Title: Activity 1 – HTTPS Key Exchange Analysis

S1: Wireshark Setup and Filtering

Wireshark was launched on the loopback interface (lo0) with the display filter:

tcp.port==8443 and tcp and not tcp.len==0

The following columns were enabled: No., Time, Source, Src port, Destination, Dest port, Protocol, Length, Info.

S2: Trigger HTTPS Request via Firefox

A local HTTPS server (https_echo.js) was run using Node.js on https://localhost:8443. Firefox was used to access the page, and a message was sent via the text box to initiate a handshake.

S3a: TLS Handshake Packet Screenshots

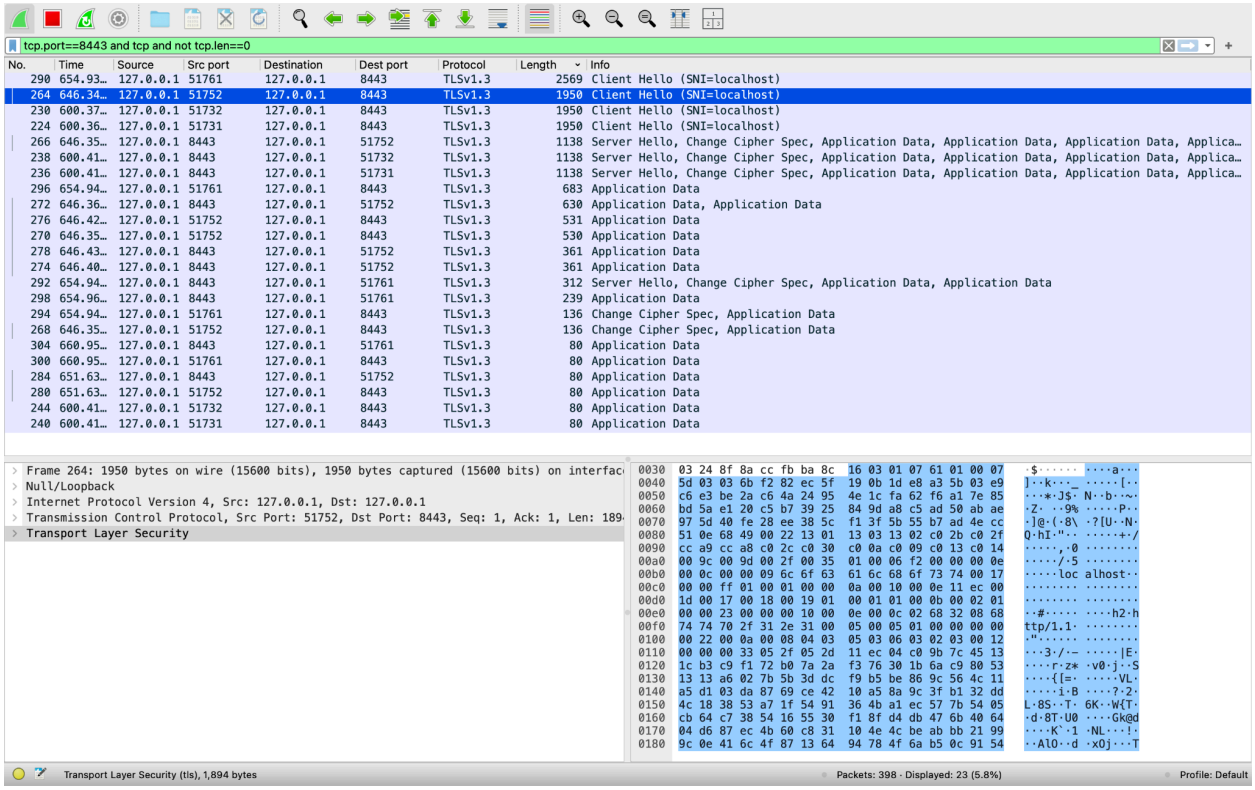


