MSCS-535: Project 2: Project Report: JavaScript Code Injection & Secure Web App Design - Final Report

---

Sandesh Pokharel

University of the Cumberlands

MSCS-535 Secure Software Development

Brandon Bass

March 30, 2025

🔍 Introduction

This project was developed to demonstrate the risks associated with JavaScript code injection via web applications and how such vulnerabilities can be exploited by attackers. Specifically, this project highlights:

- Cross-Site Scripting (XSS) vulnerabilities
- Dynamic evaluation of code using JavaScript's eval() function
- Effective mitigation strategies to secure web applications

We built two versions of a simple comment-posting web application:

- A **vulnerable version** intentionally designed to allow code injection
- A **secure version** hardened against XSS and code injection threats

🔗 GitHub Repository

You can find the full project source code on GitHub at the following link:

GitHub: https://github.com/sanspokharel26677/MSCS-535-Project2

💡 Objective

The main goals of this project were:

- To simulate a realistic web vulnerability scenario
- To demonstrate how attackers can exploit eval() and unsanitized input
- To implement effective security measures using secure development principles
- To validate the fixes by testing and comparing vulnerable vs secure behavior

⚠️ Vulnerable Version Overview

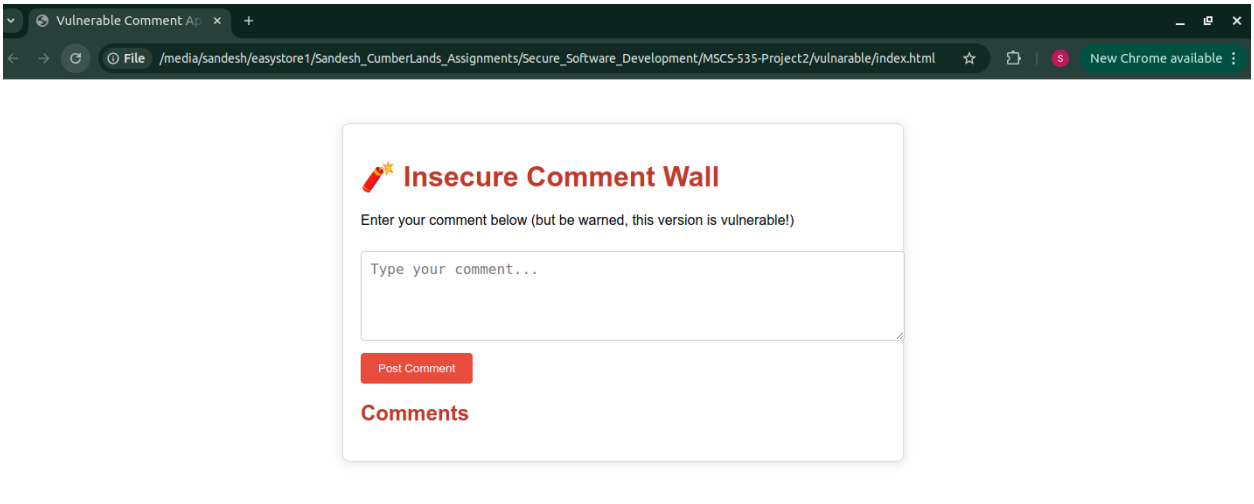
The vulnerable version includes the following issues:

- User input is rendered using innerHTML without any sanitization.
- The application uses eval() to evaluate user input.
- Inline onclick handlers are used, which violate strong CSP rules.

Key Features of the Vulnerability:

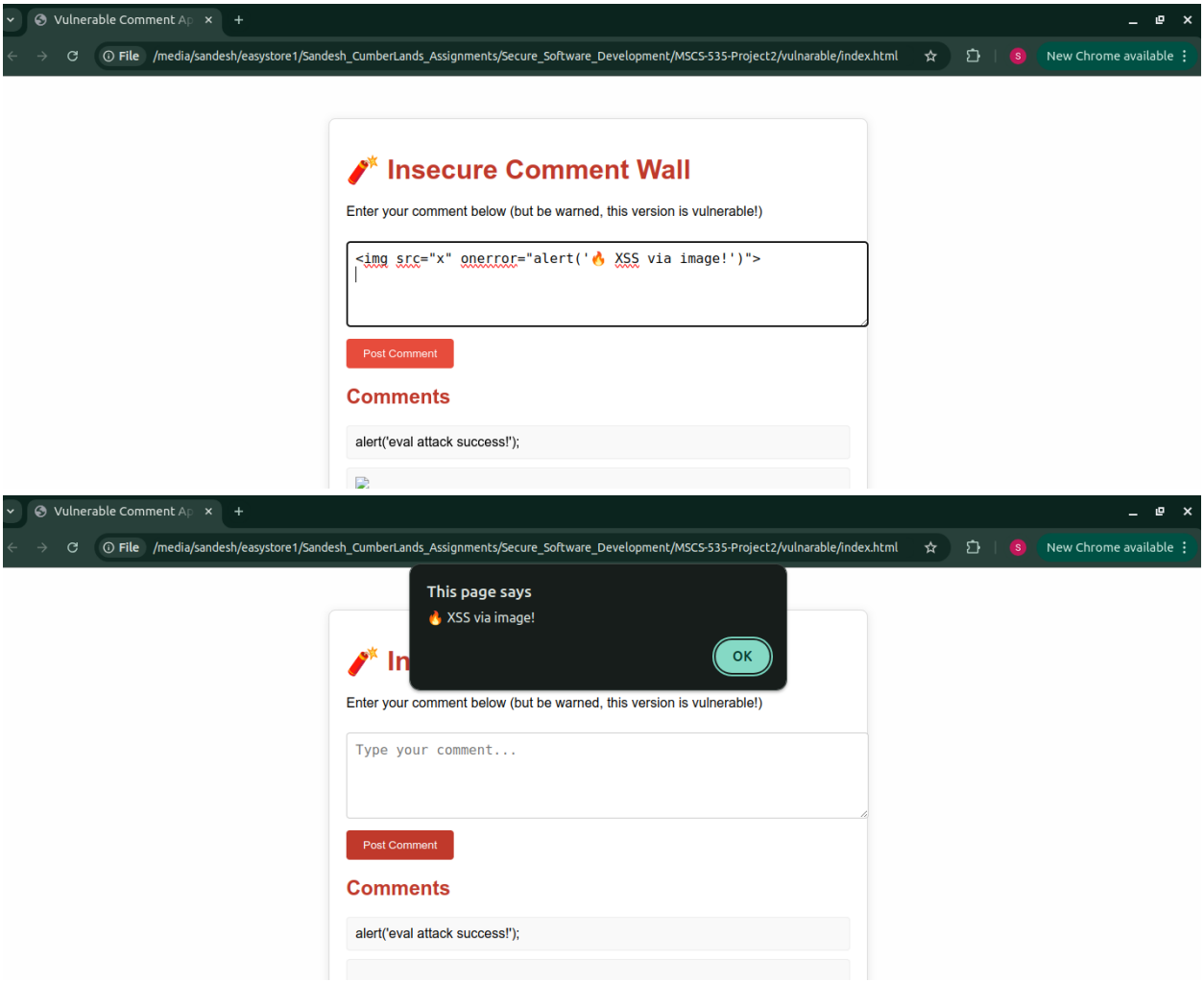
- Allows **Reflected XSS** using input like <script>alert('XSS')</script>
- Allows **Persistent XSS** using <img src=x onerror=alert('XSS')>
- Executes arbitrary code via eval(), such as alert('Hacked!')

Screenshot 1:



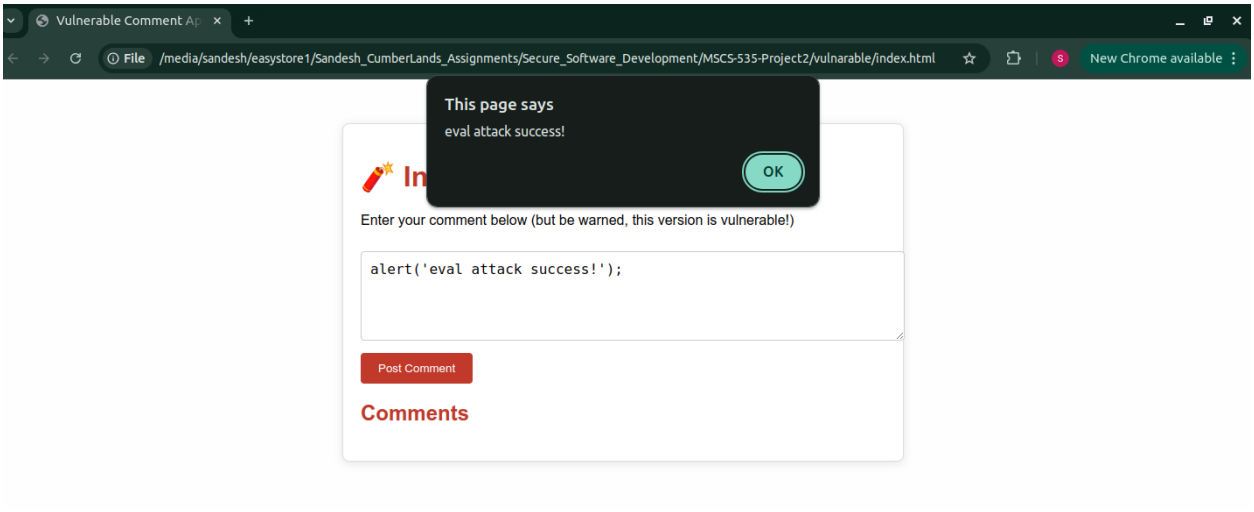
Screenshot of the vulnerable app's UI with the comment input box and post button.

