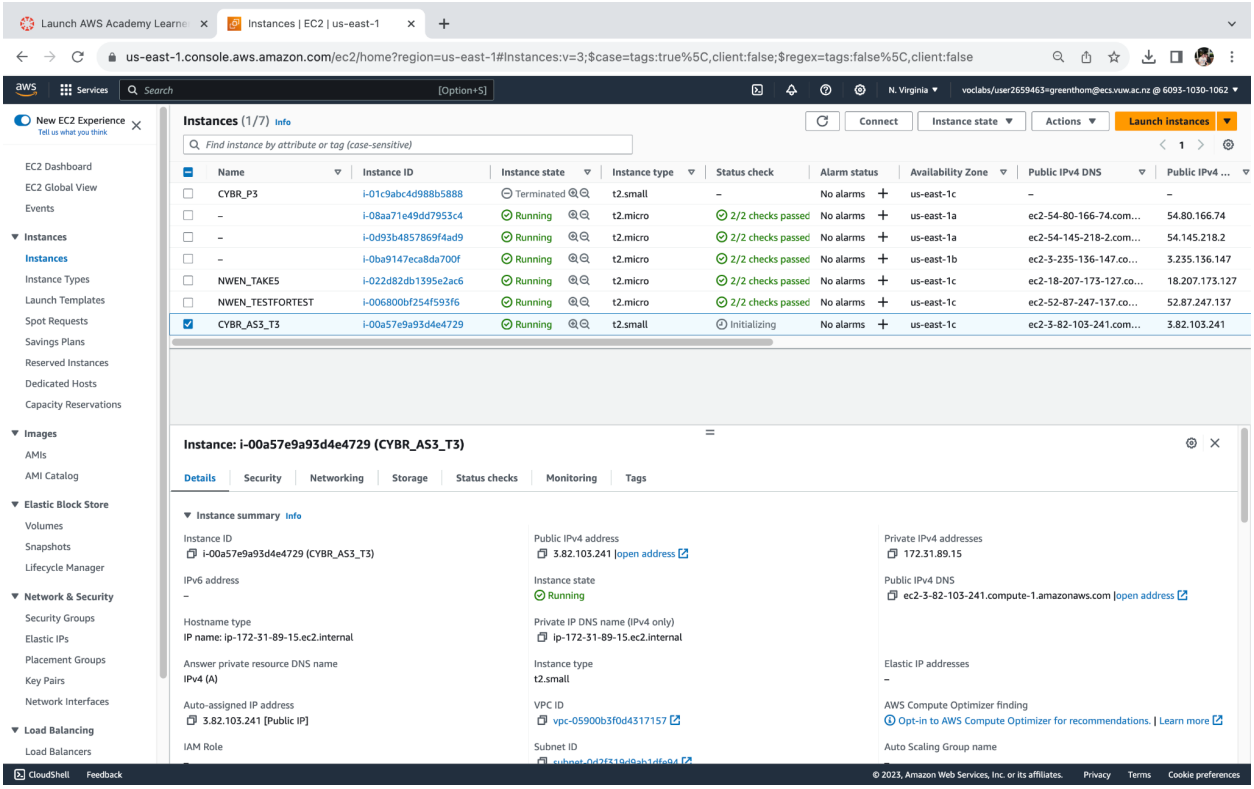
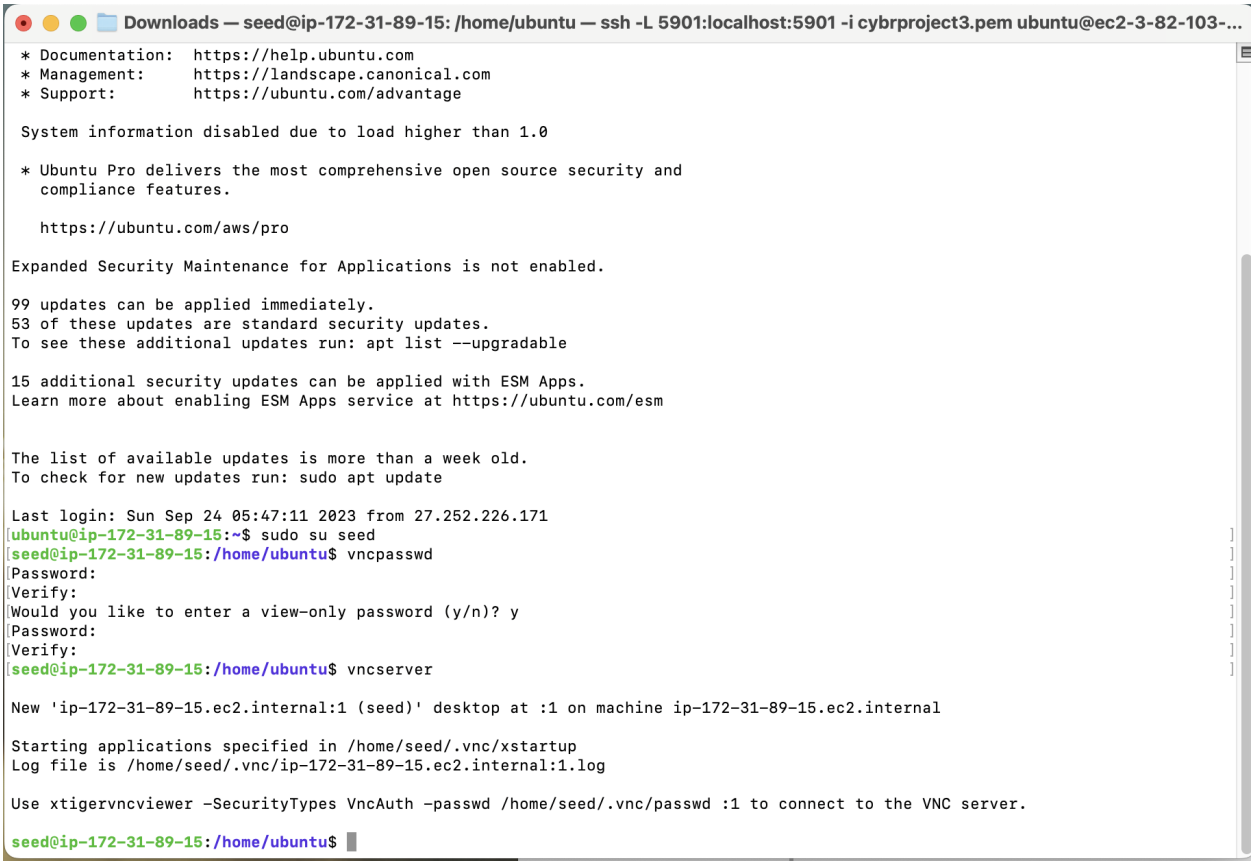
CYBR271 Assignment 3
greenthom - 300536064

Started instance using AMI provided.



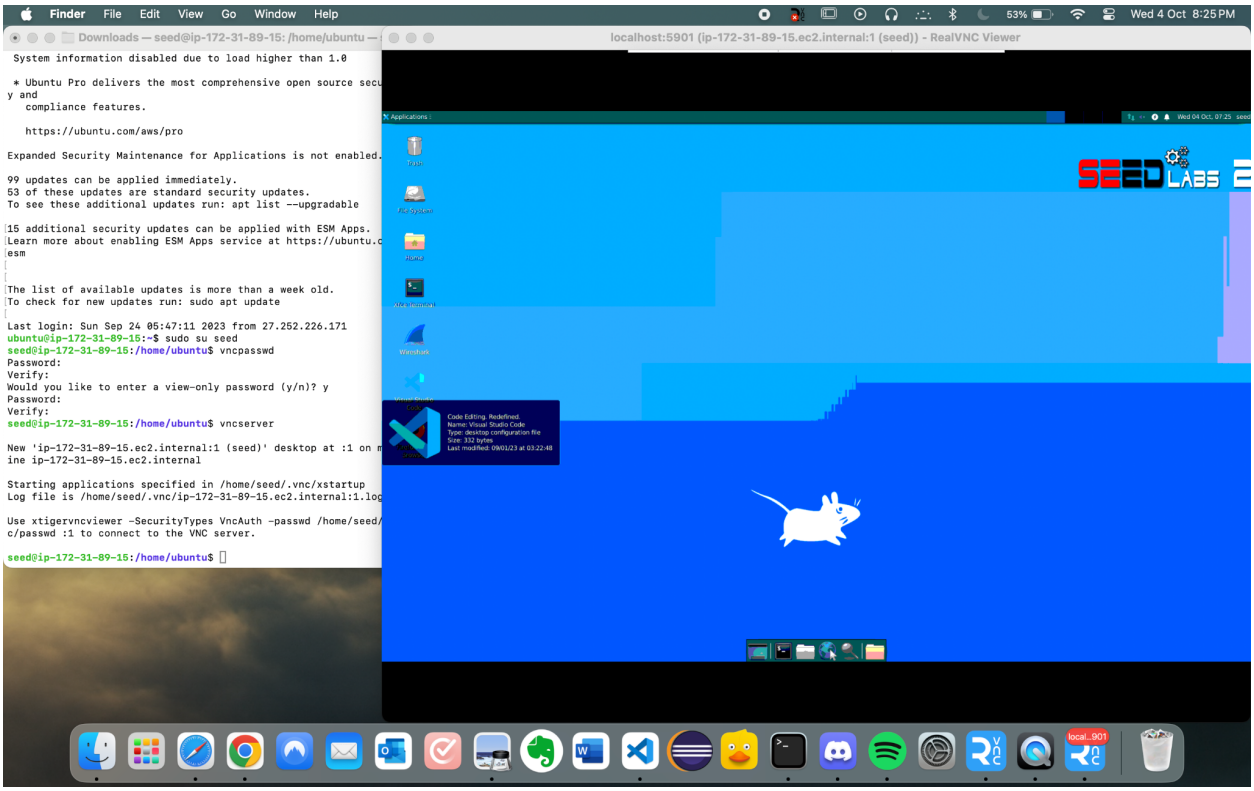
SSH into the instance using '`ssh -L 5901:localhost:5901 -i'` and my key.pem and the instance public IP.

Sign in to VNC using my username and password so I can then initiate it to be displayed on the VNC server using the command 'vncserver'

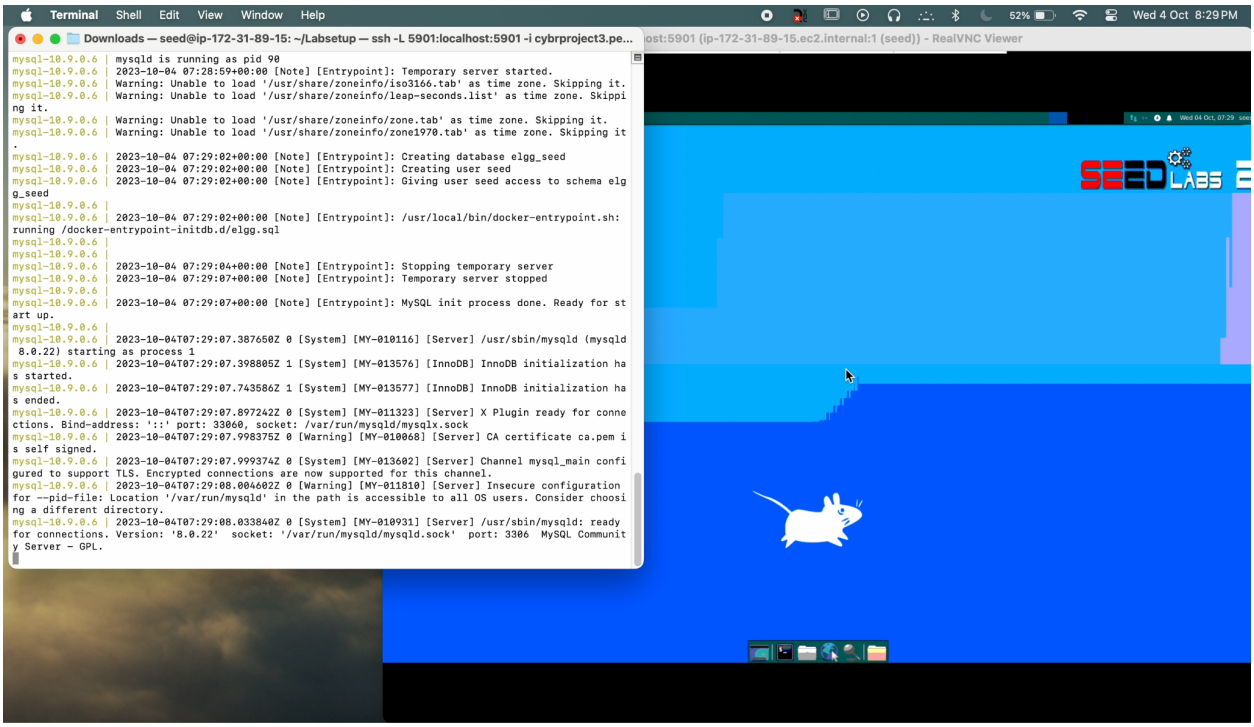


.

