Computer System Security

Lab Assignment 5 Report

Team 16

Sonam Ghatode

Arjun Katneni

12.04.2019

Index

Cross-Site Request Forgery (CSRF):

Cross-Site Request Forgery (CSRF or XSRF), also known as one-click attack or session riding, is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the web application trusts. Ways to transfer such malicious commands include specially-crafted image tags, hidden forms, and JavaScript XMLHttpRequests, for example, can all work without the user's interaction or even knowledge. Unlike cross-site scripting (XSS), which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has in a user's browser.

CSRF is an attack that tricks the victim into submitting a malicious request. It inherits the identity and privileges of the victim to perform an undesired function on the victim's behalf. For most sites, browser requests automatically include any credentials associated with the site, such as the user's session cookie, IP address, Windows domain credentials, and so forth. Therefore, if the user is currently authenticated to the site, the site will have no way to distinguish between the forged request sent by the victim and a legitimate request sent by the victim.

CSRF attacks target functionality that causes a state change on the server, such as changing the victim's email address or password, or purchasing something. Forcing the victim to retrieve data doesn't benefit an attacker because the attacker doesn't receive the response, the victim does. As such, CSRF attacks target state-changing requests.

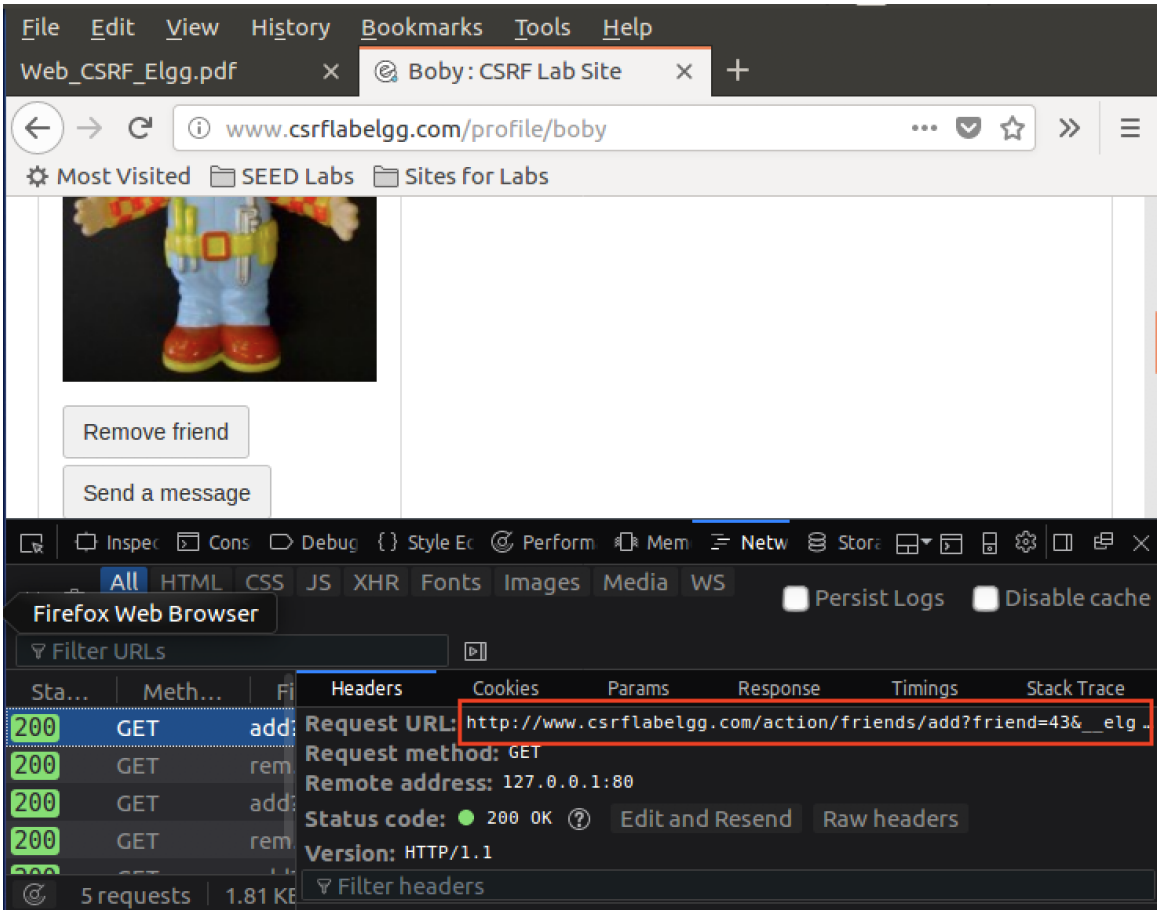
It's sometimes possible to store the CSRF attack on the vulnerable site itself. Such vulnerabilities are called ***stored CSRF flaws***. This can be accomplished by simply storing an ***img*** or ***iframe*** tag in a field that accepts HTML, or by a more complex cross-site scripting attack. If the attack can store a CSRF attack on the site, the severity of the attack is amplified. In particular, the likelihood is increased because the victim is more likely to view the page containing the attack than some random page on the Internet. The likelihood is also increased because the victim is sure to be authenticated to the site already.

CSRF Attack using GET Request:

In this task we have to create ***http://www.csrflabattacker.com*** in such a way that if Alice clicks on the link, it adds ***Boby*** as her friend on Elgg. To do that, we figured out where site details is stored by having a loot at ***/etc/apache2/sites-available/00-default.conf***:

```
<VirtualHost *:80>
        ServerName http://www.csrflabelgg.com
        DocumentRoot /var/www/CSRF/Elgg
</VirtualHost>
<VirtualHost *:80>
        ServerName http://www.csrflabattacker.com
        DocumentRoot /var/www/CSRF/Attacker
</VirtualHost>
```