Screenshot 2:



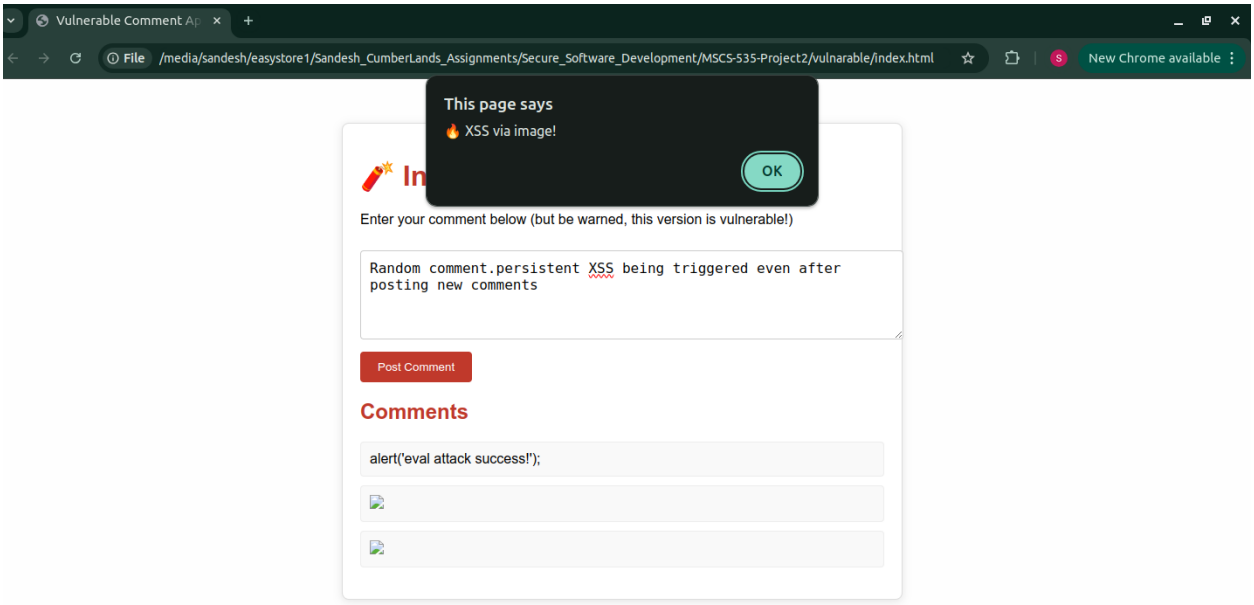
Screenshot showing an alert triggered from an XSS payload.

📂 Screenshot 3:



Screenshot showing eval() execution with an input like alert('Eval worked!').

📂 Screenshot Placeholder 4:



Screenshot of persistent XSS being triggered even after posting new comments.