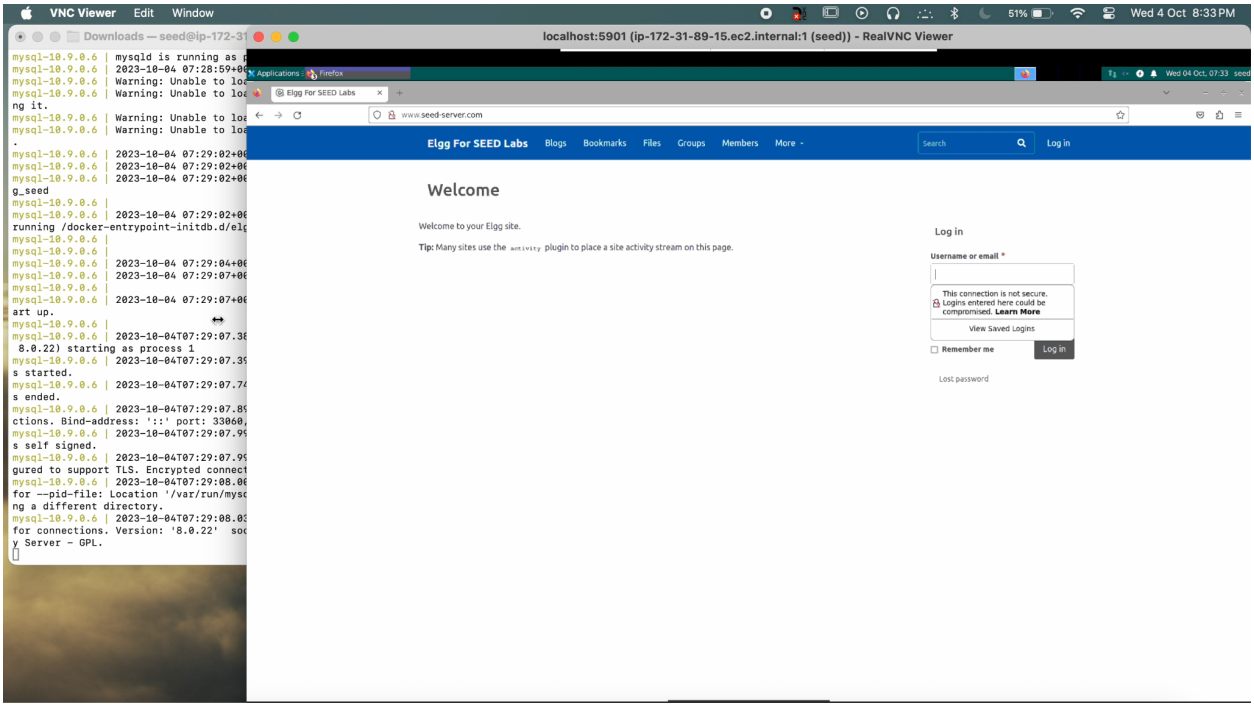
I opened up the VNC connecting to my lab by running' `localhost:5901.'`



Then run `dcup` to start the Elgg application from here; I can connect to `http://www.seed-server.com/` via my VNC viewer.

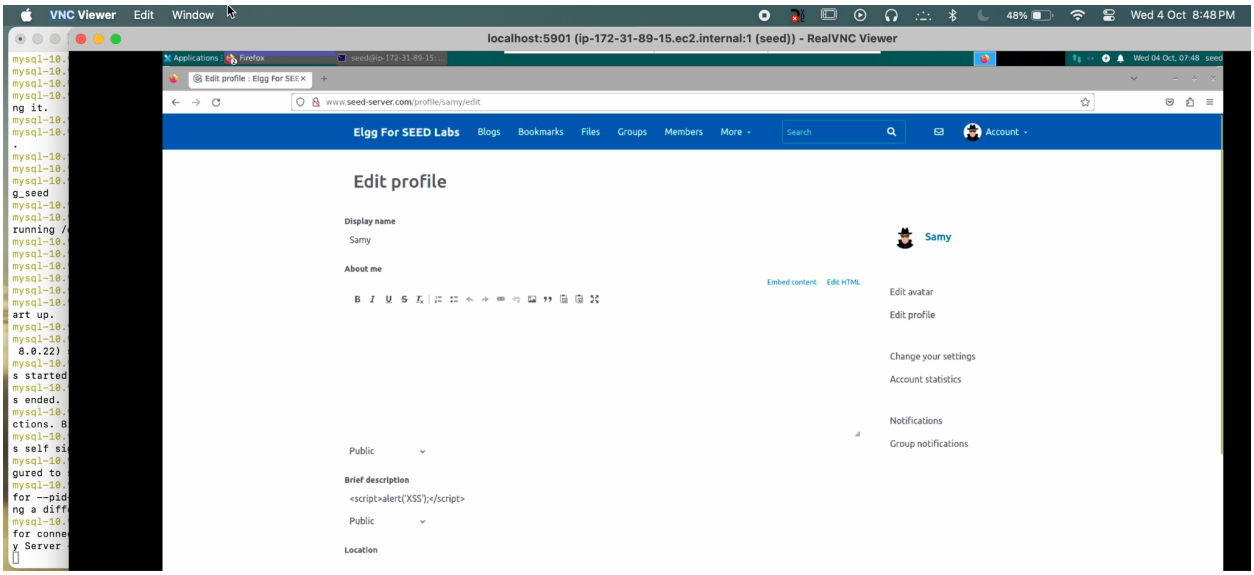


Here, I have successfully connected to `http://www.seed-server.com/` via my VNC

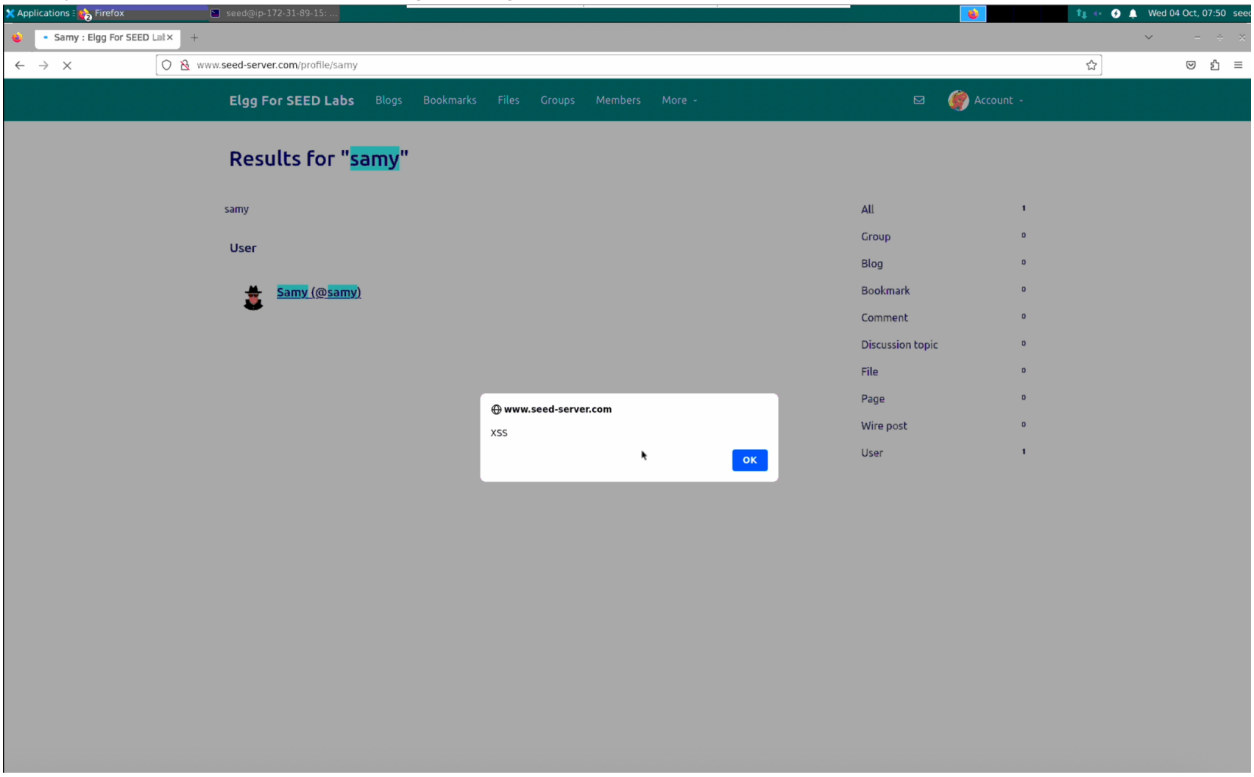


Task 1: Posting a Malicious Message to Display an Alert Window

This image shows us injecting JavaScript code into the 'brief description' of the profile Samy.

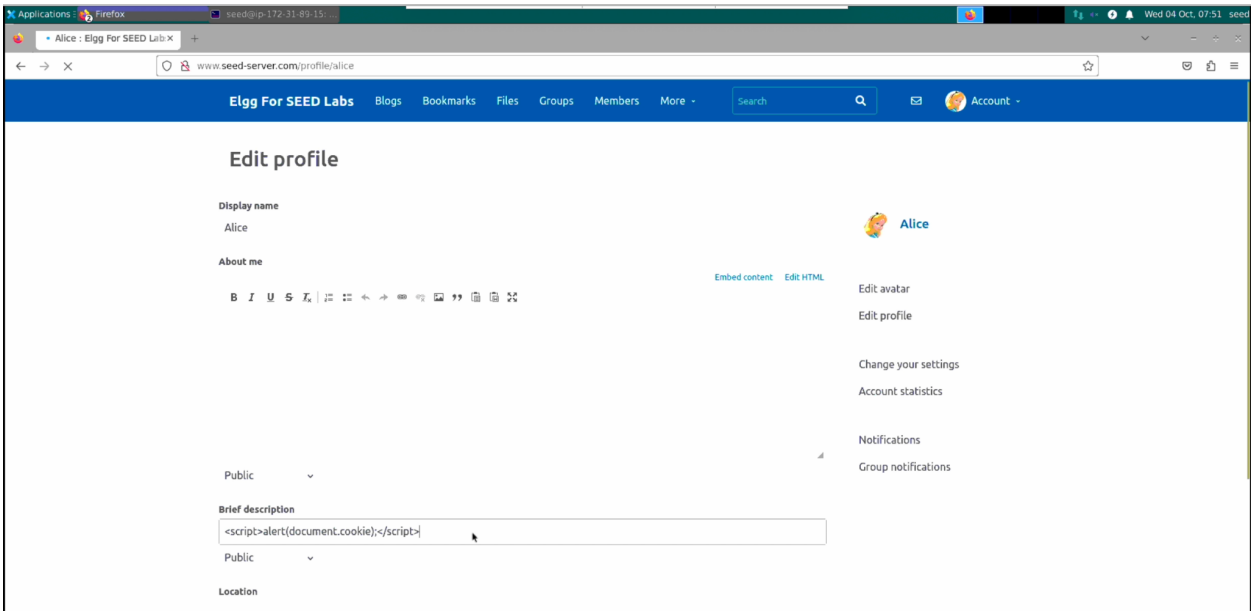


This shows that injecting this script triggers a popup on the user's window when you visit Samy's profile. Even though this is harmless, this can be further exploited as we now know that Samy's profile can make viewing through other profiles vulnerable.

