

# SEED Labs: XSS Lab

Shivali Mate

## I. INTRODUCTION

This lab explored how Cross-Site Scripting (XSS) vulnerabilities can be exploited in a web application and how attackers leverage injected JavaScript to steal data, modify user profiles, and create self-propagating worms. Using a modified version of the Elgg social network, attacks similar to the Samy Worm were reproduced that spread across MySpace in 2005. The lab combined offensive tasks such as cookie theft, forged requests, and worm propagation with defensive tasks using Content Security Policy (CSP). These exercises provided hands-on experience with core web security concepts including session cookies, HTTP requests, browser trust models, and client-side defenses.

## II. ENVIRONMENT AND SETUP

The lab was performed inside a SEEDUbuntu environment where Elgg and multiple CSP-configured virtual hosts were deployed using Docker containers. DNS entries were added to map several example domains to the server container at 10.9.0.5, enabling realistic testing across multiple websites. Access to the Elgg server allowed testing of XSS payloads, while tools such as the Firefox HTTP Header inspection tool and the browser developer panel supported observation of outgoing requests. This setup provided a controlled environment to simulate attacker behavior and defensive mechanisms without modifying the host system.

### A. HTTP Headers

HTTP headers played a key role in analyzing forged requests and browser behavior throughout this lab. Using the Firefox HTTP Header inspection tool, the exact GET and POST requests generated by both legitimate actions and malicious payloads could be observed. This visibility revealed critical fields such as cookies, CSRF tokens, timestamps, request parameters, and content types.

### B. Task 1 & Task 2 : Displaying a Message and Cookies

To begin, a simple JavaScript payload was injected into the user profile to confirm that inline scripts executed when another user viewed the page.

```
<script>alert('XSS');</script>
```

As shown in Figure 1: XSS Alert Triggered After Login, the alert window verified successful code execution. Next, the payload was modified to display the user's cookies using `document.cookie`, demonstrating how an attacker can access sensitive session data. The output displayed in the alert confirmed that cookies were accessible through the injected script.

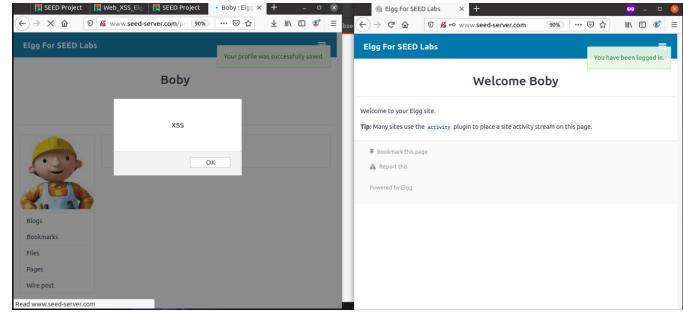


Fig. 1: XSS alert message after login

### C. Task 3 & Task 4 : Stealing Cookies and Becoming a Friend

In the next phase, the JavaScript payload was extended to exfiltrate cookies by creating a fake image request, sending them to the attacker's netcat listener. Figure 2 captured GET Request Containing Cookie Parameter shows how the victim's browser unknowingly transmitted data to the attacker using netcat. The attack then shifted to forging HTTP requests, where an Ajax GET request was constructed to automatically add Samy as a friend when a victim viewed his profile.

```
[11/09/25] user@VM-.../LabSetup$ nc -lknv 5555
Listening on 0.0.0.0 5555
Connection received on 192.168.122.228 43530
GET /?c=Elgg%3D!scktb0rmc9t6ftcl760ecogpv HTTP/1.1
Host: 10.9.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: image/webp, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.seed-server.com/profile/alice
```

Fig. 2: Victim's Cookie transmitted to attacker

Figure 3: Add-Friend HTTP Request Captured in Developer Tools illustrates how the forged request mimicked a legitimate friend-addition action.

- 1) Question 1: Explain the purpose of Lines 1 and 2, why are they needed?

Lines 1 and 2 extract the Elgg timestamp (`__elgg_ts`) and CSRF security token (`__elgg_token`). These values are required because Elgg validates them on every state-changing request; without them the forged request would be rejected as unauthorized.

- 2) Question 2: If the Elgg application only provide the Editor mode for the "About Me" field, i.e., you cannot switch to the Text mode, can you still launch a successful attack?

No, the attack would not work if only the "Editor" mode were available. Editor mode escapes HTML tags and prevents raw JavaScript execution, meaning `<script>` tags would not run. Attackers require Text mode to insert unmodified JavaScript code.