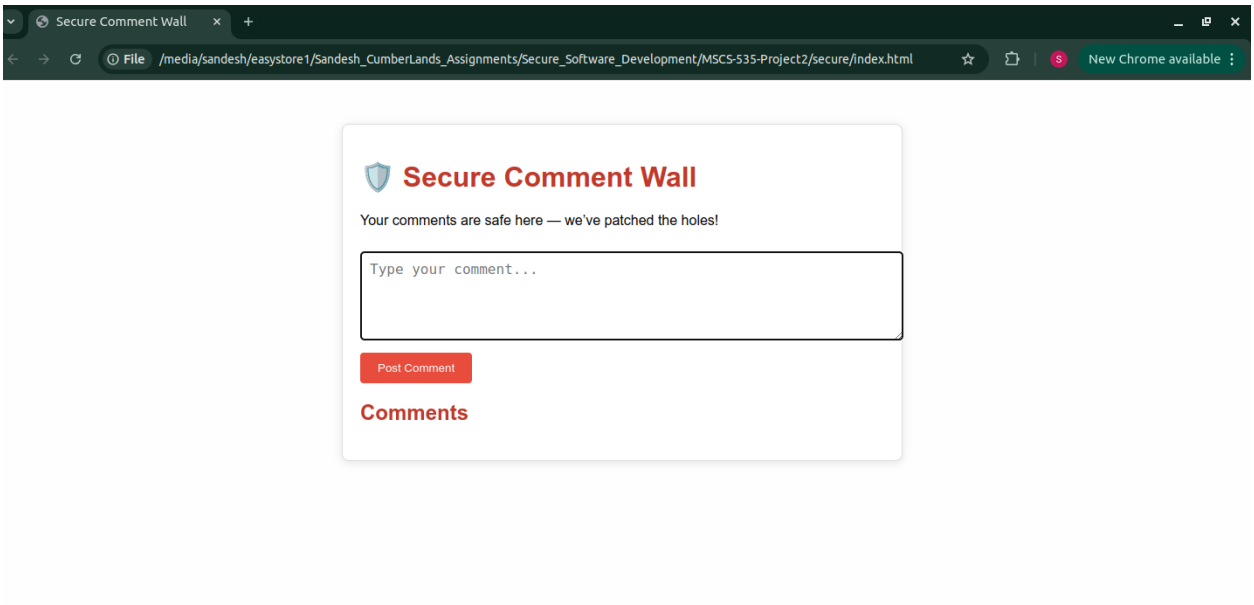
☑ Secure Version Overview

The secure version implements the following mitigations:

- Replaces innerHTML with innerText to avoid script injection.
- Removes use of eval() completely.
- Sanitizes all user inputs using a custom sanitize() function.
- Replaces inline event handlers with addEventListener().
- Adds a strict Content Security Policy (CSP) using meta tag:

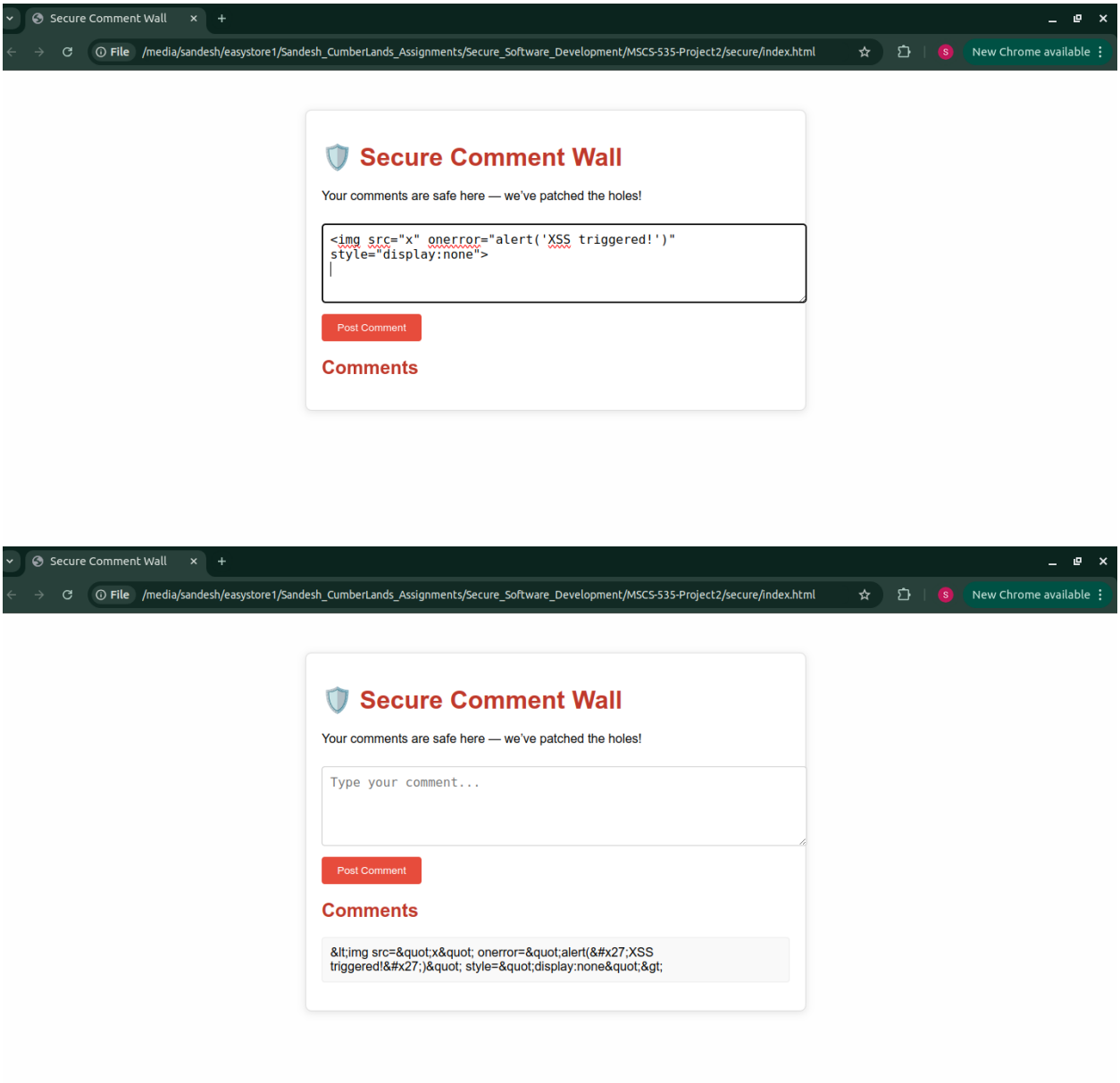
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self';">