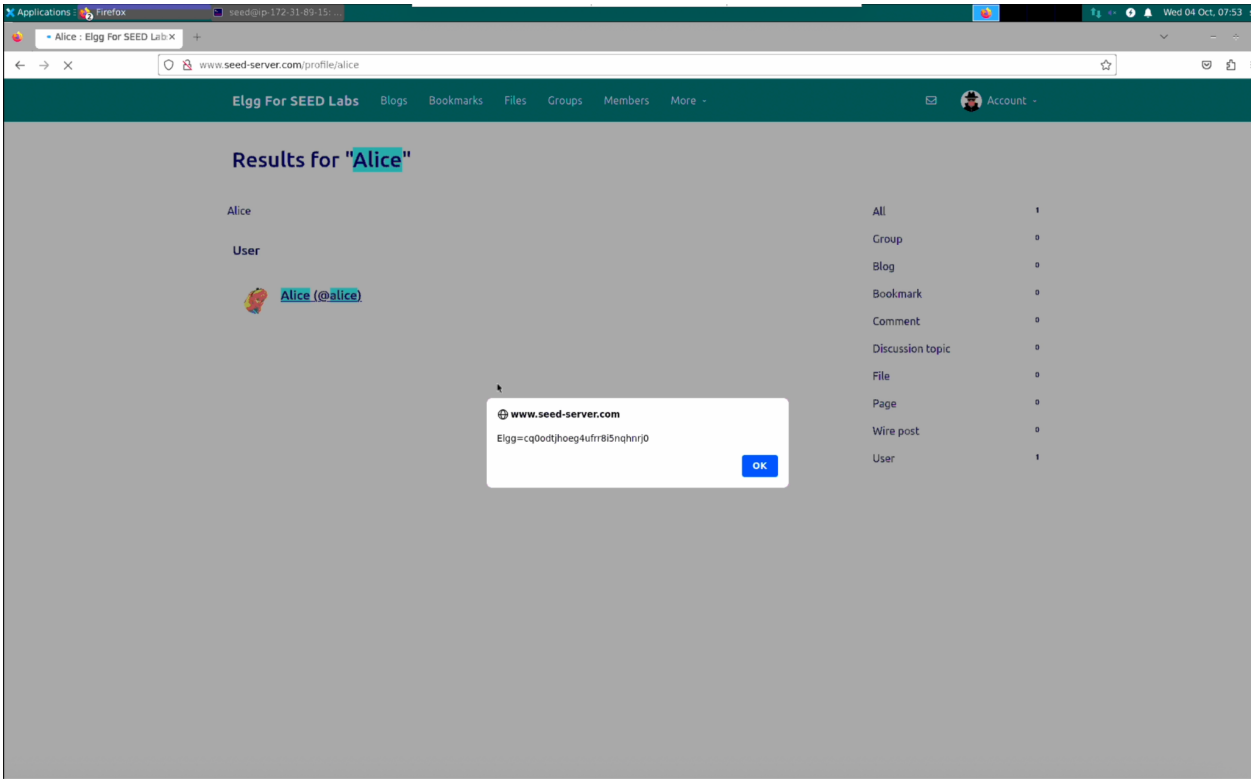


Task 2: Posting a Malicious Message to Display Cookies

Similar to the attack in task one. We are inserting JavaScript into a profile to display a popup. This time, we show the viewer's cookie to the exploited target. Here, we add this exploit script to the brief description of Alice's profile.

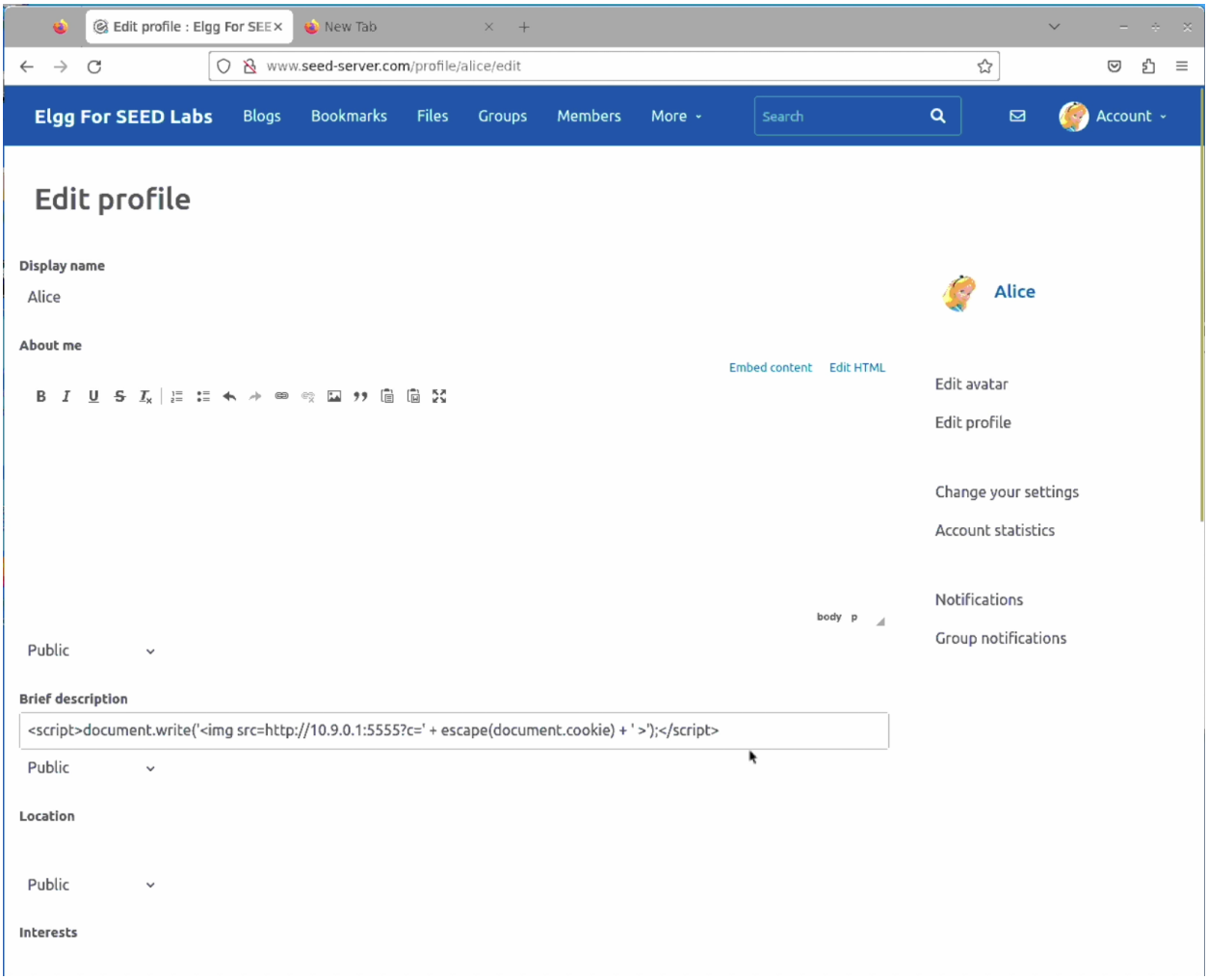


As we can see here, when we are logged in under Samy and visit Alice's profile, it presents a popup displaying the user's viewer sessions cookie

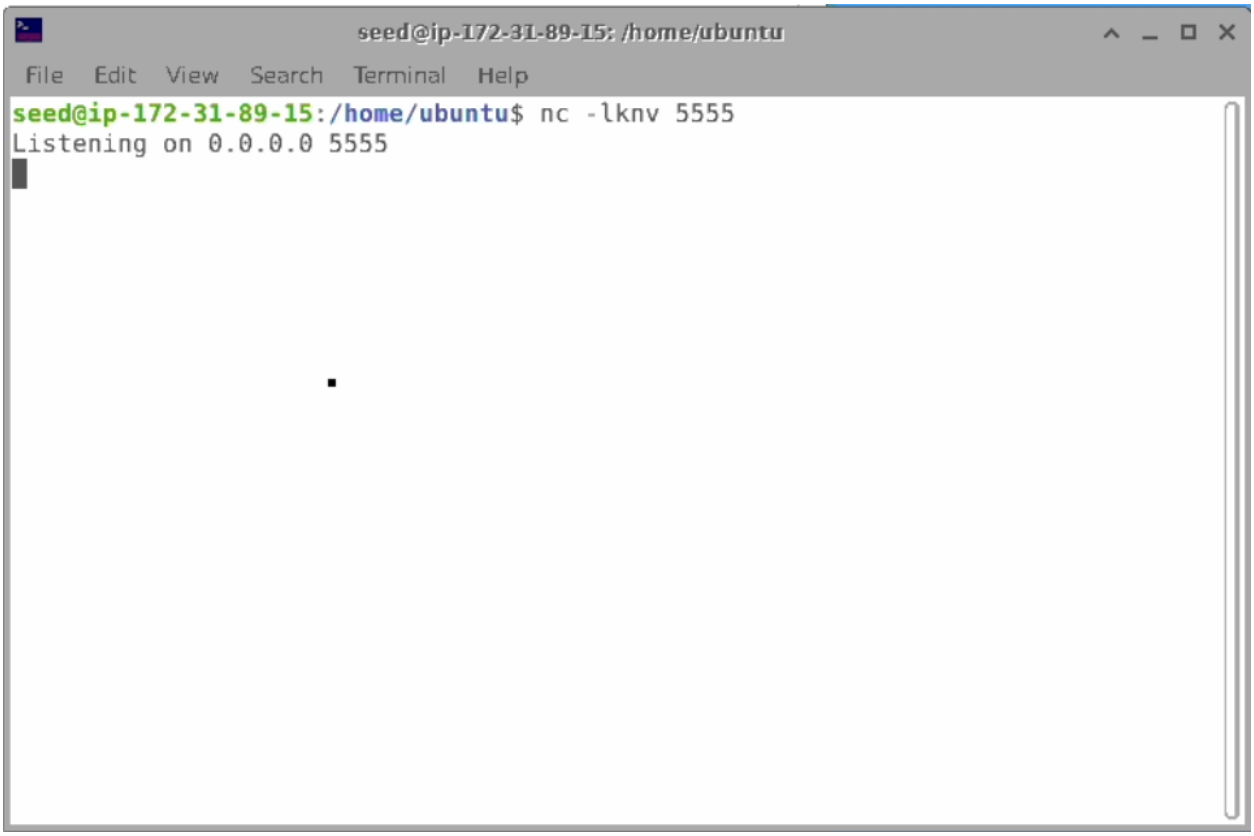


Task 3: Stealing Cookies from the Victim's Machine

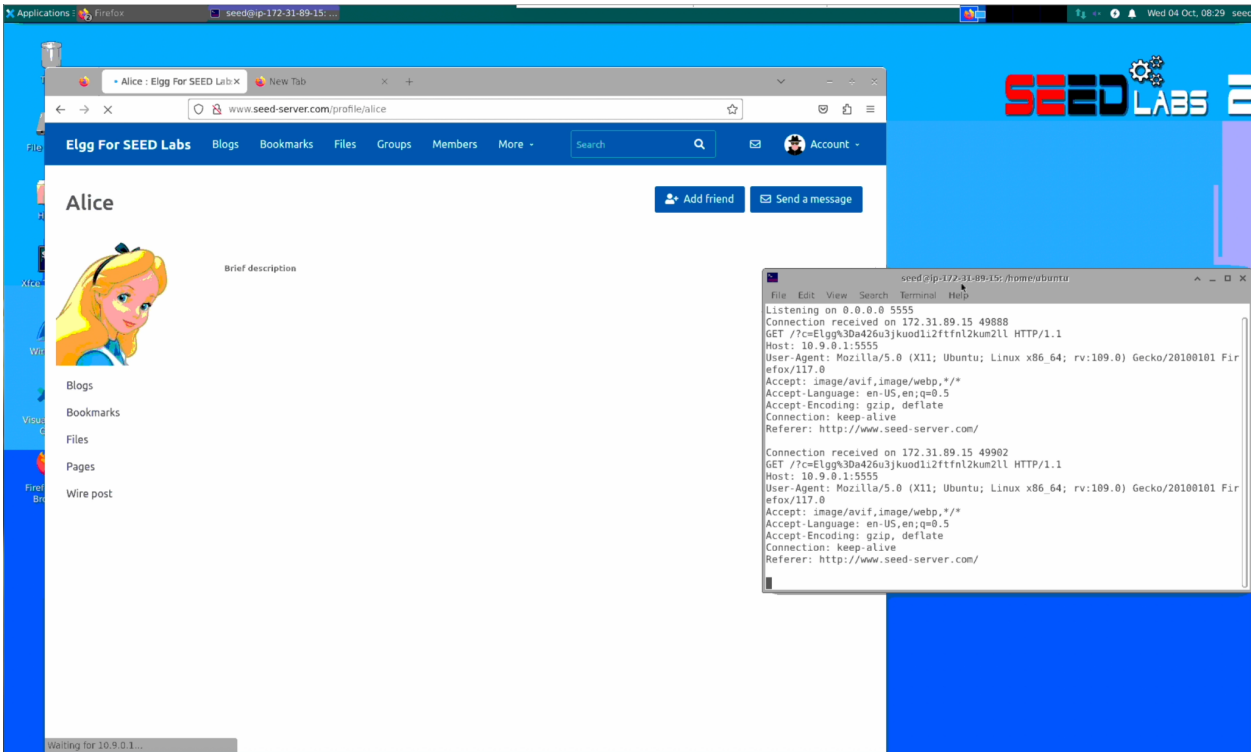
We are adding a script that the current session cookie will be sent to the listener through an HTTP GET request when viewing Alice's profile. This request was sent to local host 10.9.0.1 and port 5555 with the parameter c , which contains the users's cookies.



Here, we set the listener to discover from any IP address (0.0.0.0) and on port 5555. From here, we then login to another user and go onto Alice's profile so it can steal the current session cookie of that user.

