content, using a series of elements to enclose different parts of the content to make it appear or act a certain way.

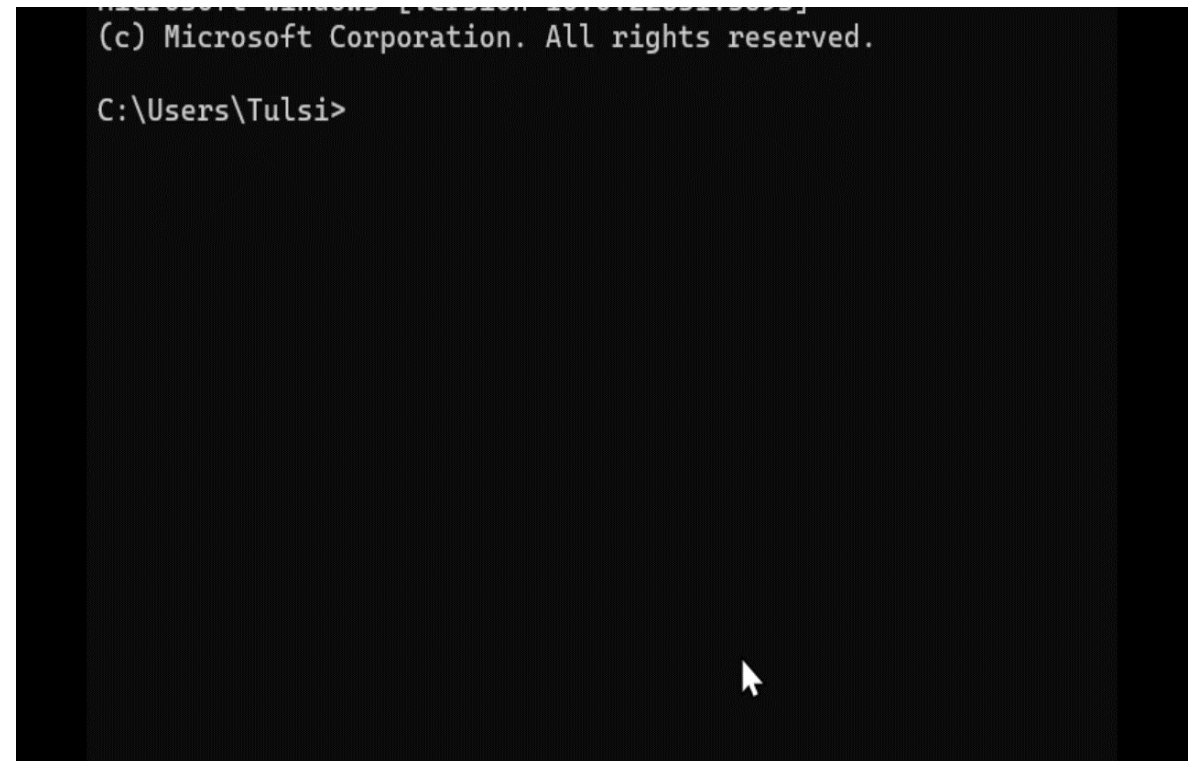
CODE

This code here will be reviewed using htmlhint and manually.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Sample Web Page</title>
6      <script>
7          function showMessage() {
8              const message = new URLSearchParams(window.location.search).get('message');
9              document.getElementById('message').innerHTML = message;
10         }
11     </script>
12 </head>
13 <body onload="showMessage()">
14     <h1> <center> Welcome to the Sample Web Page </center> </h1>
15     <center> <div id="message"></div>
16     <form action="/submit" method="post">
17         <input type="hidden" name="csrf_token" value="static_csrf_token">
18         <button type="submit">Submit</button>
19     </form> </center>
20 </body>
21 </html>
22
```

REVIEWING CODE USING HTMLHINT

- Here I have used htmlhint as a code analyzer.
- Htmlhint is effective for identifying common HTML coding errors and enforcing best practices, it has limitations and may not cover all security vulnerabilities.
- Manual code reviews can uncover security vulnerabilities by inspecting the logic and implementation details that automated tools like htmlhint might miss.