Screenshot 5:



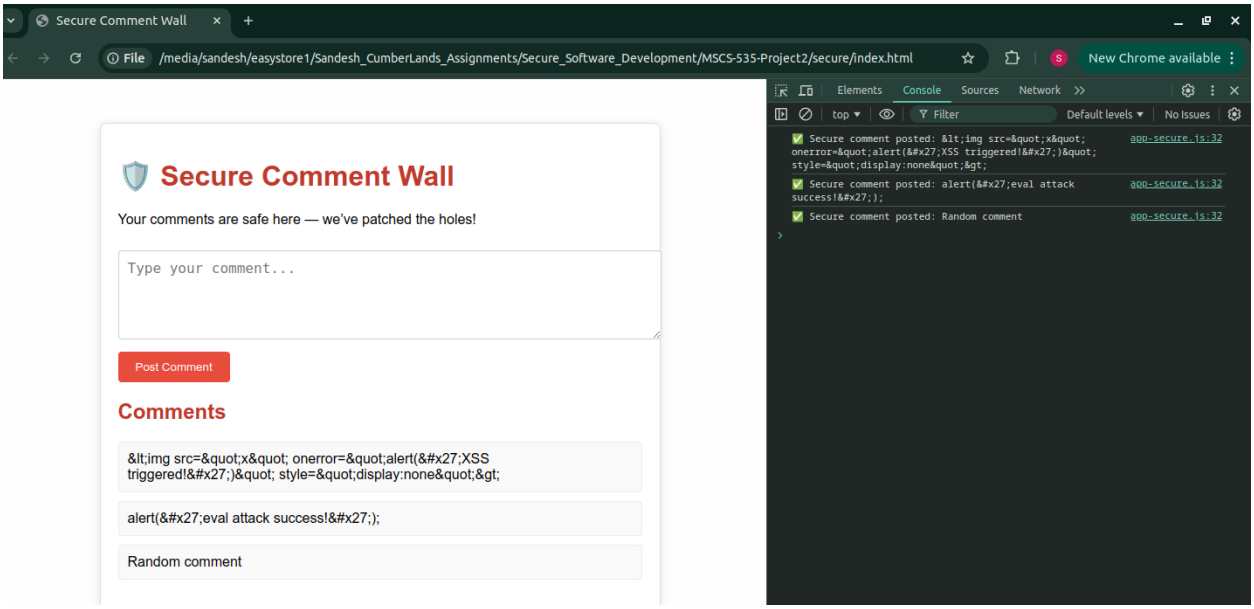
Screenshot of the secure app's UI with comment input box and post button.