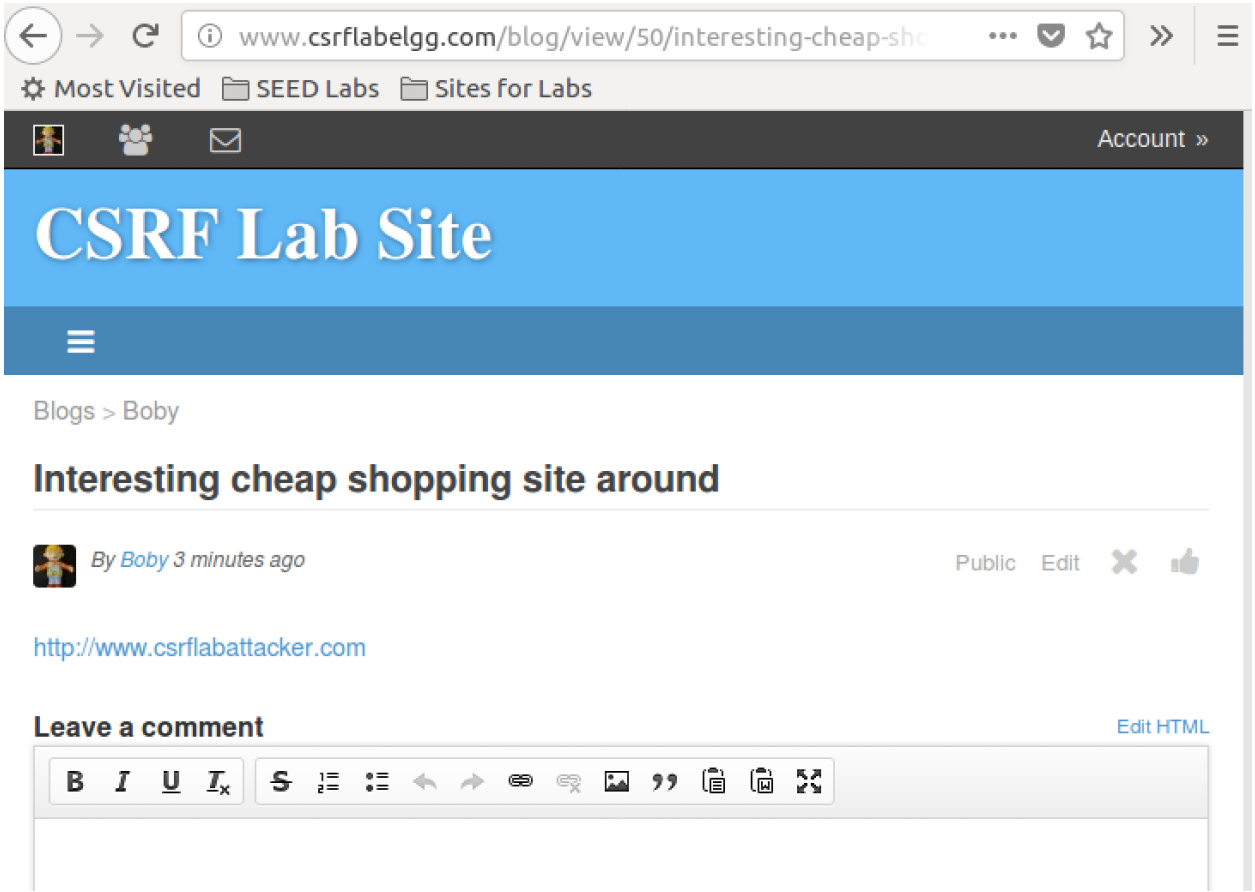
We created a file ***index.html*** at path ***/var/www/CSRF/Attacker/*** to make it the default loading page whenever someone clicks on ***http://www.csrflabattacker.com.*** To launch the CSRF attack, we first figured out what is sent when ***Alice*** tries to add ***Boby*** as friend. When ***Alice*** clicks on ***Add Friend*** button, following requests are sent:



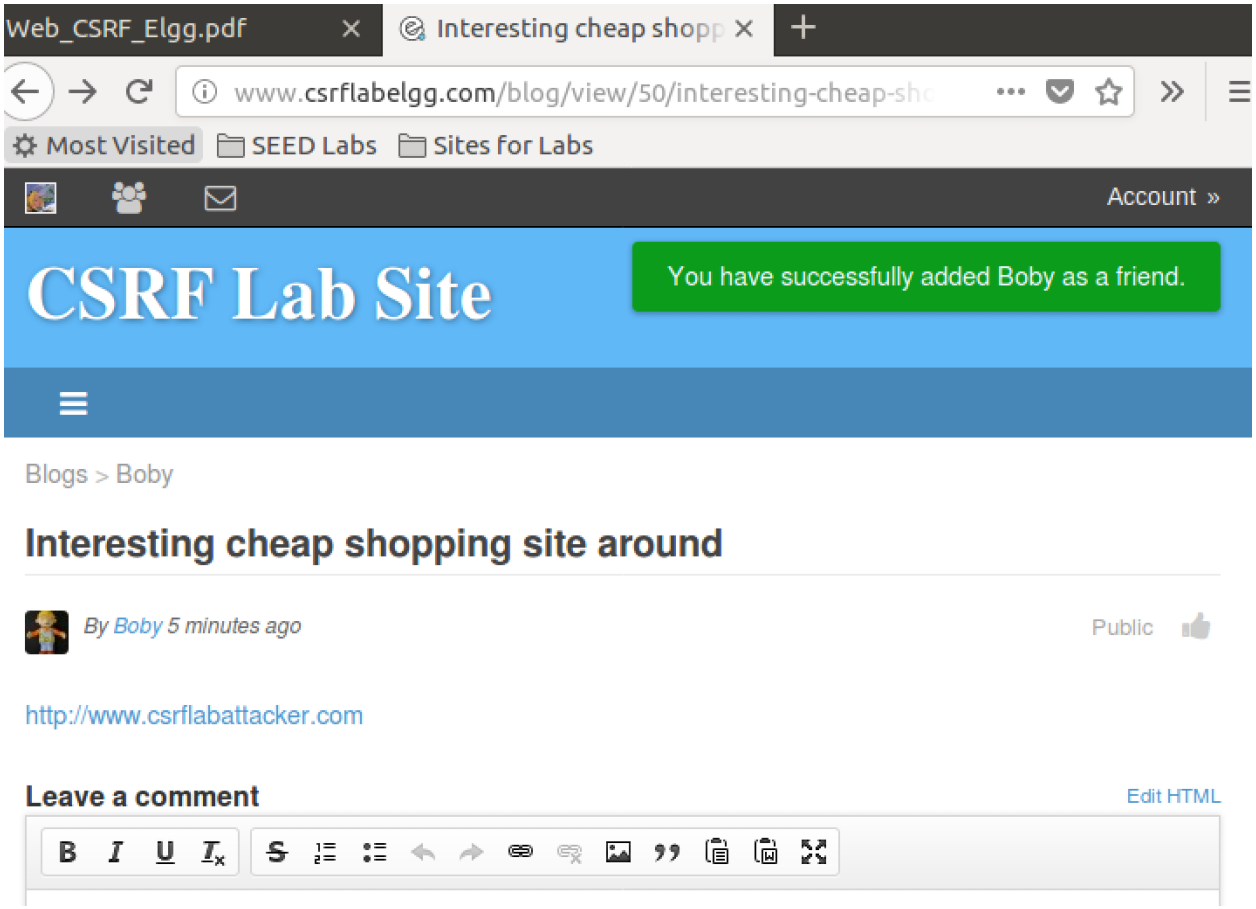

```
http://www.csrflabelgg.com/action/friends/add?
friend=43&__elgg_ts=1575656357&__elgg_token=-
VIADcxLInvK9wB0WhT_7g&__elgg_ts=1575656357&__elgg_token=-VIADcxLInvK9wB0WhT_7g
```