```
Microsoft Windows [Version 10.0.17134.1]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Tulsi>htmlhint
```

IDENTIFIED VULNERABILITIES IN THE HTML CODE

After the manual code review, these are the identified vulnerabilities

- Cross-site scripting: The 'showMessage' function directly injects user input into the HTML without sanitization, making it vulnerable to XSS attacks.
- Static CSRF token: The CSRF token is static, which makes it ineffective in preventing CSRF attacks. CSRF tokens should be dynamic and unique for each session or request.
- Center Tag (Deprecated): The '<center>' tag is deprecated in HTML5. Although this is not a security vulnerability, it's a best practice to use CSS for styling.

RECOMMENDATIONS FOR SECURE CODING PRACTICES

1. Prevent Cross-Site Scripting (XSS)

Original code:

```
const message = new URLSearchParams(window.location.search).get('message');  
document.getElementById('message').innerHTML = message;
```

Updated code:

Use 'textContent' instead of 'innerHTML' to prevent XSS.

```
const message = new URLSearchParams(window.location.search).get('message');  
document.getElementById('message').textContent = message;
```

2. Implement Proper CSRF Protection

Original code:

```
<input type="hidden" name="csrf_token" value="static_csrf_token">
```

Updated code:

Use dynamic CSRF tokens generated by the server for each session or request.

```
<input type="hidden" name="csrf_token" value="dynamic_csrf_token">
```

Ensure that the server-side application generates and validates the CSRF token for each request.

3. Replace Deprecated '<center>' Tag

Original code:

```
<h1> <center> Welcome to the Sample Web Page </center> </h1>
<center> <div id="message"></div>
<form action="/submit" method="post">
|   <input type="hidden" name="csrf_token" value="{{ csrf_token }}">
|   <button type="submit">Submit</button>
</form> </center>
```

Use CSS for centering content.

Updated code:

This is the final fixed code.

```
<head>
  <meta charset="UTF-8">
  <title>Sample Web Page</title>
  <style>
    .center {
      text-align: center;
    }
  </style>
  <script>
    function showMessage() {
      const message = new URLSearchParams(window.location.search).get('message');
      document.getElementById('message').textContent = message; // Use textContent to prevent XSS
    }
  </script>
</head>
<body onload="showMessage()">
  <h1 class="center">Welcome to the Sample Web Page</h1>
  <div id="message" class="center"></div>
  <form action="/submit" method="post" class="center">
    <input type="hidden" name="csrf_token" value="dynamic_csrf_token">
    <button type="submit">Submit</button>
  </form>
</body>
```

CROSS-SITE SCRIPTING (XSS)

Definition: Cross-site scripting is a type of security vulnerability that allows attackers to inject malicious scripts into web pages that are used by other users and these scripts can manipulate the content of the website, steal data, or perform other malicious activities. There are three main types of XSS

- **Persistent XSS(Stored XSS):** A type of XSS where the malicious script is permanently stored on the target server and executed when a user accesses the stored data.
- **Non Persistent XSS(Reflected XSS):** A type of XSS where the malicious script is reflected off a web application and executed immediately in the user's browser as part of a specific request.
- **DOM-based XSS:** A type of XSS where the vulnerability lies in the client-side code, and the malicious script manipulates the Document Object Model (DOM) environment of the web page.

HOW XSS ATTACKS WORK

- **Injection:** The attacker injects a malicious script into a web page or application.
- **Execution:** The malicious script executes in the context of the user's browser. This script can perform various malicious actions, such as stealing cookies, session tokens, or other sensitive information.
- **Impact:** The attacker can hijack user sessions, deface websites, spread malware, or redirect the user to malicious websites.

PREVENTING XSS ATTACKS

- **Input Validation and Sanitization:** Validate and sanitize all user inputs. Ensure that data is properly escaped before it is included in web pages.
- **Content Security Policy (CSP):** Implement a strong Content Security Policy to restrict the sources from which scripts can be loaded.
- **Encoding Output:** Encode data before rendering it in the browser. Use the appropriate encoding for the context (HTML, JavaScript, URL, etc.).
- **Use Security Libraries and Frameworks:** Utilize security features provided by modern web development frameworks that automatically handle input sanitization and encoding.
- **Regular Security Testing:** Regularly test your web applications for vulnerabilities using automated tools and manual penetration testing.