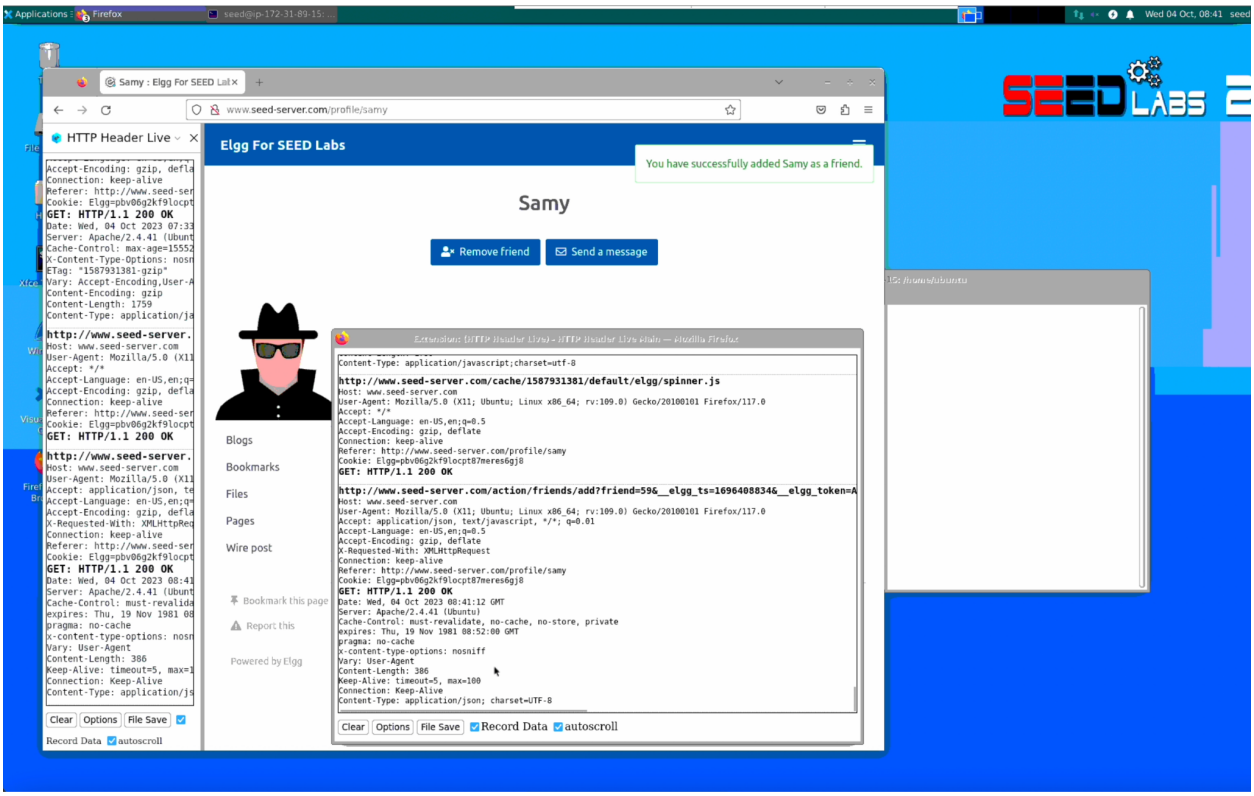


We then login to Samy's account and visit Alice's profile. From here, we can see that when Alice viewed Samy's profile, it transmitted their current session cookie to my system. You can also see additional information regarding their web browser and operating system. From the response of the HTTP GET request, we can see that the cookie we obtain is Elgg=(3D - ASCII for =) a426u3jkuod1i2ftfnl2kum2ll. This shows that this is successful.



Task 4 - Becoming the Victim's Friend

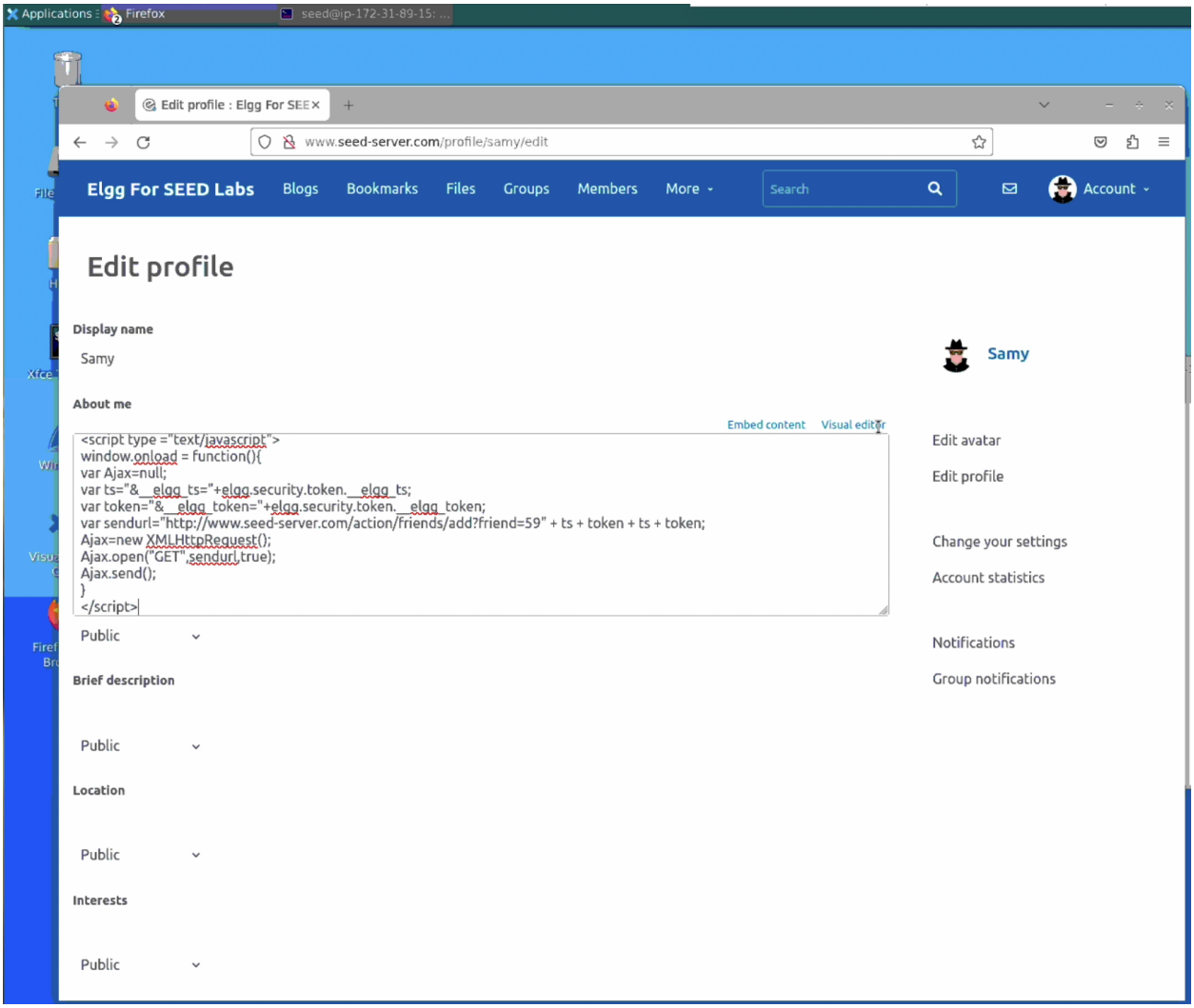
Accessed when adding Samy as a friend



This task aims to have the victim add you as a friend when the victim visits the profile. We looked at the format of the request made to the server when adding a friend legitimately. From here, we studied the layout of the request so we could exploit it. As we can see the layout is: http://www.seed-server.com/action/friends/add?friend=59 + time + token + time + token

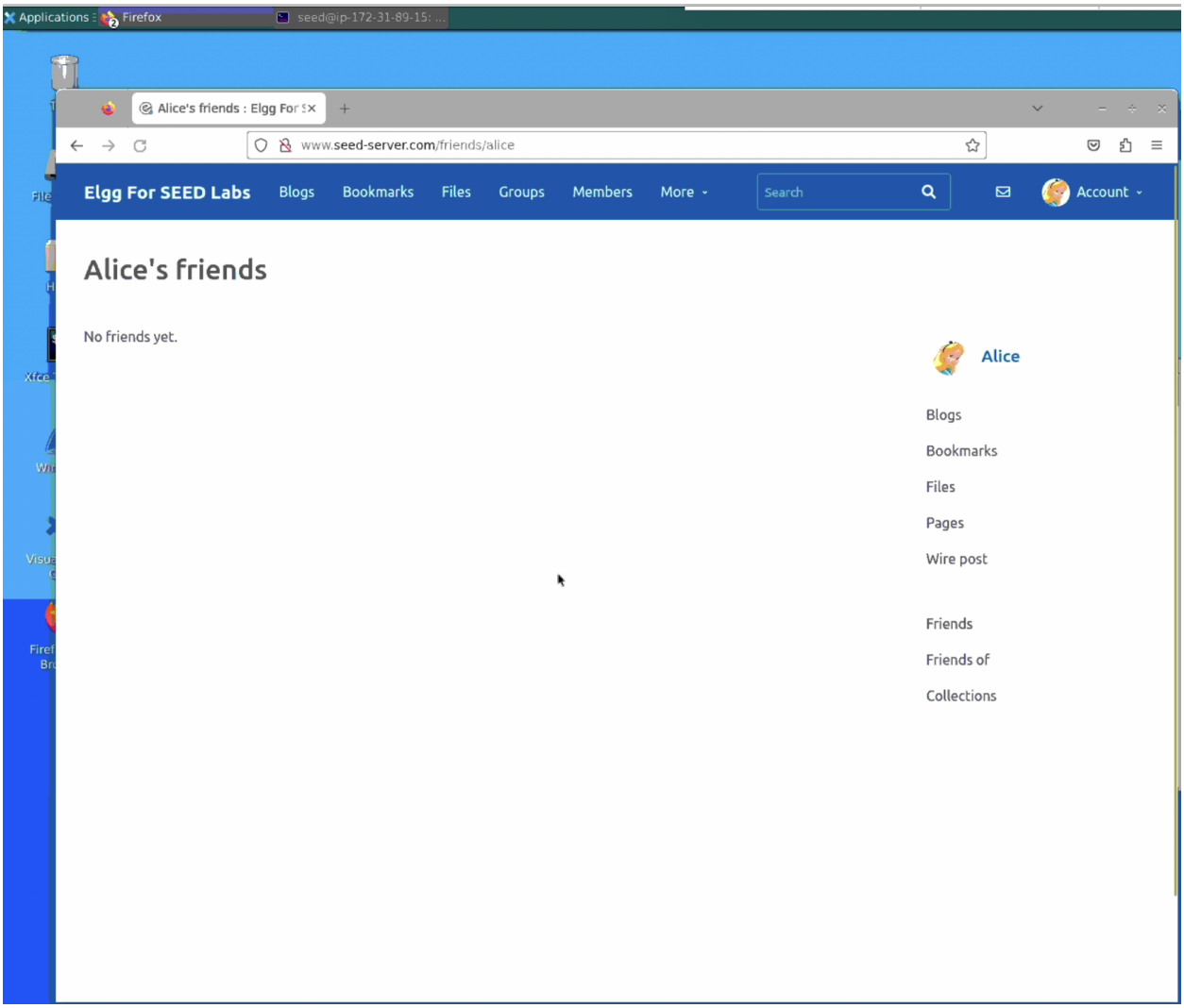
We can exploit this and add it to the user's profile here.