This implies ***guid*** for ***Boby*** is ***43*** and the ***GET*** request sent to add ***Boby*** as friend includes his ***guid***. So we edited ***index.html*** of ***http://www.csrflabattacker.com*** as following:

And then we posted a blog from ***Boby's*** profile with link in its description:

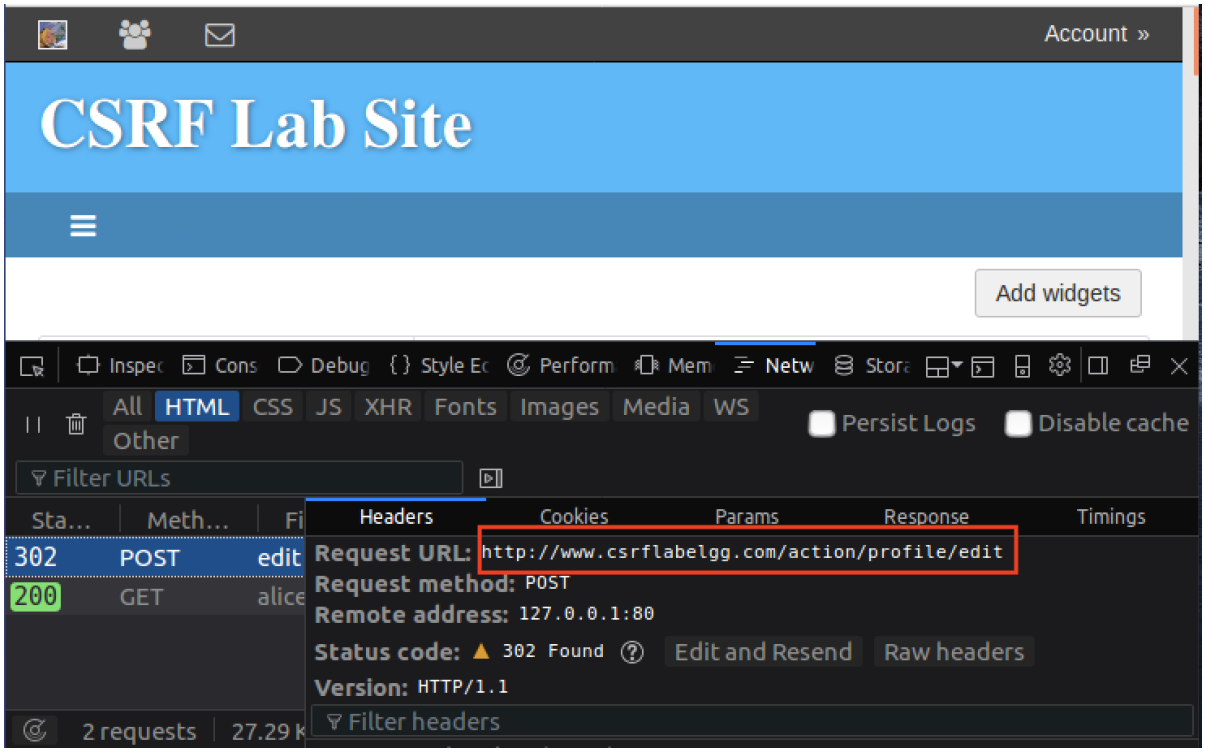


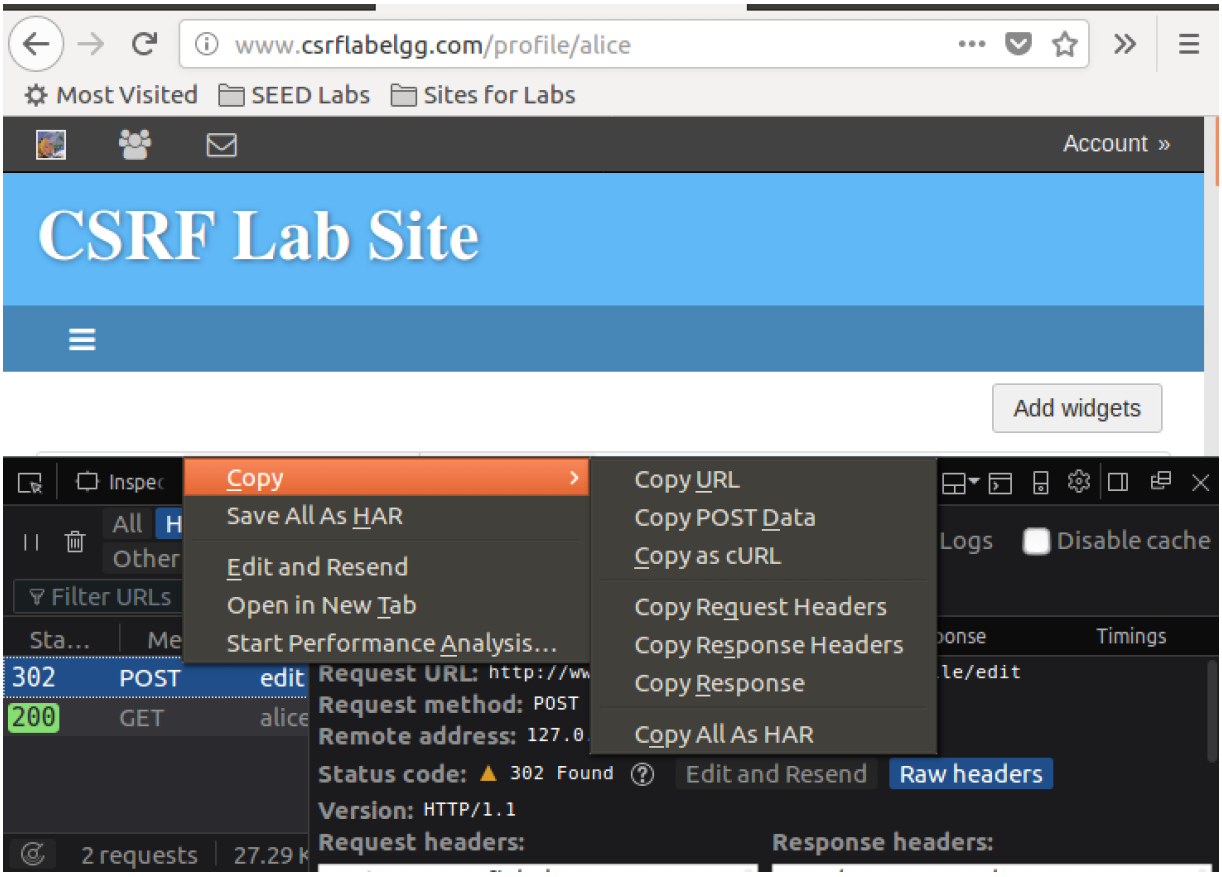
The link will redirect ***Alice*** to ***http://www.csrflabattacker.com*** which has an image tag, where the source of the image is the URL that is called when ***Alice*** wants to add ***Boby*** as friend. We logged in from ***Alice's*** account and clicked the link specified in the blog and following was the observation:

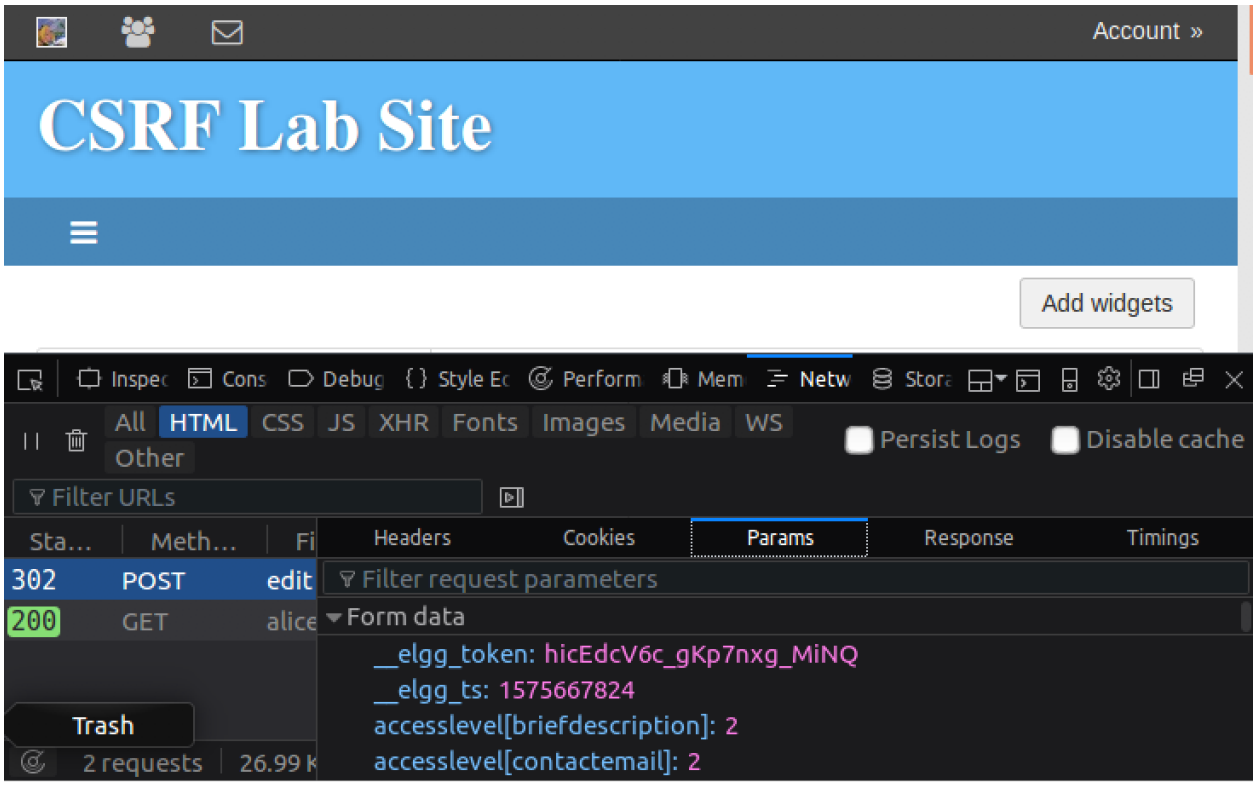


CSRF Attack using POST Request:

In this task, we had to edit ***Alice's*** profile in such a way that it has ***Boby is my hero*** in description. For that, we first monitored how editing the profile request works. We used ***Firefox's Network Inspection Tool*** to monitor how the ***POST*** request is formed and what are the ***parameters*** sent while requesting a profile edit. Following are the screenshots for the same:


Account »
CSRF Lab Site
Add widgets
All HTML CSS JS XHR Fonts Images Media WS Other
Persist Logs
Disable cache
Filter URLs
Headers Cookies Params Response Timings
302 POST edit
200 GET alice
Request URL: http://www.csrflabelgg.com/action/profile/edit
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
2 requests
27.29 K


www.csrflabelgg.com/profile/alice
Most Visited
SEED Labs
Sites for Labs
Account »
CSRF Lab Site
Add widgets
Copy
Save All As HAR
Edit and Resend
Open in New Tab
Start Performance Analysis...
Copy URL
Copy POST Data
Copy as cURL
Copy Request Headers
Copy Response Headers
Copy Response
Copy All As HAR
302 POST edit
200 GET alice
Request method: POST
Status code: 302 Found
Edit and Resend
Raw headers
Version: HTTP/1.1
Request headers:
Response headers:
2 requests
27.29 K