Figure 1: Client Hello – Packet #264

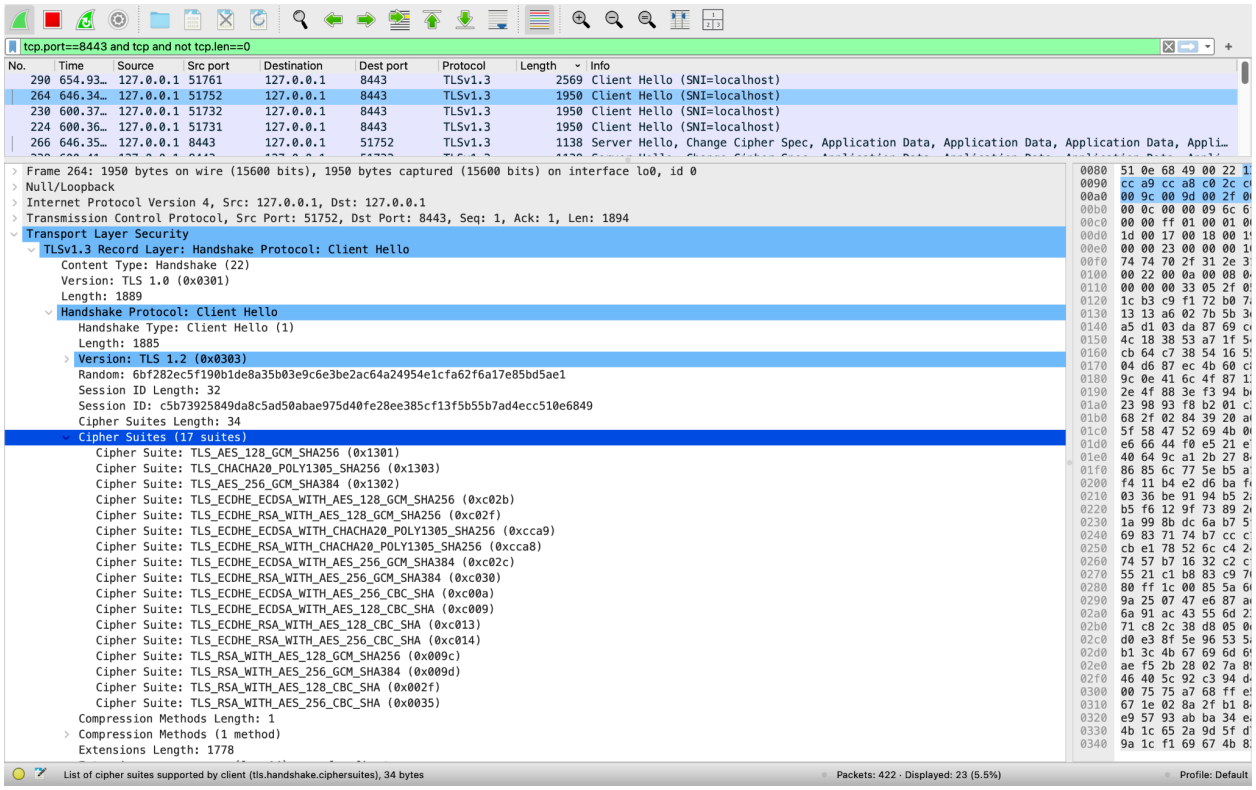


Figure 2: Server Hello – Packet #266

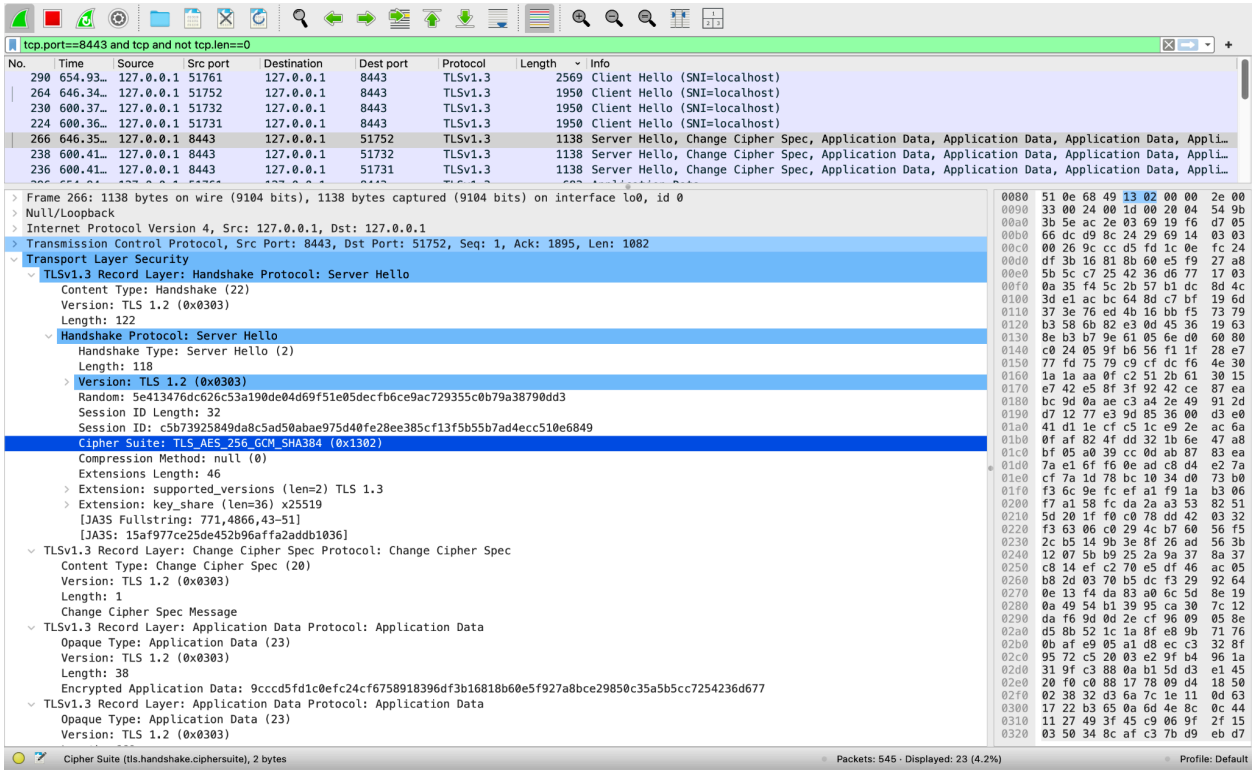


Figure 3: Encrypted Application Data – post-handshake

The handshake process between client and server was successfully captured using Wireshark.

- **Client Hello** initiates the handshake and proposes cipher suites, protocol version (TLS 1.3), session ID, and client random value.

- **Server Hello** responds by selecting a cipher suite (TLS_AES_256_GCM_SHA384), protocol version, session ID, and includes the server random value.

- Following this, the handshake proceeds with encrypted packets (Application Data), indicating that both client and server have successfully exchanged keys and transitioned to a secure encrypted session.

S3b: Cipher Suites

📎 **Screenshots**:

- Cipher suites list from Client Hello

- Server's selected cipher suite from Server Hello

The Client Hello proposed 17 cipher suites including:

- **TLS_AES_128_GCM_SHA256**

- **TLS_AES_256_GCM_SHA384**

- **TLS_CHACHA20_POLY1305_SHA256**

- **Multiple older RSA-based cipher suites**

The Server Hello chose:

- **TLS_AES_256_GCM_SHA384**

This is a strong cipher suite supporting authenticated encryption with AES and SHA-384, appropriate for TLS 1.3.