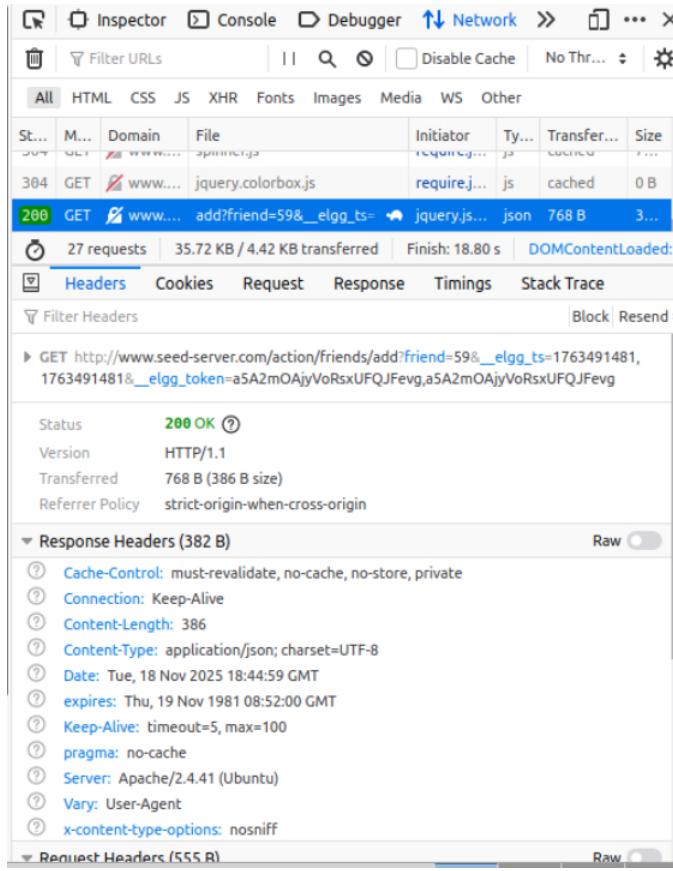


Fig. 3: Add-friend inspection in GET request Network tab

#### D. Task 5 : Modifying the Victim's Profile

The attack escalated by forging a POST request that modified the victim's profile, specifically replacing their "About Me" field. Using parameters captured from a legitimate profile update, a malicious request was constructed and automatically sent from the victim's browser. Figure 4: Victim Profile Showing 'You Have Been Hacked by Samy!' demonstrates the successful overwrite, confirming full control over the victim's profile data.

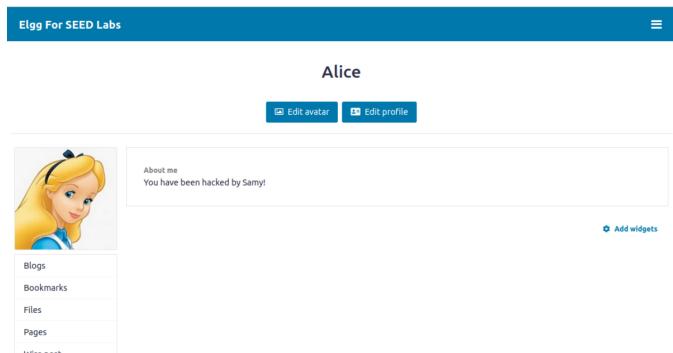


Fig. 4: Alice profile modified ("You have been hacked by Samy!")

- 1) Question 3: Why do we need Line 1? Remove this line, and repeat your attack. Report and explain your

observation.

Line 1 prevents the worm from modifying Samy's own profile by checking whether the currently logged-in user is Samy. Removing this line causes the payload to overwrite Samy's profile with the attack text, breaking the worm and interrupting self-propagation because the original malicious script would be destroyed.

#### E. Task 6 : Writing a Self-Propagating XSS Worm

The final offensive task combined all previous steps to construct a self-propagating worm. Using DOM extraction, the script copied its own contents, URL-encoded them, and injected the worm into the victim's profile along with the forged friend-request code and profile-modification request. Figure 5: Automatic Add-Friend and Edit Requests Triggered When Visiting Samy's Profile shows the worm's behavior upon profile access.

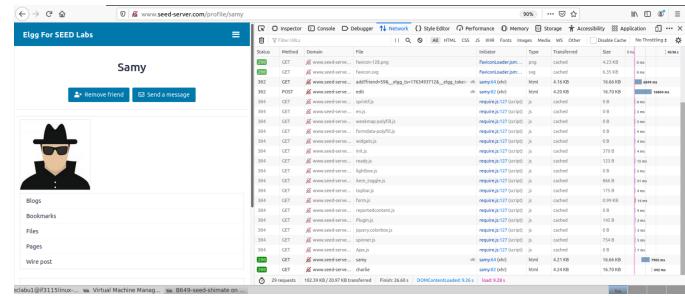


Fig. 5: Worm propagation GET requests (Charlie visiting Samy)

Figure 6 has Worm Propagation Showing 'Samy Owns You' in the Compromised Profile confirms successful spread as new victims became attackers.