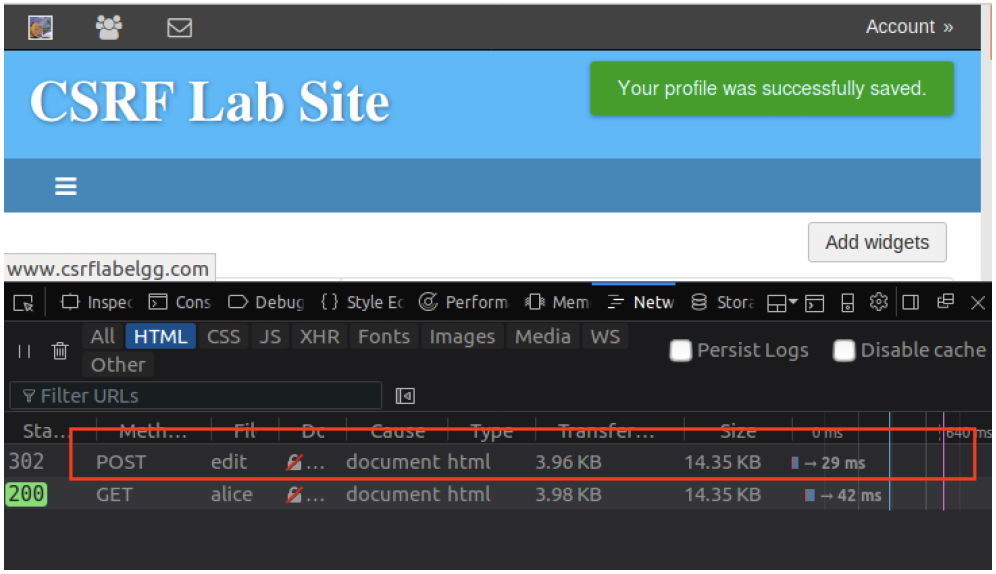
Following are the ***parameters*** sent with the ***POST*** request:

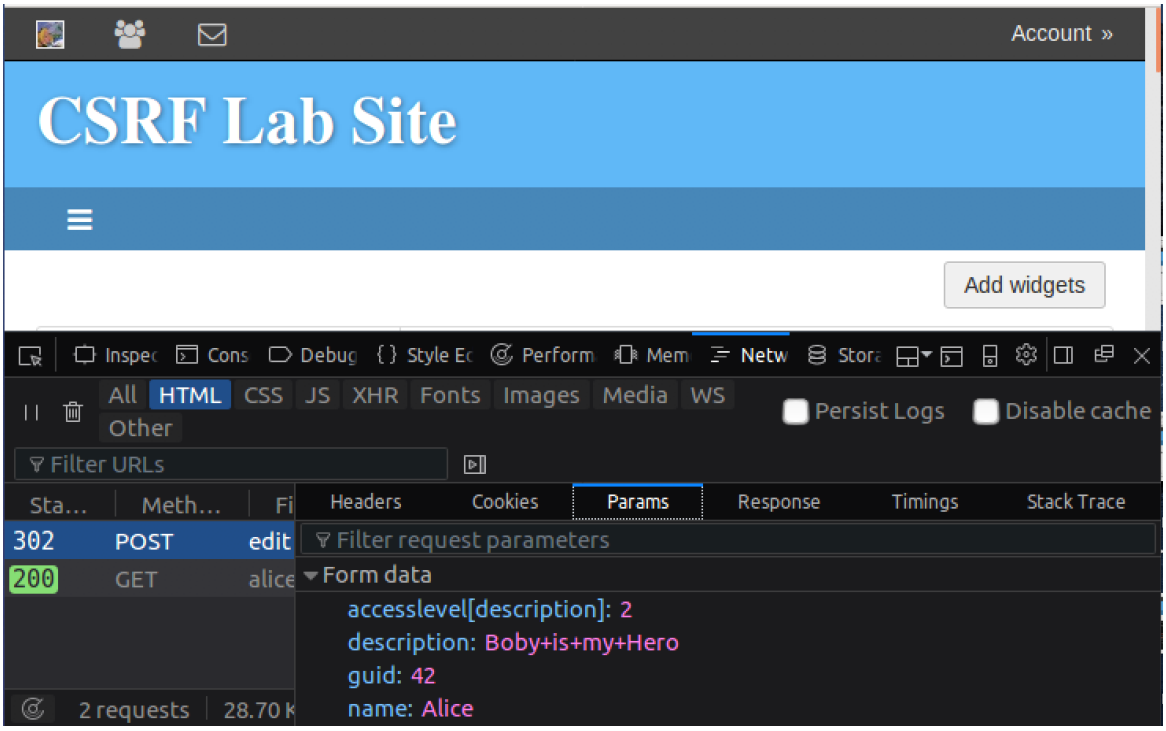
```
__elgg_token=a1y5b07jIkaiDqQiYmFqIQ
__elgg_ts=1575657958
name=Alice
description=<p>Boby+is+my+Hero</p>

accesslevel[description]=2
briefdescription
accesslevel[briefdescription]=2
location
accesslevel[location]=2
interests
accesslevel[interests]=2
skills
accesslevel[skills]=2
contactemail
accesslevel[contactemail]=2
phone
accesslevel[phone]=2
mobile
accesslevel[mobile]=2
website
accesslevel[website]=2
twitter
accesslevel[twitter]=2
guid=42
```

After figuring out how the post request is sent and what are the parameters sent with it that defines an user i.e. ***Alice's guid***, we edited ***index.html*** of ***http://www.csrflabattacker.com*** as following to carry out CSRF using ***POST*** method:

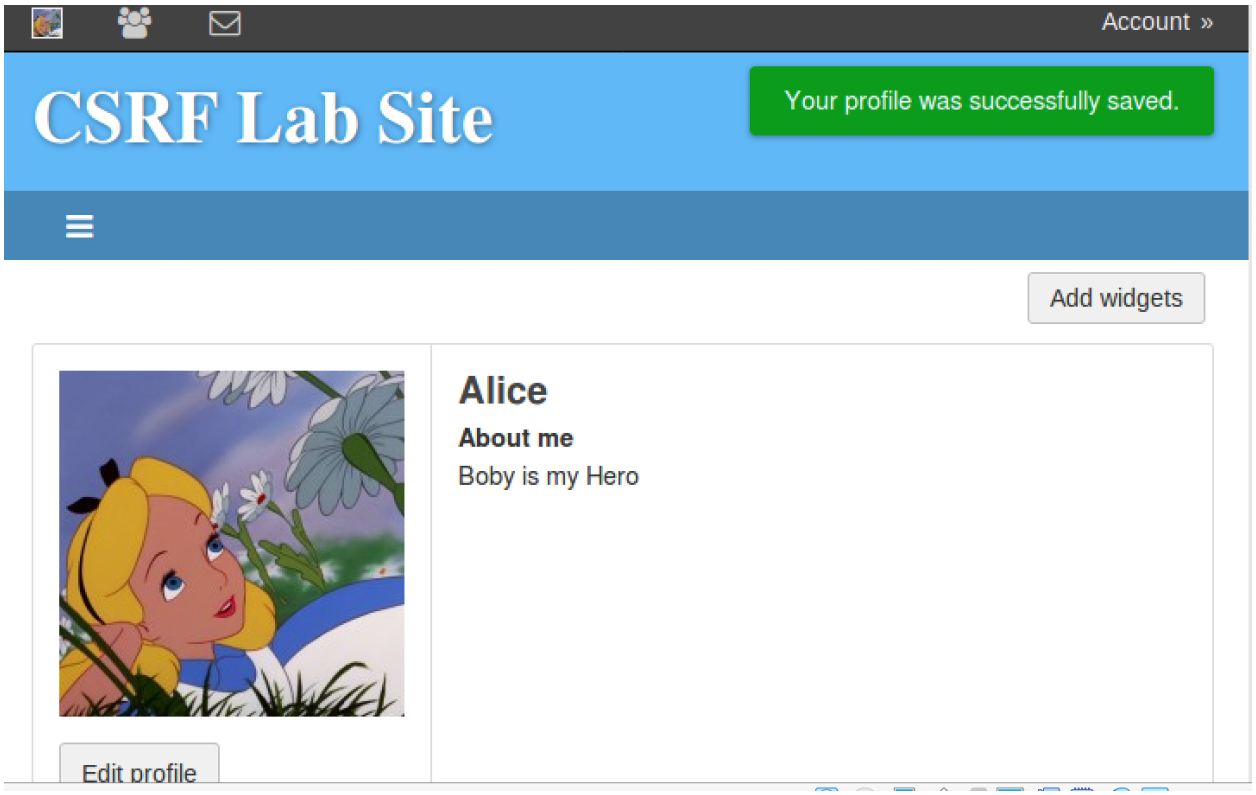
```
<meta charset="utf-8"/>
<html>
<body>
<h1>Shopping Site</h1>
<script type="text/javascript">
function forge_post()
{
var fields;
fields += "<input type='hidden' name='name' value='Alic
e'/>";
fields += "<input type='hidden' name='description' valu
e='Boby is my Hero'/>";
fields += "<input type='hidden' name='accesslevel[descr
iption]' value='2'/>";
fields += "<input type='hidden' name='guid' value='42'/
>";
var p = document.createElement("form");
p.action = "http://www.csrflabelgg.com/action/profile/e
dit";
p.innerHTML = fields;
p.method = "post";
                                        9,1           Top
```

After editing the ***index.html***, we logged in using ***Alice's*** account and checked whether the exploits works or not. After logging in using ***Alice's*** account, we clicked on the link shared in the blog posted by ***Boby***. As soon as ***Alice*** clicks on the link, she's redirected to the ***http://www.csrflabattacker.com*** and from there a ***POST*** request is sent:



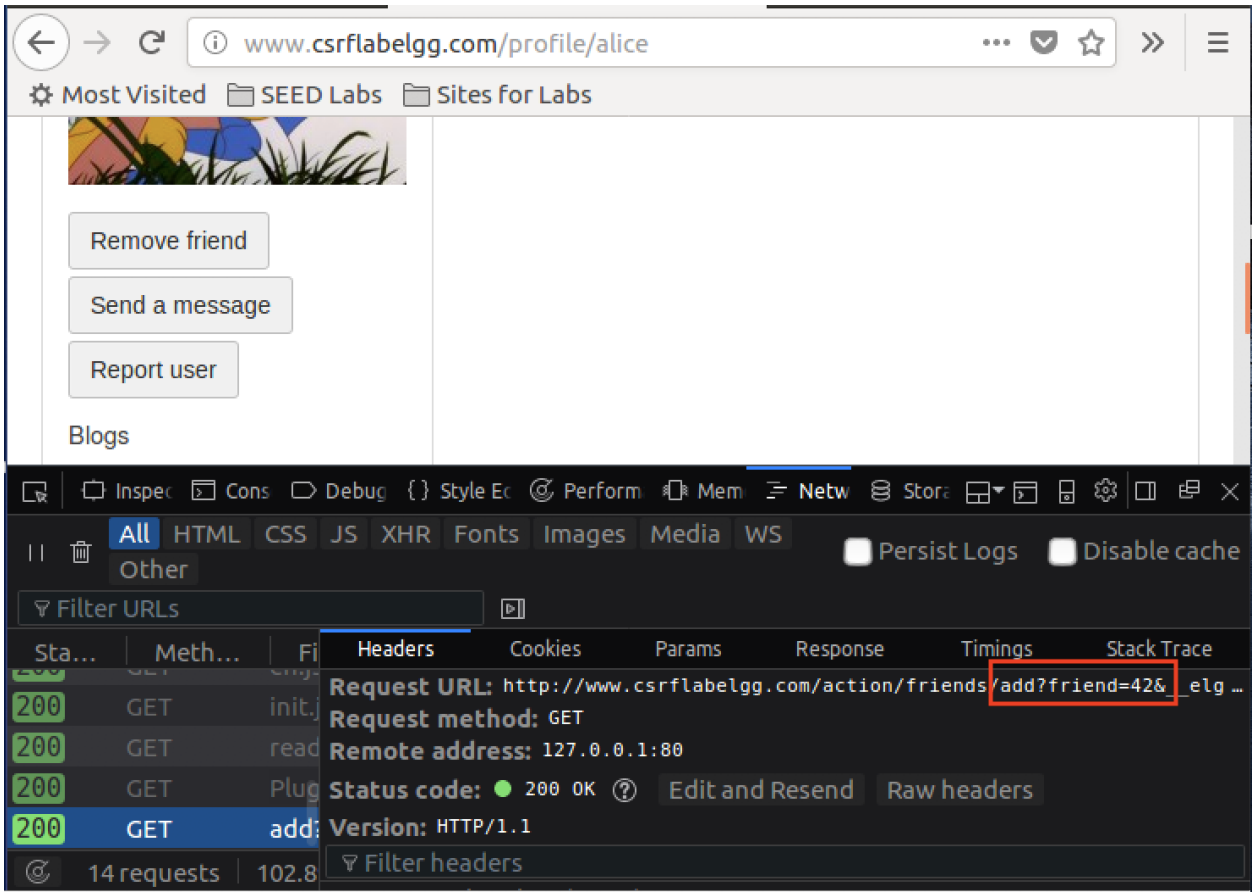


After the ***POST*** request becomes successful, following is the observation in ***Alice's*** profile:



Answer 1:

To get ***Alice's guid***, ***Boby*** can try to add ***Alice*** as friend and monitor the ***GET*** request sent over to get ***Alice's guid***:



In the screenshot above, it was clear that ***Alice's guid*** is ***42***. That way ***Boby*** could figure out her ***guid*** to launch targeted attack.

Answer 2:

If ***Boby*** wants to launch the attack on anyone who visits his website, he has to figure out a way to get ***guid*** of user whoever visits his profile. This could not be accomplished since ***guid*** is calculated beforehand and there is no way ***Boby*** could figure out how it is calculated or how it is fetched in the webpage. ***Boby*** can try but to figure out a way, but there is no way the internals of the website will be visible to him.

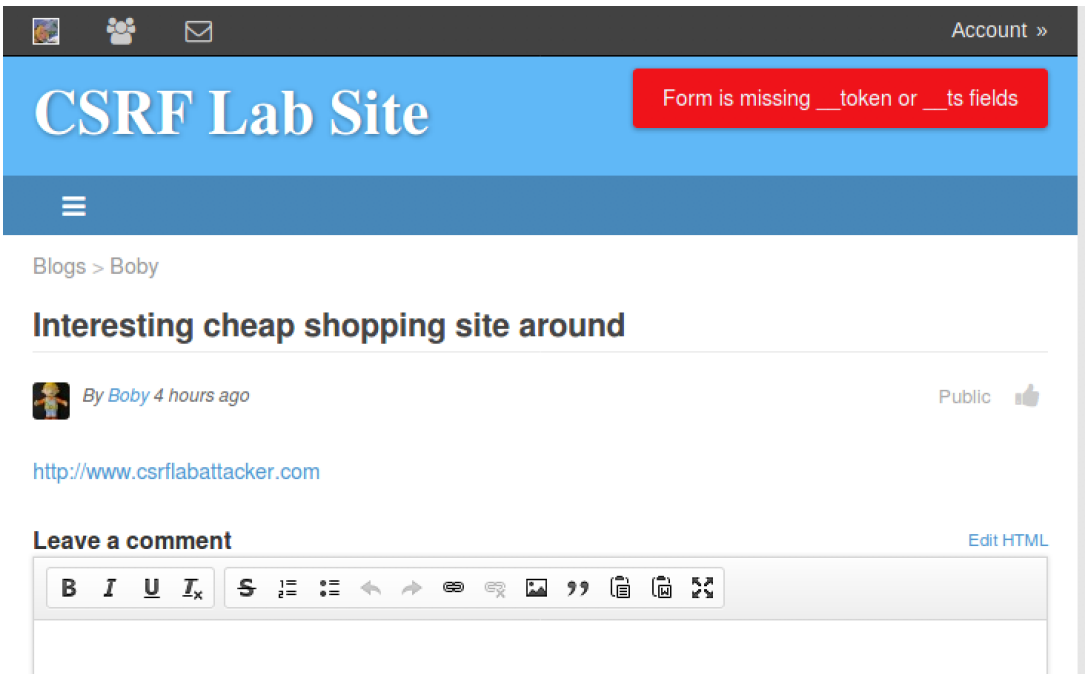
Implementing a countermeasure for Elgg:

As for the attack to work, in the ***Actionservice.php*** file of Elgg the steps to validate token and timestamp are omitted by giving ***return true;*** above all the code in the ***gatekeeper*** function. Hence we find the file in the Elgg directory and we modified the file to apply any counter measures.

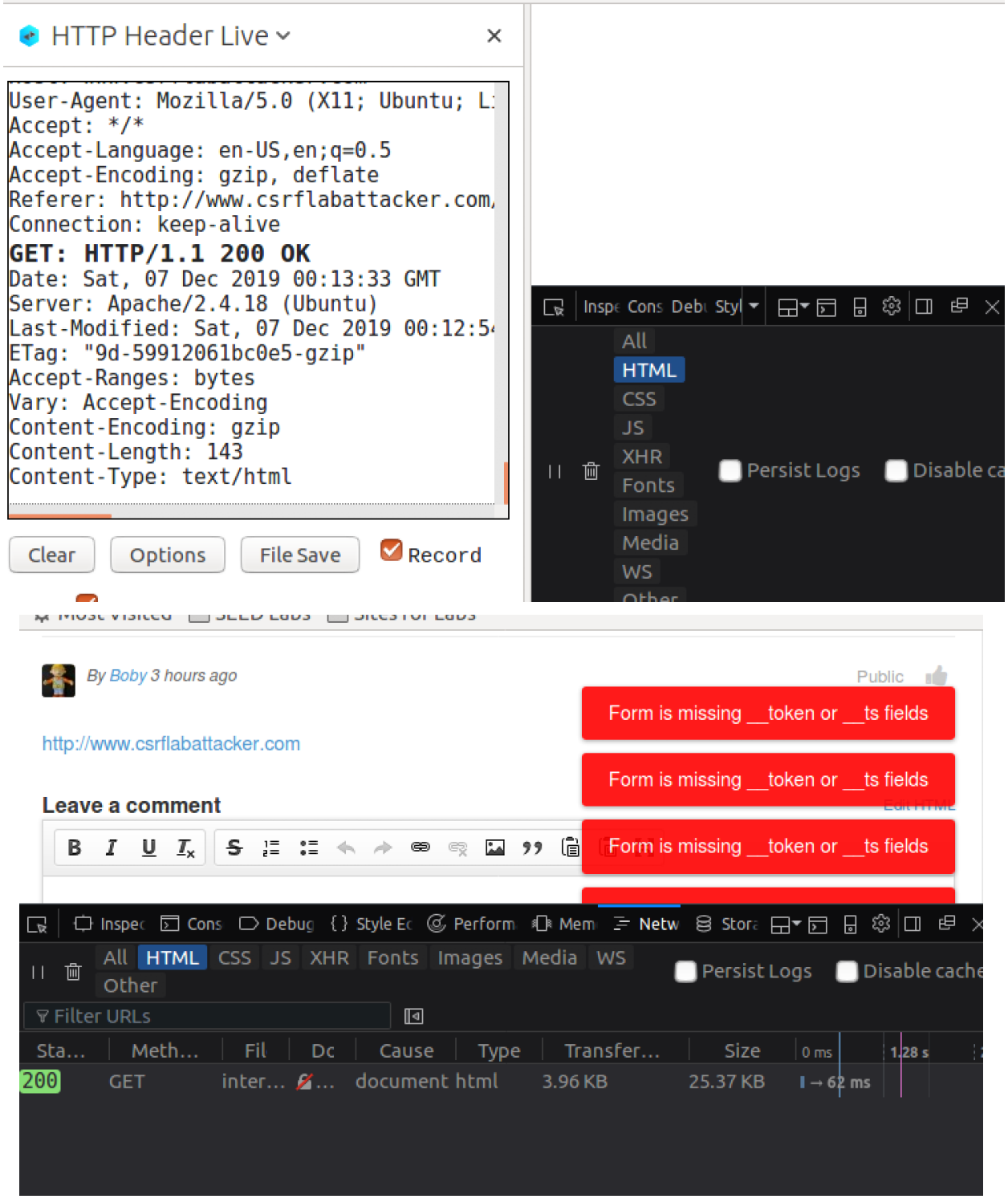
We commented the ***return true;*** statement from the code, resulting in the rest of the function code execution, which validates the time stamp and token as a result shows error message.