We then added this to Samy's 'about me' using the editing HTML

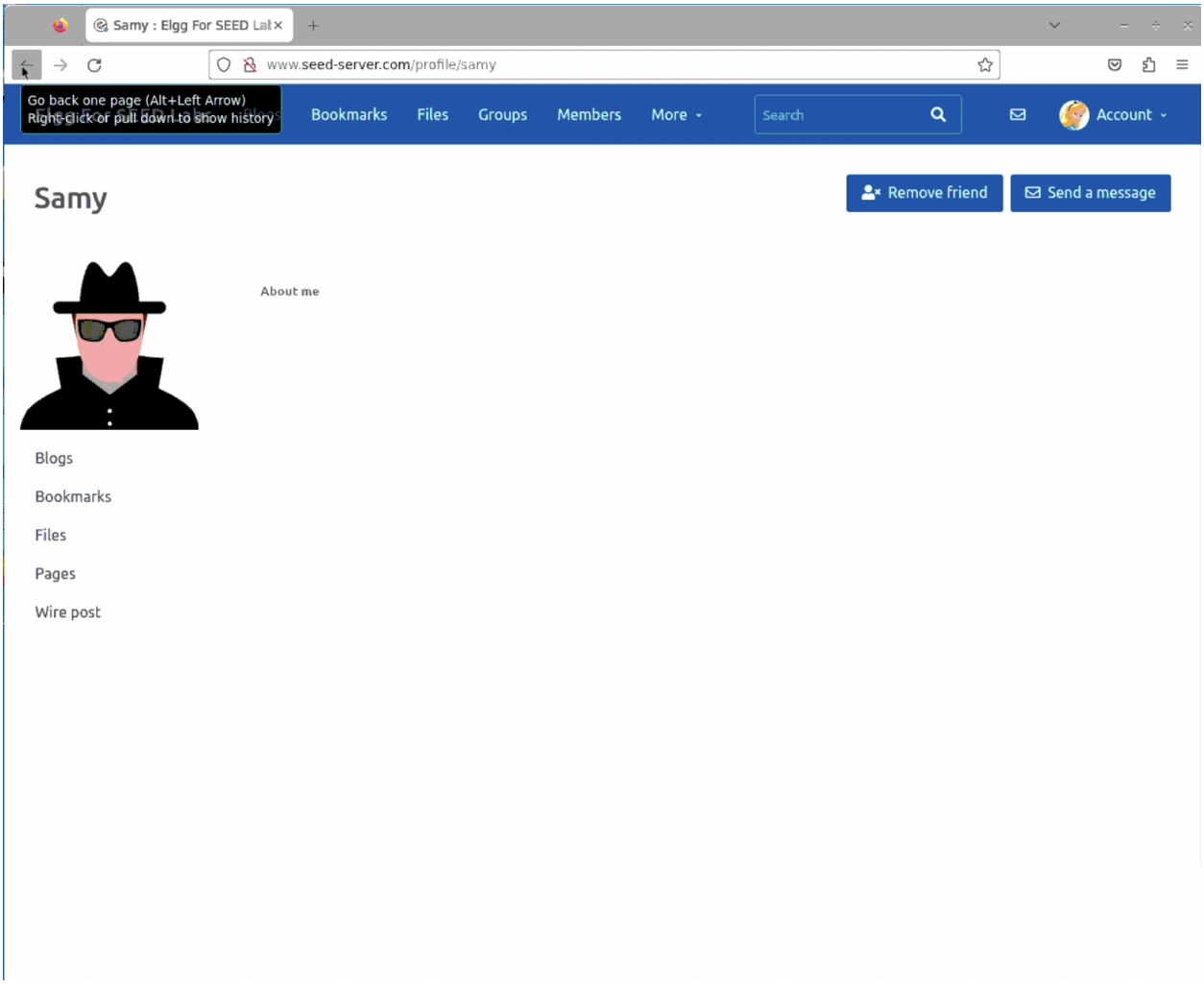


Question 1: Explain the purpose of Lines 1 and 2, why are they needed?
They are essential when making a proper request and getting the security and Elgg tokens. They are part of Elgg way of authenticating appropriate and suitable requests made by valid users and a solution to CSRF attacks. The attack would not execute a successful attack without these values as they are required for any HTTP requests. Therefore, to bypass this, we use the 'Ajax.open("GET", sendurl, true)' and send it as a get request to the server. This onload event activates whenever a user loads Samy's profile.

As we can see here, before visiting Samy's profile, Alice does not have any friends



After visiting Samy's profile, we see that Samy has been added to Alice's friends without action.

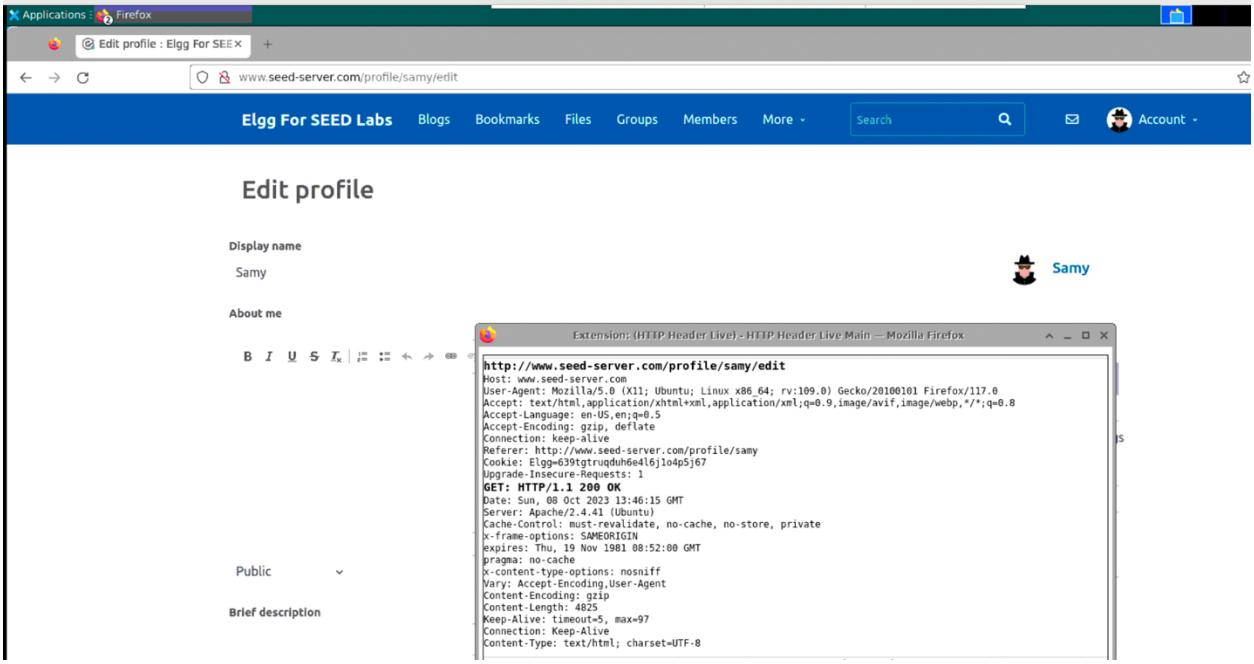


Question 2: If the Elgg application only provides the Editor mode for the "About me" field, i.e., you cannot switch to the Text mode, can you still launch a successful attack?

Launching an attack in editor mode would prove challenging for many reasons, primarily because characters used in the script ("< >") are sanitized upon saving. This would render the script ineffective when a different user views the profile. An attack would be successful if we could find a strategy to bypass this and load the code. We can use brief descriptions, but there are also character limitations. To bypass this, we would save the code in a hosted JavaScript file and use an HTTP GET request to execute it. Without other fields and removing "About Me," this attack wouldn't work correctly.

Task 5: Modifying the Victim's Profile

Here, we see the request made when editing a user's profile



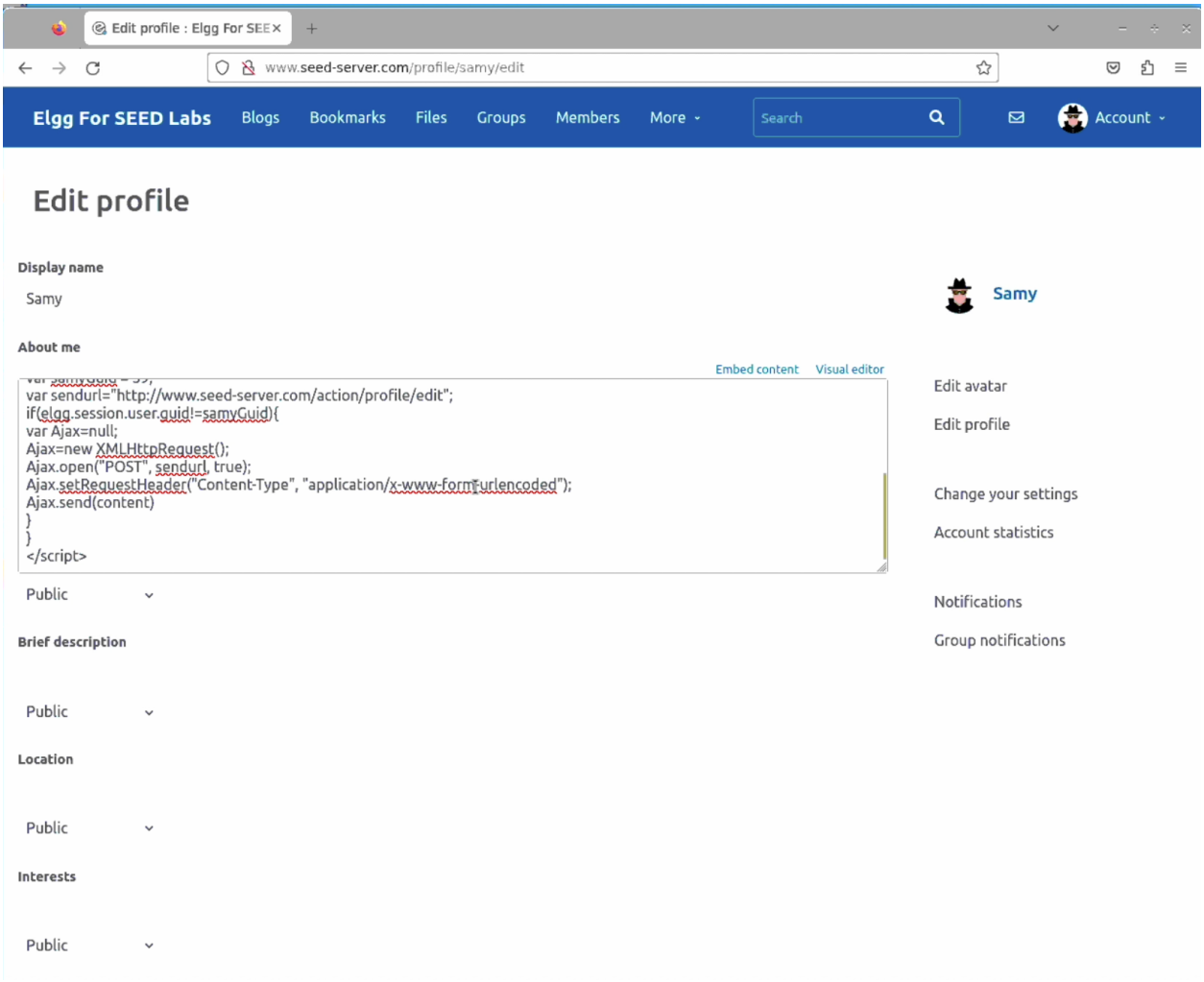

```
00918&name=Samy&description=<p>Hi</p> &accesslevel[description]=2&briefdescription=&accesslevel
```

We then use the information found from the request to adapt this code so it can exploit modifying the victim's profile

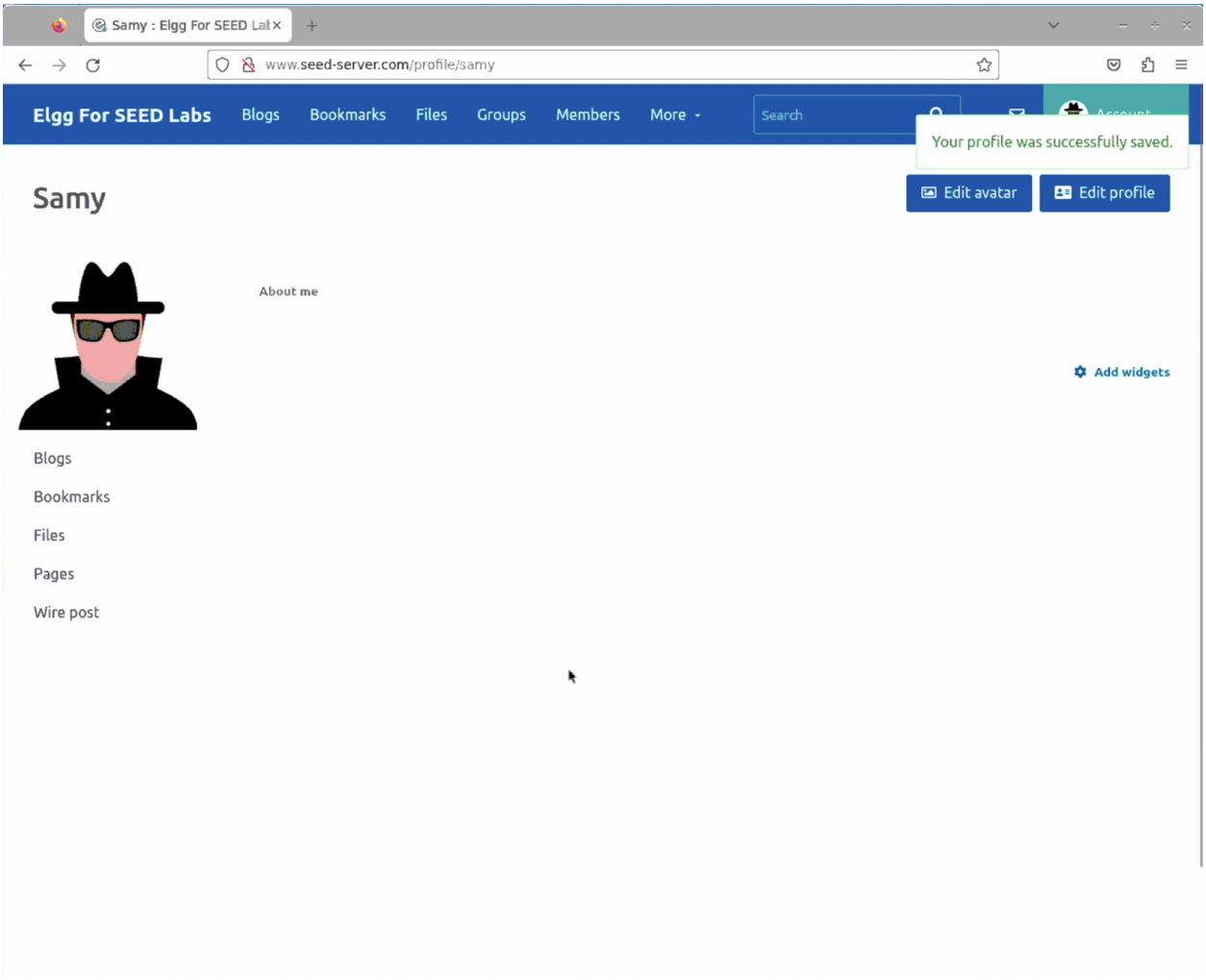
```
<script type="text/javascript">
window.onload = function(){
var userName="&name="+elgg.session.user.name;
var guid="&guid="+elgg.session.user.guid;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
var token="&__elgg_token="+elgg.security.token.__elgg_token;
var desc = "&description=changed your profile mate!" + "&accesslevel[description]=2";
var content = token + ts + userName + desc + guid;
var samyGuid = 59;
var sendurl="http://www.seed-server.com/action/profile/edit";
if(elgg.session.user.guid!=samyGuid){
var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST", sendurl, true);
Ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
Ajax.send(content)
}
}
</script>
```

We started by analyzing the request made to the server when editing the user's profile. We looked at the format and structure of the request to edit the profile through an XSS attack. When making the request, we could see that the request comprised the user's session tokens, followed by a timestamp and username. The updated fields with the access level and user GUID followed this.