S3c: Client Random & Server Random

- **Client Random** (from Client Hello):

  6fbf228ec5f19bb1de83b50e39c6e3be2ac6a424954e1cfa62f6a17e85bd5ae1

- **Server Random** (from Server Hello):

  5e413476d626c53a190e0d46f951e05decfb6ce9ac729355c0b79a38790dd3

These random values are used during key exchange to generate the session keys for encrypted communication. Each handshake will produce new random values, ensuring perfect forward secrecy.

S4a: Firefox Certificate Details

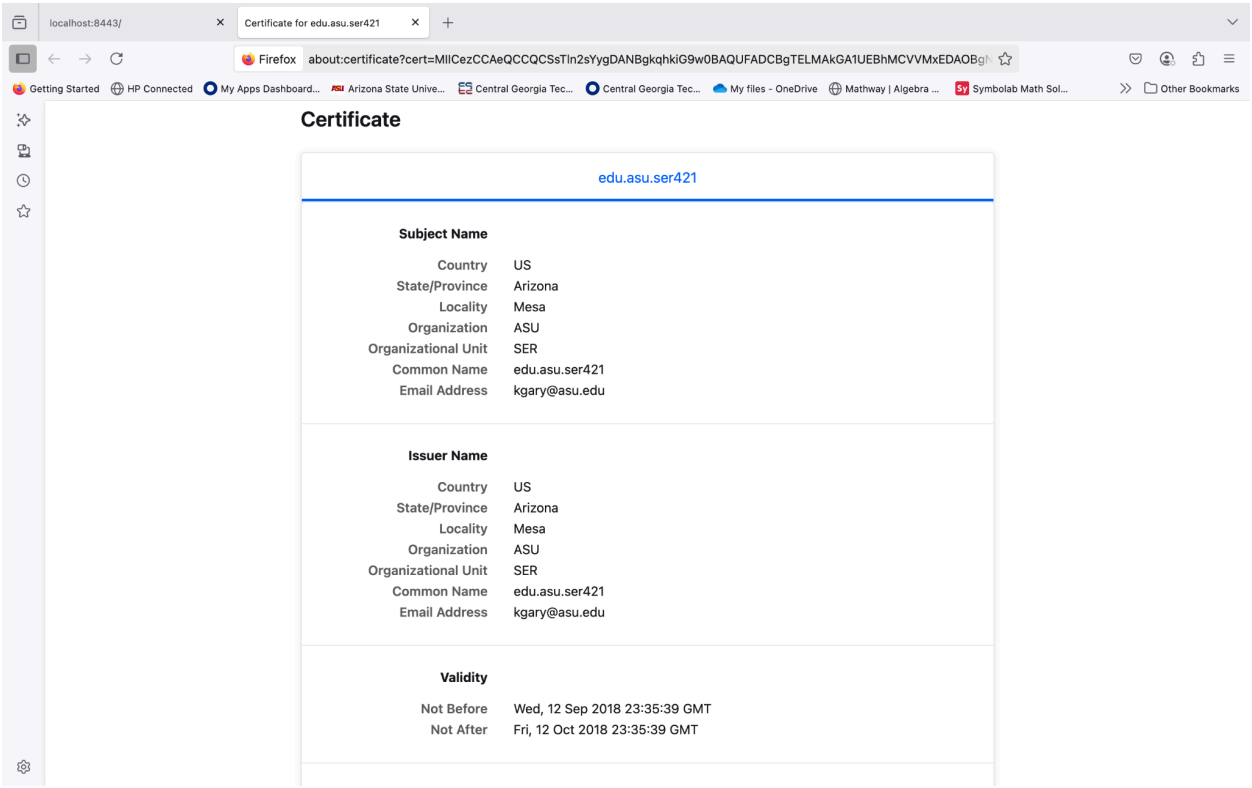


Figure 4: *Certificate Subject and Issuer Information*

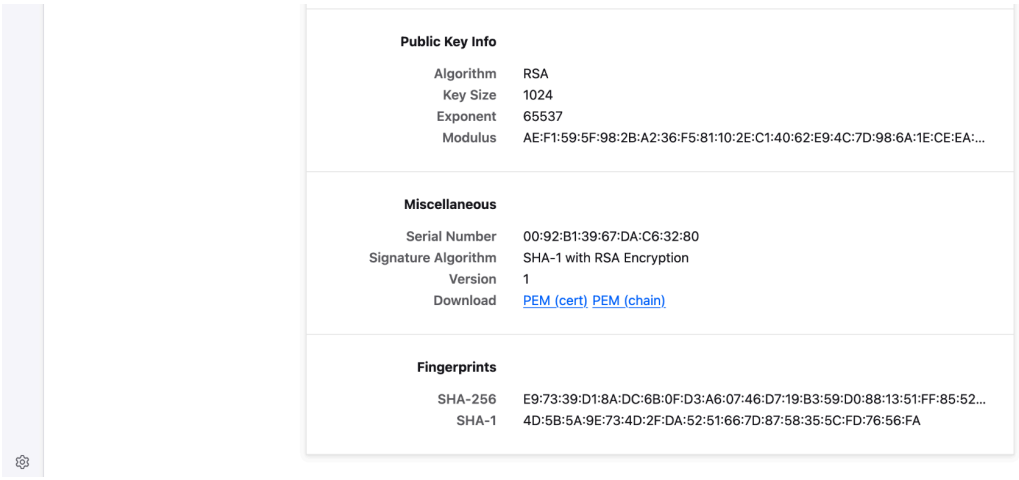


Figure 5: *Public Key, Signature, and Fingerprints*

Firefox was used to inspect the HTTPS certificate served by the local Node.js server at https://localhost:8443. The certificate contained the following details:

Subject and Issuer:

- Country: US
- State/Province: Arizona
- Locality: Mesa
- Organization: ASU
- Organizational Unit: SER
- Common Name: edu.asu.ser421
- Email: kgary@asu.edu

Validity Period:

- Not Before: Wed, 12 Sep 2018 23:35:39 GMT
- Not After: Fri, 12 Oct 2018 23:35:39 GMT

Public Key Info:

- Algorithm: RSA
- Key Size: 1024 bits
- Exponent: 65537
- Signature Algorithm: SHA-1 with RSA Encryption

Fingerprints:

- SHA-256: E97339D18ADC6B0FD3A60746D719B359D0881351FF…
- SHA-1: 4D5B5A9E734D2FDA5251667D8758355CFD7656FA

This certificate was **self-signed**, meaning the issuer and subject are the same (edu.asu.ser421). It is not trusted by default browser root CAs.