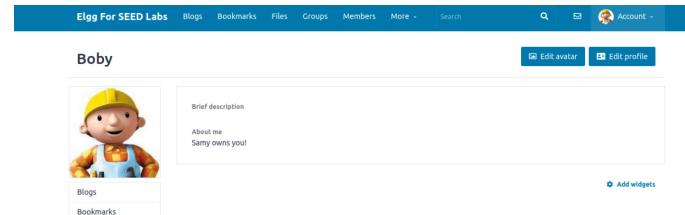


Fig. 6: Bob visiting Charlie propagation result ("Samy owns you")

#### F. Task 7 : Defeating XSS Using Content Security Policy (CSP)

The final section examined CSP behavior across three domains with different configurations. Figure 7: CSP Results Before Configuration Changes shows that example32a allowed all scripts, example32b blocked most inline and external scripts, and example32c selectively allowed scripts based on nonce values. After modifying the Apache configuration and PHP CSP header, Figure 8: CSP Results After Configuration Changes shows that required script blocks were permitted according to the task instructions.

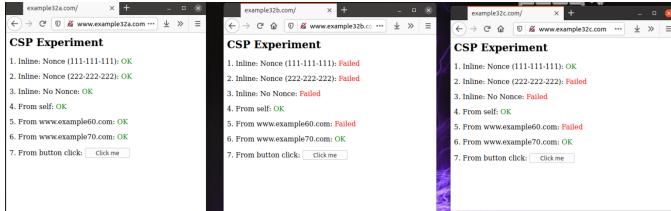


Fig. 7: CSP before screenshots for example32a/b/c

1) *Describe and explain your observations when you visit these websites:* Example32a executed all scripts because it had no CSP enabled. Example32b blocked inline scripts and scripts from example60.com due to its restrictive policy. Example32c permitted only the inline script with the correct nonce and scripts from trusted domains specified in its CSP header.

2) *Click the button in the web pages from all the three websites, describe and explain your observations:* In example32a, clicking the button executed the inline JavaScript immediately. In example32b and example32c, the click event failed because inline event handlers were not allowed without explicit nonce authorization.

3) *Change the server configuration on example32b (modify the Apache configuration), so Areas 5 and 6 display OK. Please include your modified configuration in the lab report.:*  Modifying Apache config for example32b at /etc/apache2/sites-available/apache\_csp.conf

```
script-src 'self'
    www.example60.com
    *.example70.com
```

4) *Change the server configuration on example32c (modify the PHP code), so Areas 1, 2, 4, 5, and 6 all display OK. Please include your modified configuration in the lab report:*  Modifying Apache config for example32c at /var/www/csp/phpindex.php

```
script-src 'self'
    'nonce-111-111-111'
    'nonce-222-222-222'
    www.example60.com
    *.example70.com
```

5) *Please explain why CSP can help prevent Cross-Site Scripting attacks:* CSP prevents XSS by restricting where JavaScript is allowed to come from, enforcing trust boundaries between code and data. Inline scripts and untrusted third-party sources are blocked unless explicitly authorized with whitelisted domains or nonce values. This ensures that injected scripts, such as those used in XSS attacks, cannot execute even if they are successfully placed in the HTML.

### III. CONCLUSION

This lab provided practical exposure to both the exploitation and mitigation of Cross-Site Scripting vulnerabilities. The offensive tasks illustrated how attackers steal data, forge authenticated actions, and create self-replicating worms using nothing more than injected JavaScript. The defensive CSP exercises showed how well-configured security policies can

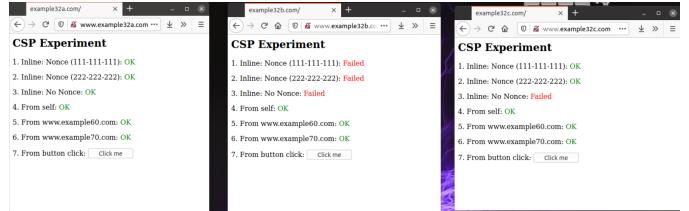


Fig. 8: CSP after screenshots for example32a/b/c

block such attacks at the browser level. Overall, the lab deepened the understanding of web application security and emphasized the importance of defense-in-depth against client-side threats.

### REFERENCES

- [1] W. Du, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition. Available at: <https://www.handsongsecurity.net/>.
- [2] SEED Labs, Cross-Site Scripting Attack Lab (Elgg), [https://seedsecuritylabs.org/Labs\\_20.04/Web/Web\\_XSS\\_Elgg/](https://seedsecuritylabs.org/Labs_20.04/Web/Web_XSS_Elgg/).
- [3] Mozilla Developer Network, Content Security Policy (CSP), <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>.
- [4] World Wide Web Consortium (W3C), Content Security Policy Level 3, <https://www.w3.org/TR/CSP3/>.
- [5] OWASP Foundation, Cross-Site Scripting (XSS), <https://owasp.org/www-community/attacks/xss/>.
- [6] OWASP Foundation, Cross-Site Request Forgery (CSRF), <https://owasp.org/www-community/attacks/csrf/>.
- [7] Elgg Project, Elgg Developer Documentation, <https://learn.elgg.org/en/stable/>.
- [8] S. Kamkar, The Samy Worm: How I Hacked MySpace, <https://samy.pl/myspace/>.
- [9] GNU Netcat Documentation, Netcat User Guide, <http://nc110.sourceforge.net/>.