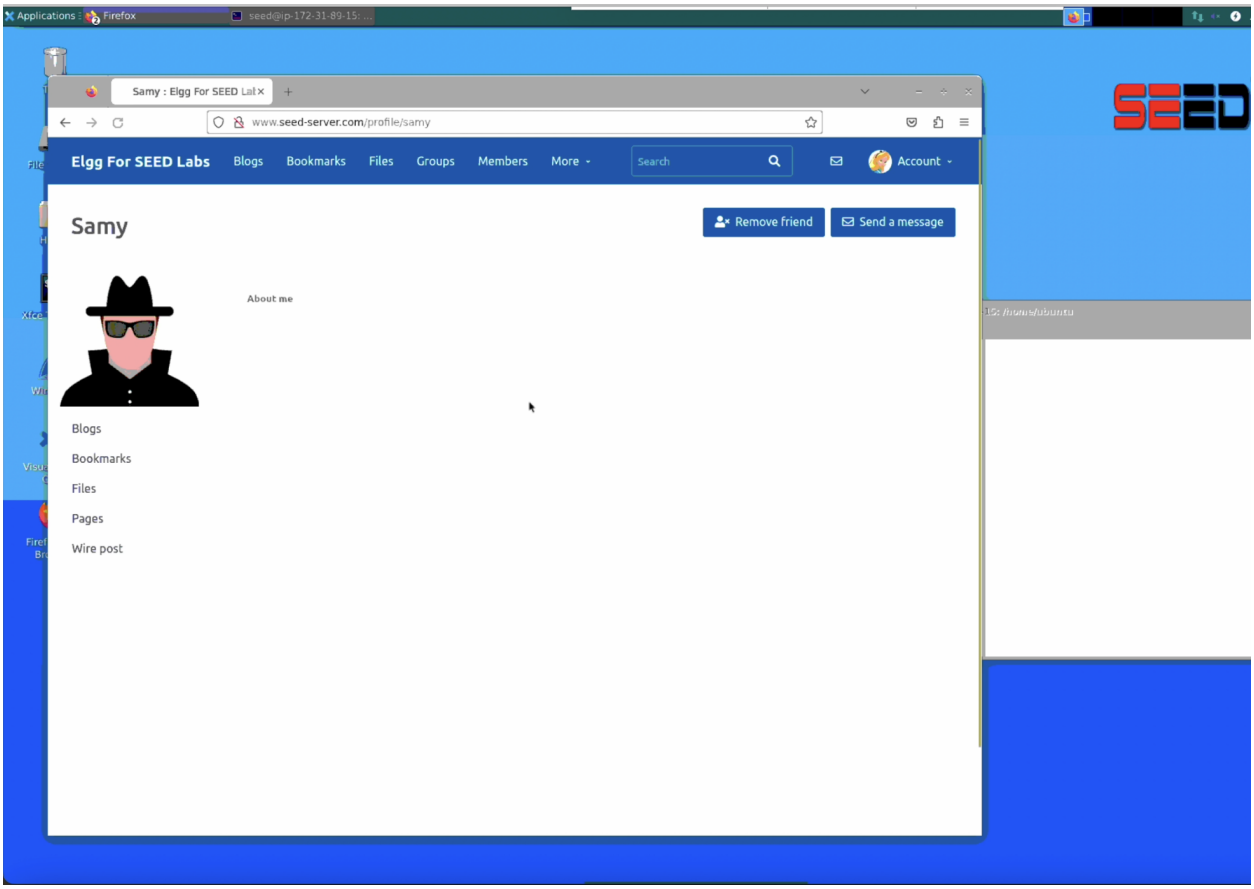
Here, we added the code to the profile



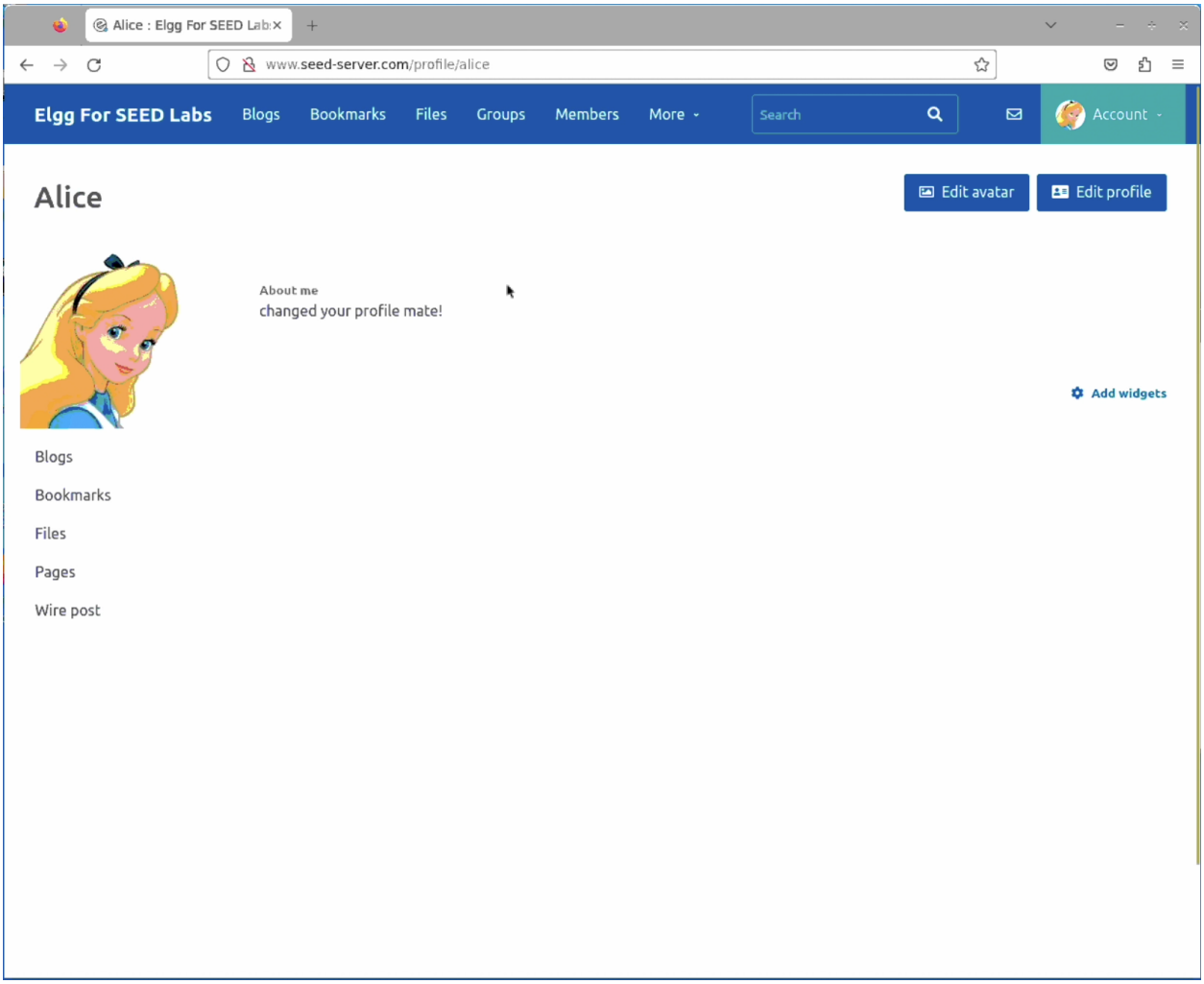
Successfully saved script



Here, we visit the profile using a different user. We can see that the attacker is not affected by his code.

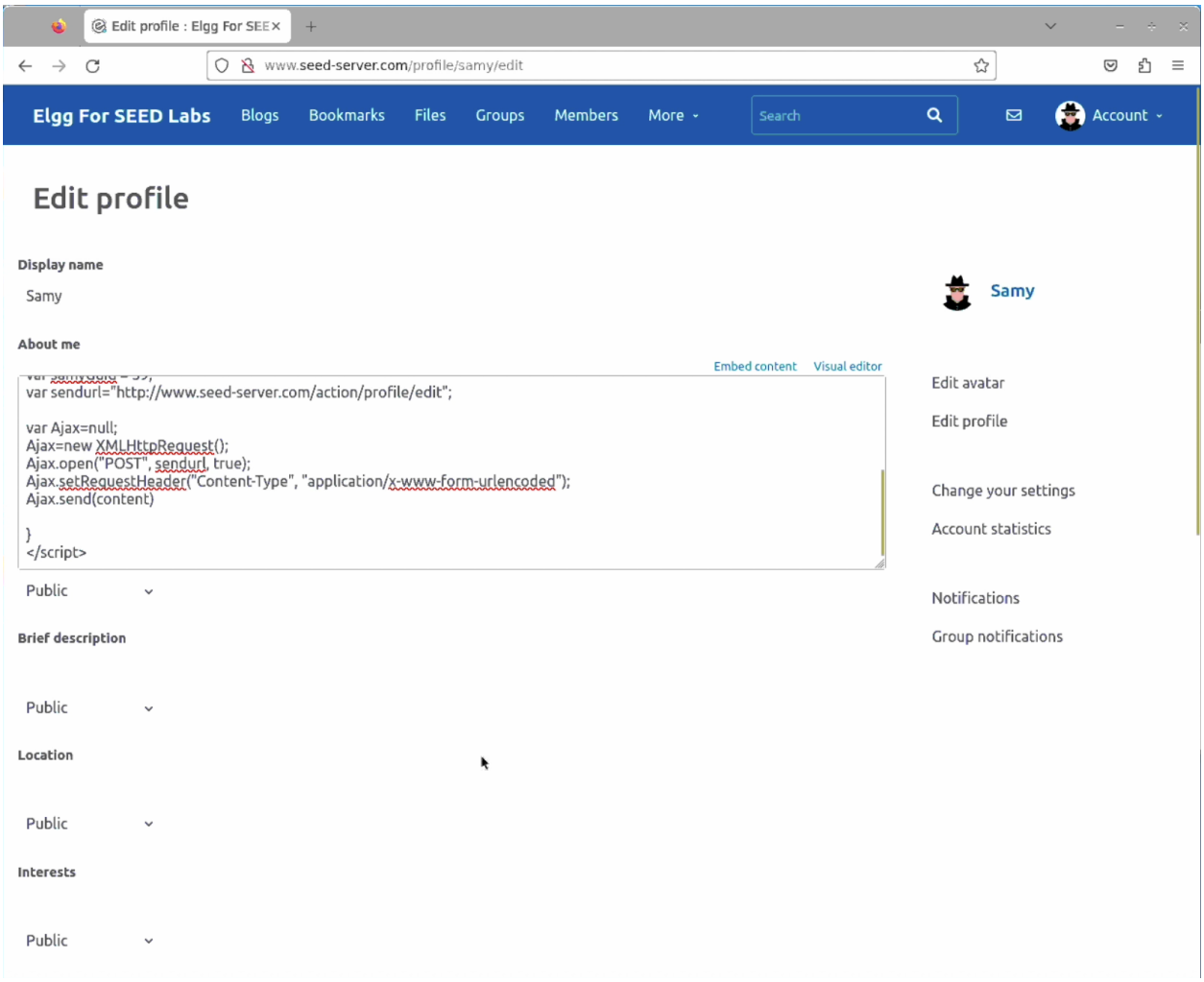


Here, we can see viewing Samy's profile caused the script embedded to be launched, which modified Alice's profile without permission.