S4b: Browser Trust Warning

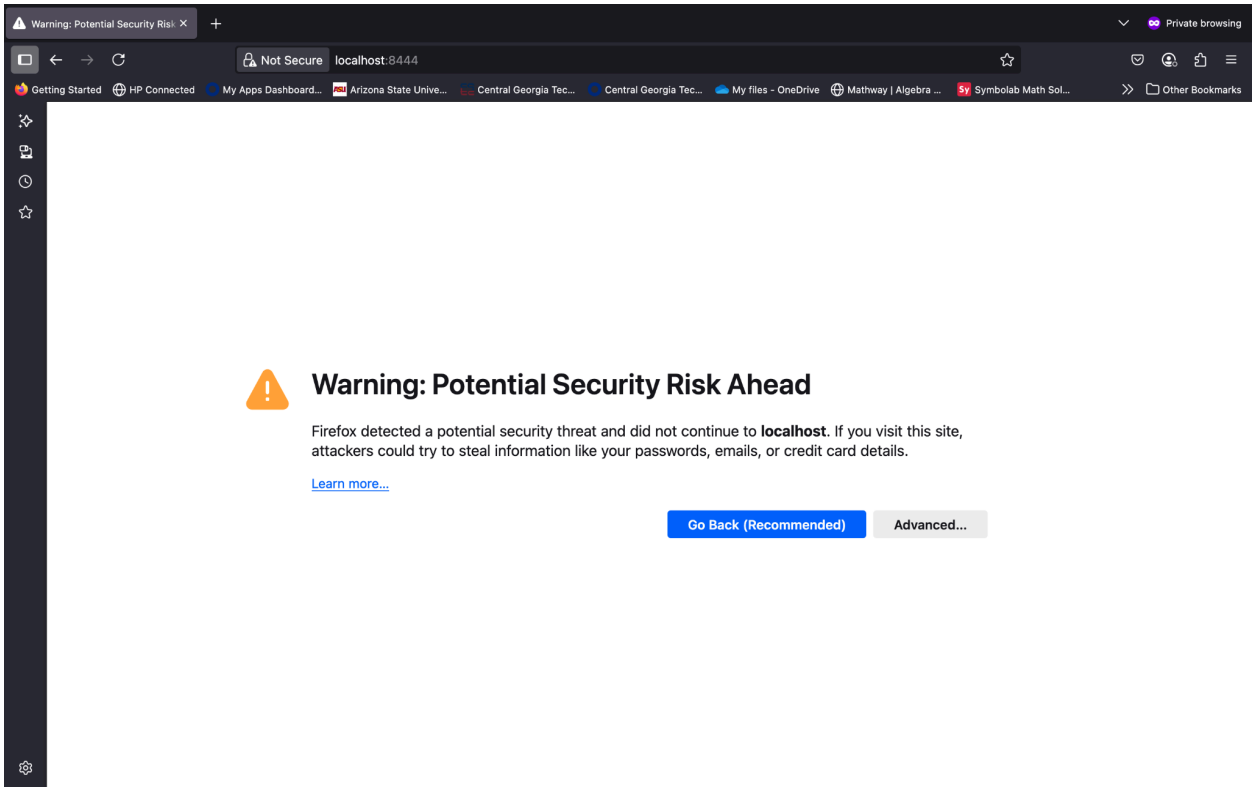


Figure 6: *Firefox Security Warning for Self-Signed Certificate*

When opening the page in a **private Firefox window**, the browser showed a **"Warning: Potential Security Risk Ahead"** message. This is expected behavior when:

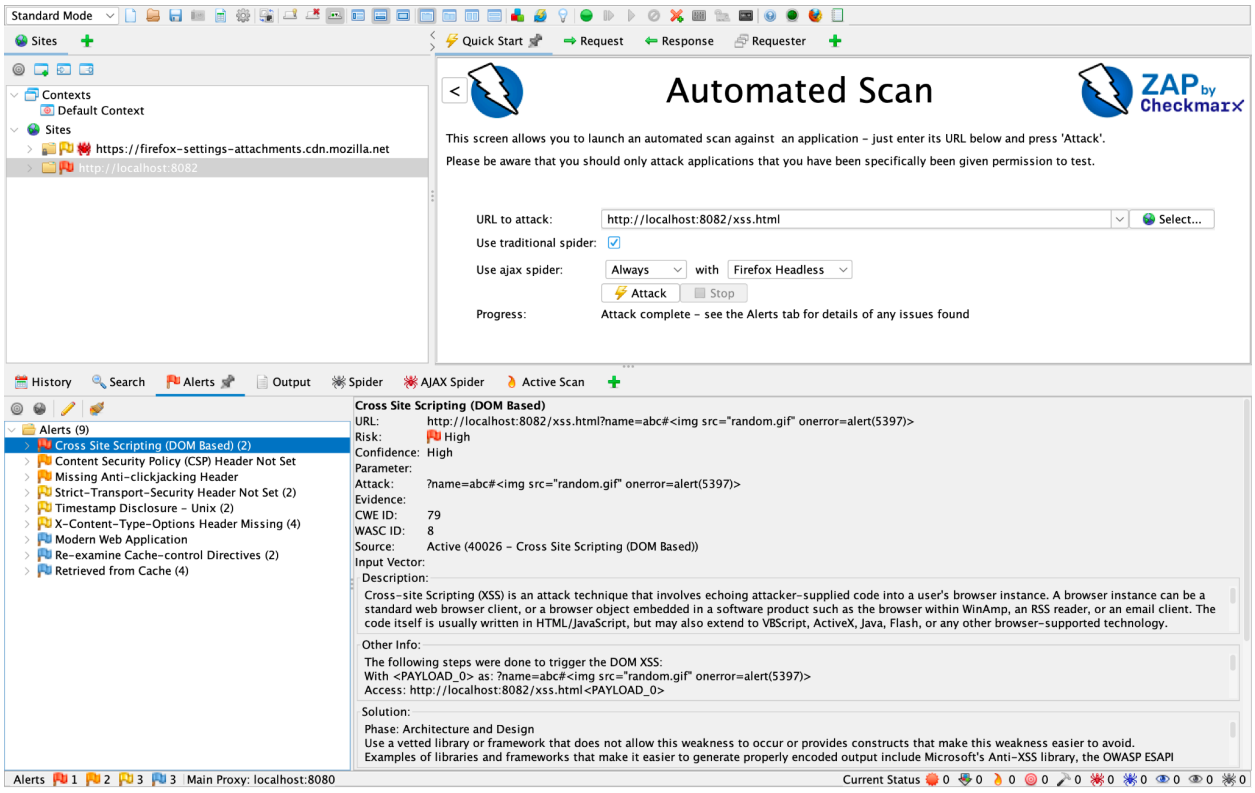
- The certificate is self-signed.
- The certificate is expired.
- The certificate is not issued by a known Certificate Authority (CA).

Users must manually accept the risk and proceed. This behavior demonstrates how browsers protect users from untrusted or improperly configured HTTPS endpoints.

Activity 2 – Written Report Section

S5.a. Full URL

http://localhost:8082/xss.html?name=abc#<img src="random.gif" onerror=alert(5397)>



S5.b. Type of XSS Attack/Vulnerability - Cross-Site Scripting (DOM-Based)

S5.c. Steps to Exploit the XSS Vulnerability

1. **Access the Vulnerable Page**

   The attacker targets the vulnerable xss.html page, which fails to sanitize user input embedded in the URL.

   Example malicious URL:

   http://localhost:8082/xss.html?name=abc#<img src="random.gif" onerror=alert(5397)>

2. **Inject Malicious Script**

   The attacker injects a payload via the name parameter using an <img> tag with an onerror attribute. This JavaScript executes when the browser attempts to load a nonexistent image.

3. **Trigger Script Execution**

   When the crafted URL is opened in a browser, the malicious content is injected into the DOM without proper sanitization. The onerror event triggers, executing alert(5397) as proof of script execution.

4. **Confirm the Exploit**

   A popup alert appears in the browser, verifying that the injected JavaScript has successfully executed—confirming the existence of a reflected XSS vulnerability.

S5.d. CWE Analysis and Recommended Mitigation

- **CWE ID**: 79
- **Title**: Cross-site Scripting (XSS)
- **Description**: This vulnerability allows attackers to inject and execute malicious scripts in web pages viewed by other users. The scripts can hijack sessions, steal credentials, or manipulate content on behalf of the user.
- **Likelihood of Exploit**: High
- **Recommended Mitigation**:

  All user-supplied input should be strictly validated and sanitized before being rendered in the browser. Use output encoding libraries (e.g., OWASP Java Encoder) to safely display dynamic content. Implementing a **Content Security Policy (CSP)** can add a second layer of defense by restricting the execution of inline scripts and untrusted sources.