Screenshot 6:



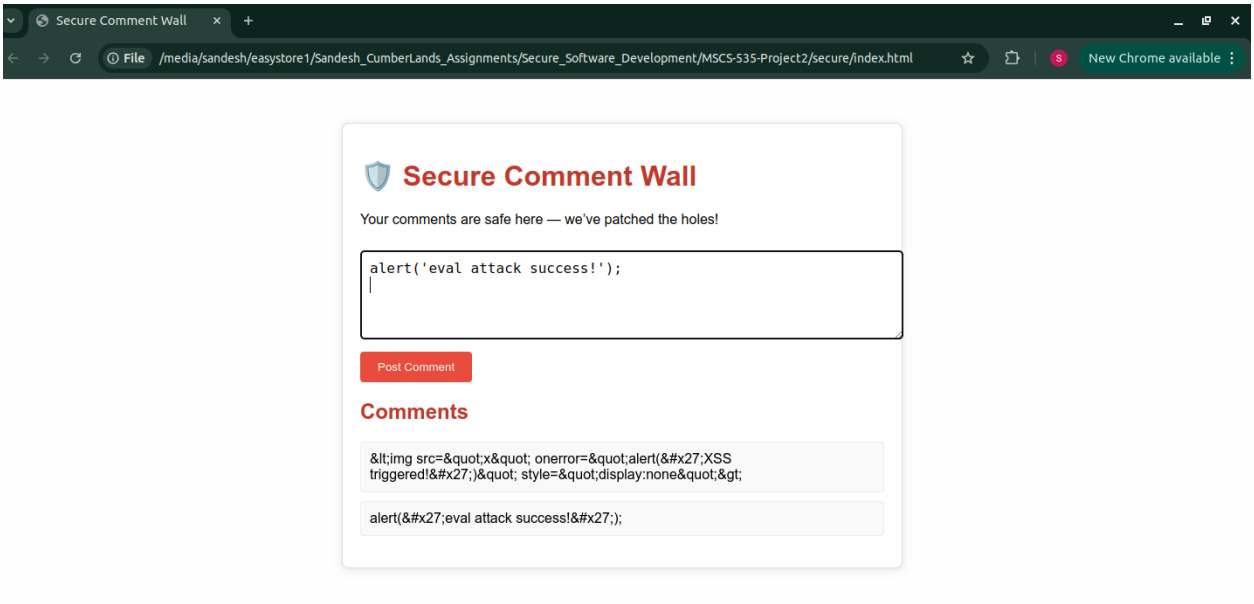
Screenshot of a malicious XSS payload showing up as escaped text instead of executing.

Screenshot 7:



Screenshot of the browser console logging a successful secure comment post.

Screenshot 8:



Screenshot proving that input like <script>...</script> or alert('...') does not trigger any code execution.

🧪 Testing & Results

We ran multiple test cases with the following payloads:

- <script>alert('XSS')</script>
- <img src=x onerror=alert('XSS')>
- alert('Eval run')

Test Case	Vulnerable Version	Secure Version
<script>...</script>	✅ Executes alert	❌ Rendered as text
<img src=... onerror=...>	✅ Executes alert persistently	❌ Rendered as text
alert('...')	✅ Executes via eval	❌ Rendered as text

🧠 Lessons Learned

- Using innerHTML with unsanitized input is extremely risky.
- eval() should **never** be used on user input.
- Basic input sanitization and innerText alone prevent most basic XSS attacks.
- Enforcing CSP and avoiding inline JS improves overall security posture.

🧰 Tools & Technologies Used

- HTML / CSS / JavaScript
- Modern browser (Chrome, Firefox)
- Python 3 HTTP server for local testing

Folder Structure

```
MSCS-535-Project2/
├── vulnerable/
│   ├── index.html
│   ├── style.css
│   └── app-vulnerable.js
├── secure/
│   ├── index.html
│   ├── style.css
│   └── app-secure.js
└── README.md
```

Conclusion

This project successfully demonstrates both the dangers and solutions around JavaScript injection. The vulnerable version clearly shows how attackers can exploit unsanitized input and dynamic code execution, while the secure version showcases simple but effective mitigations that align with secure coding practices. The project reflects the importance of input validation, safe DOM manipulation, and strict browser-side policies in modern web application security.