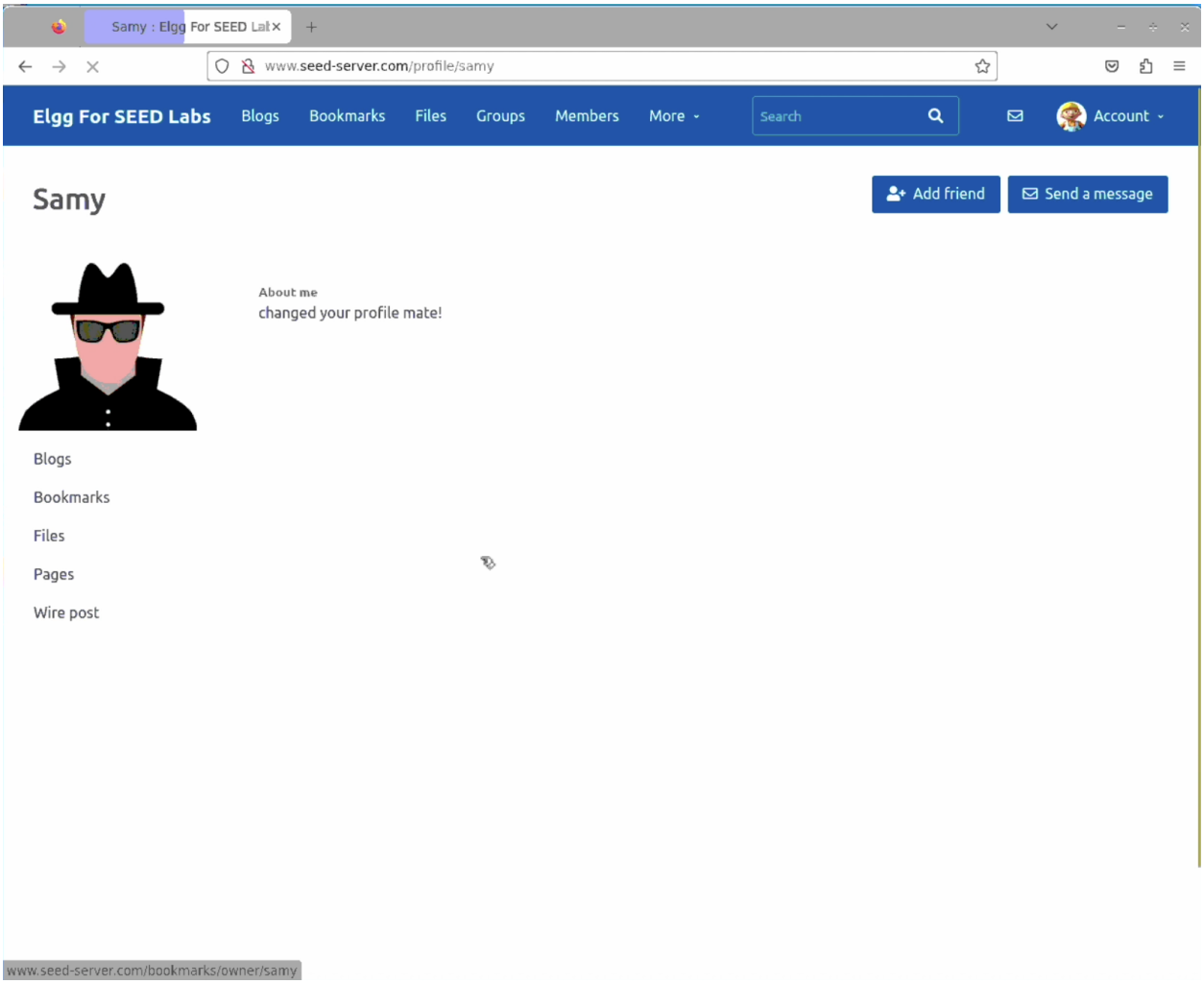


Question 3: Why do we need the if statement?

Here, we remove the if statement in the exploit code in Samy's profile

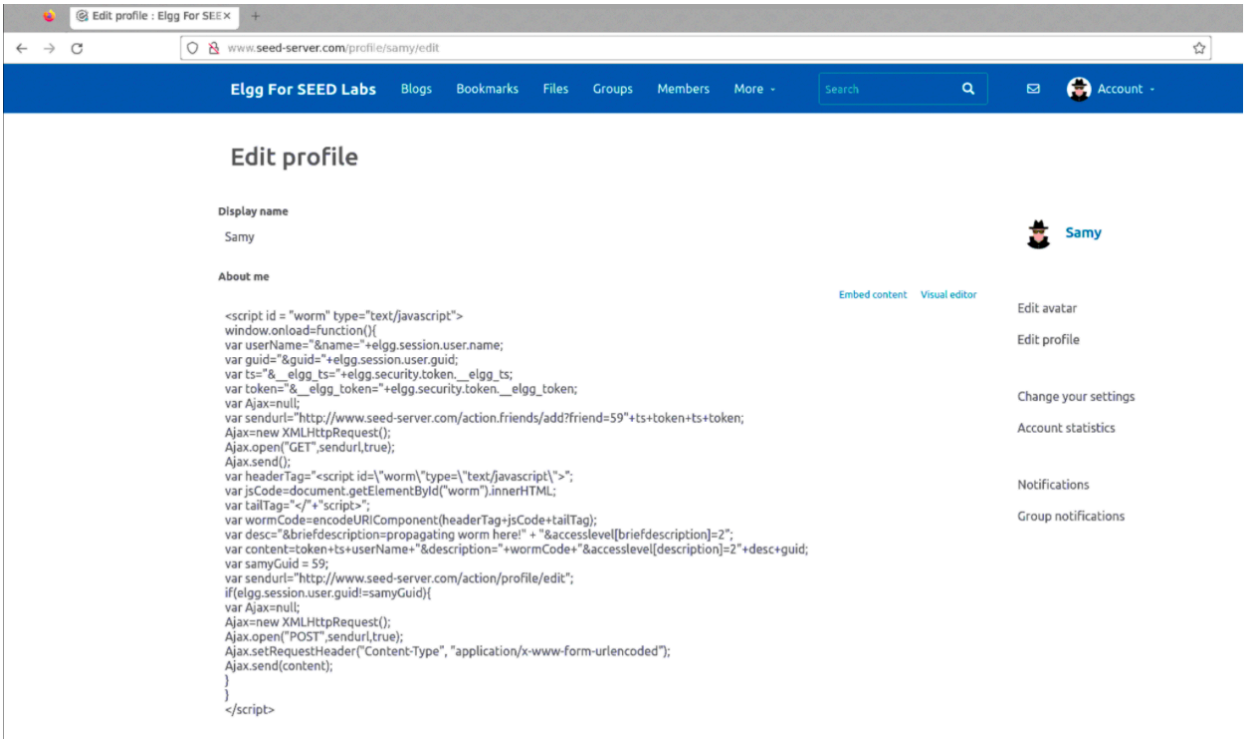


Here, we could see that removing the if statement affected the user with the exploit script rather than the user visiting the profile



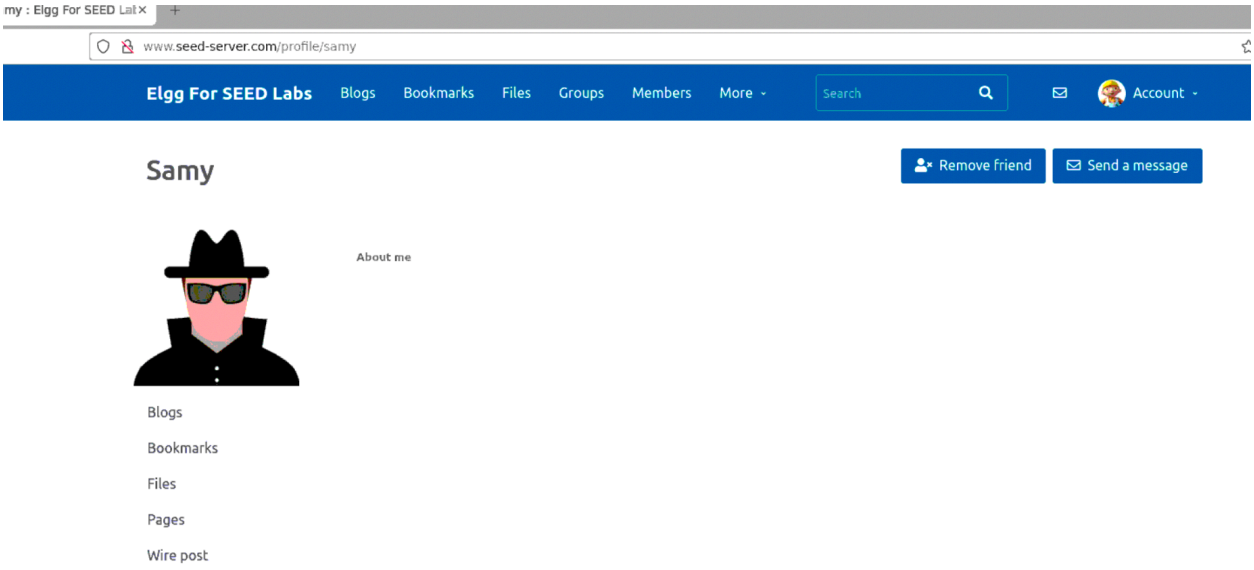
An attacker should consider the impact of profile viewing without being affected by the attack. The code is shown earlier safeguards against self-attack by using unique account identification variables. This prevents the attacker's profile from being affected when removing the attack. This code here, 'if(elgg.session.user.guid!=samyGuid)', ensures that the POST request only executes when the user's GUID does not match a specific value (59), preventing unintended self-attacks when removed.

Task 6: Writing a Self-Propagating XSS Worm

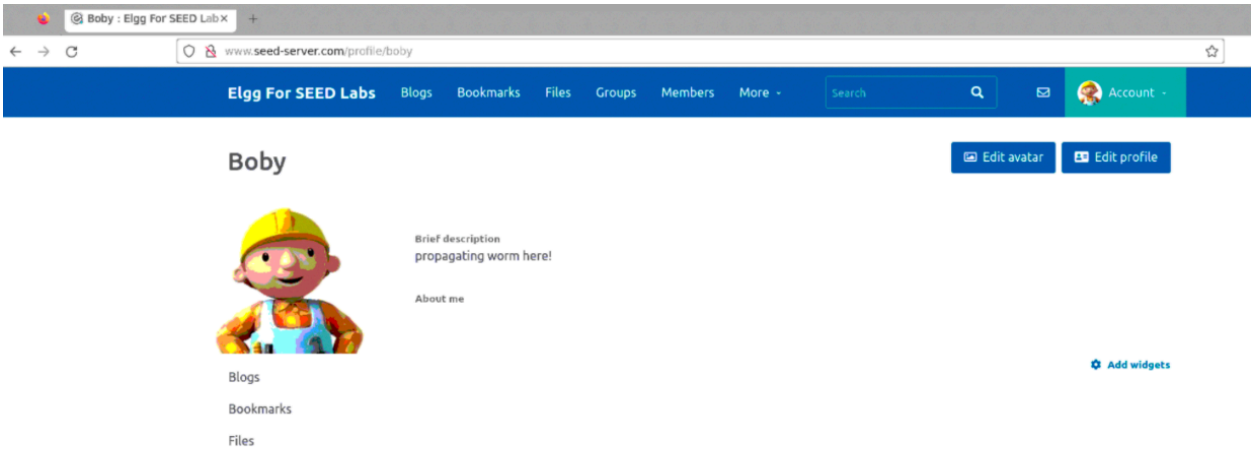


We need to add the profile to the victim's friends forcefully and embed the worm within their profile to propagate to other victims. The approach is to merge the attacks from tasks 4 and 5. One change is using the getElementById() function and .innerHTML method to reference and duplicate itself. Afterward, I append the encoded worm right after the defacing message within the attack. The subsequent steps align with the methods employed in previous approaches. I transmit the encoded content through a new XMLHttpRequest and initiate a POST request on behalf of the victim to target their profile, facilitating the spread of the worm. I also added 'AjaxFriend.open("GET", sendurlFriend, true)' to add a friend similar to task 4.