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

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

We can see the attack failing when ***Alice*** tried to access the malicious website and receives error message after returning to her page.



The error message clearly shows that ***__token*** or ***__ts*** fields are missing portraying that the ***gatekeeper*** function is executed properly and found the token and time stamp conflicting. Following is a screenshot showing Etag capture used to capture the ***token:***

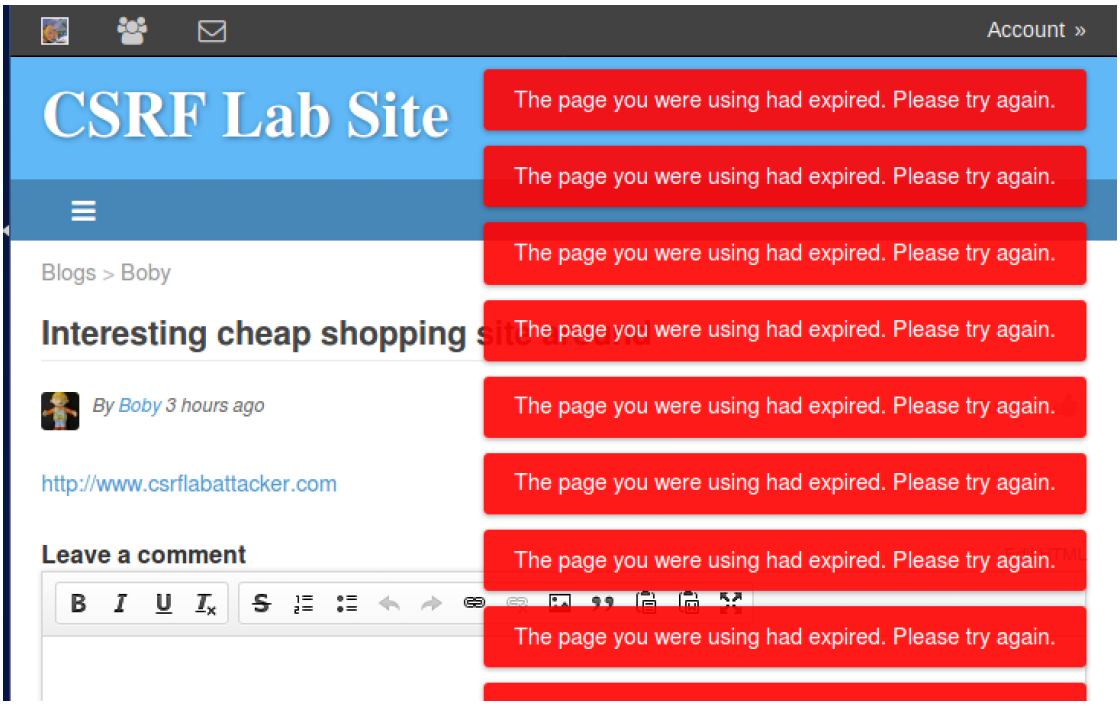


We tried to add the old ***timestamp*** and ***security token*** in the ***POST*** request parameters:

```
<meta charset="utf-8"/>
<html>
<body>
<h1>Shopping Site</h1>
<script type="text/javascript">
function forge_post()
{
var fields;
fields += "<input type='hidden' name='__elgg_ts' value=
'1575656357'/>";
fields += "<input type='hidden' name='__elgg_token' val
ue='-VIADcxLInvK9wB0WhT_7g'/>";
fields += "<input type='hidden' name='name' value='Alic
e'/>";
fields += "<input type='hidden' name='description' valu
e='Boby is my Hero'/>";
fields += "<input type='hidden' name='accesslevel[descr
iption]' value='2'/>";
fields += "<input type='hidden' name='guid' value='42'/
>";
var p = document.createElement("form");
<CSRF/Attacker/index.html" 25L, 822C 2,1          Top
```

System Settings

When we tried this attack on ***Alice***, following are the errors we got:



The screenshot above implies that these ***security tokens*** and ***timestamps*** are calculated randomly for each session and this random generation of ***timestamps*** and ***security tokens*** cannot be predicted by the attacker. These are generated randomly by a function that is called inside another function in the website code. Since we had access to the code of website, we were able to see how these are generated, but attackers would not have access to the code to launch the attack even when ***security token*** and ***timestamp*** matching is enabled.

References:

[1] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
[2] https://en.wikipedia.org/wiki/Cross-site_request_forgery