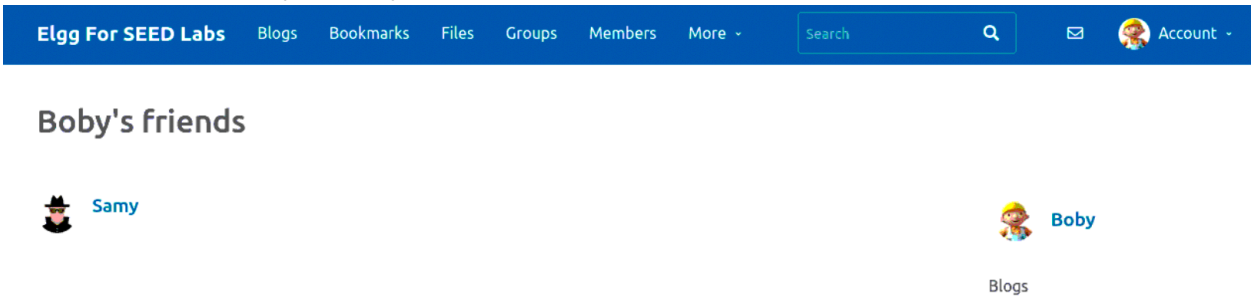
Visit Samy's profile with Bob's user



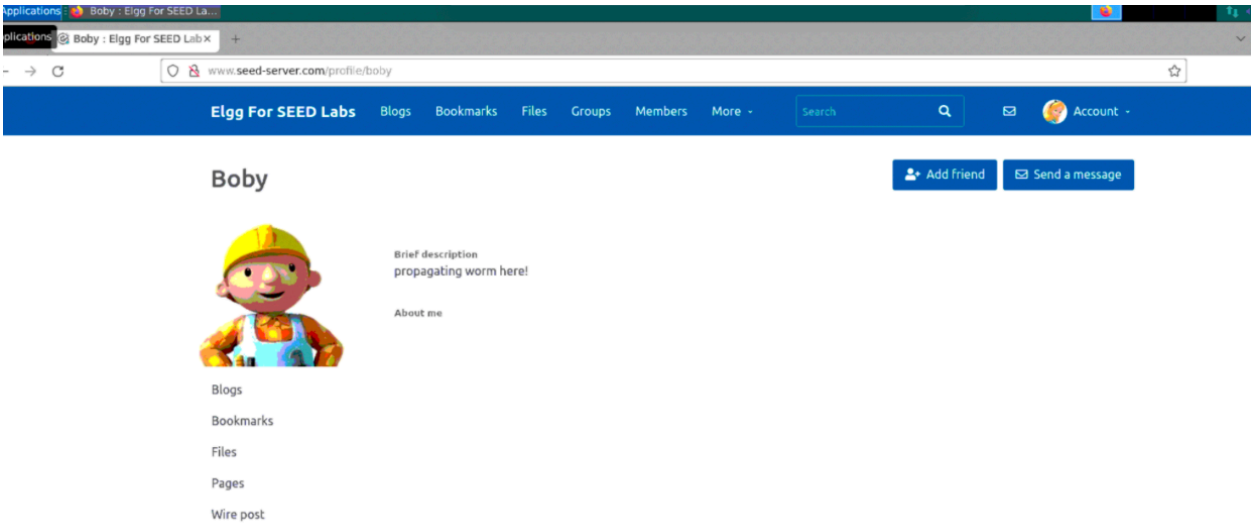
Here, we can see after visiting Samy's profile, it embedded the worm within his profile



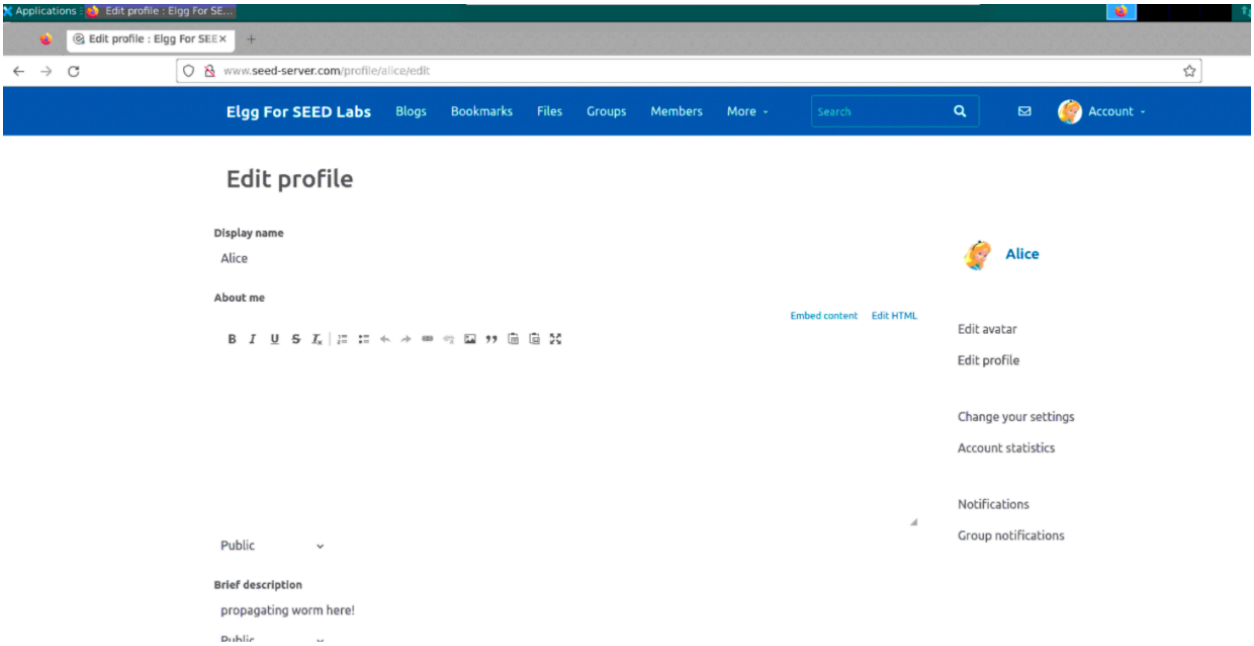
Also, has added Samy to Boby's friend list



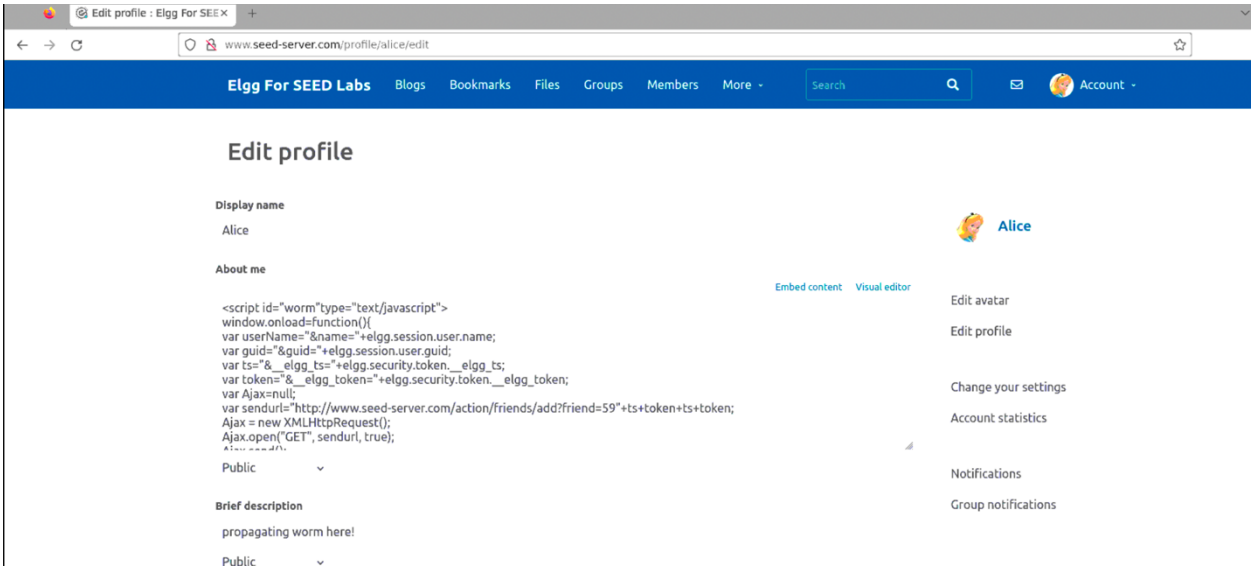
I visited Bob's profile using Alice's account



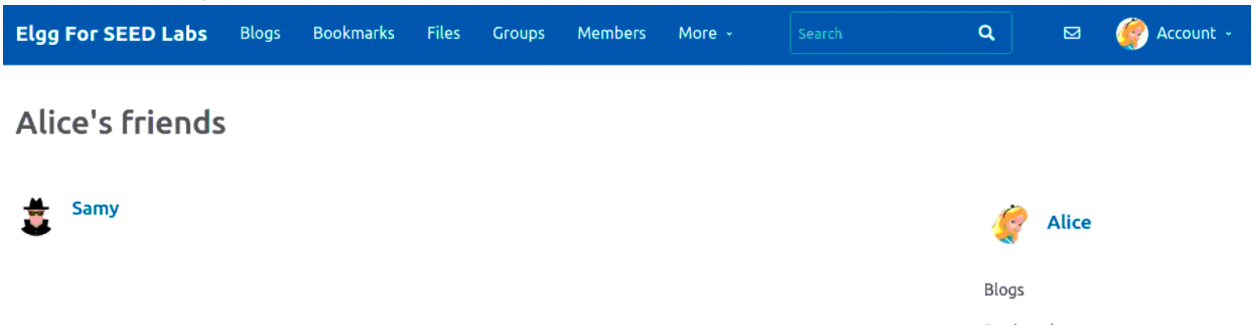
As we can see here, it propagated correctly to Alice's profile. When we edit Alice's profile, we see the propagated worm has added it to Alices about me.



When we edit Alice's HTML, we see that the worm code has been propagated correctly.



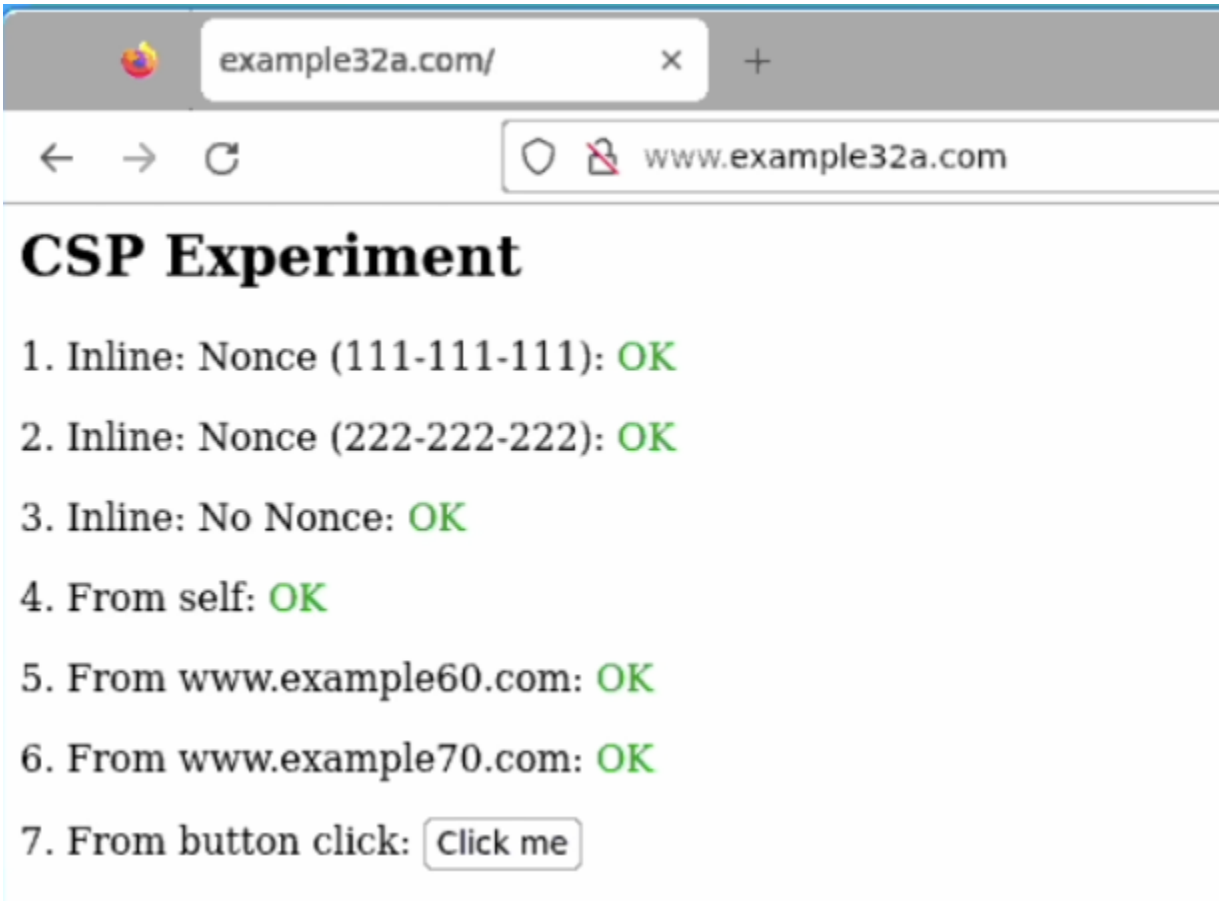
Also adds Samy to Alice's friend list



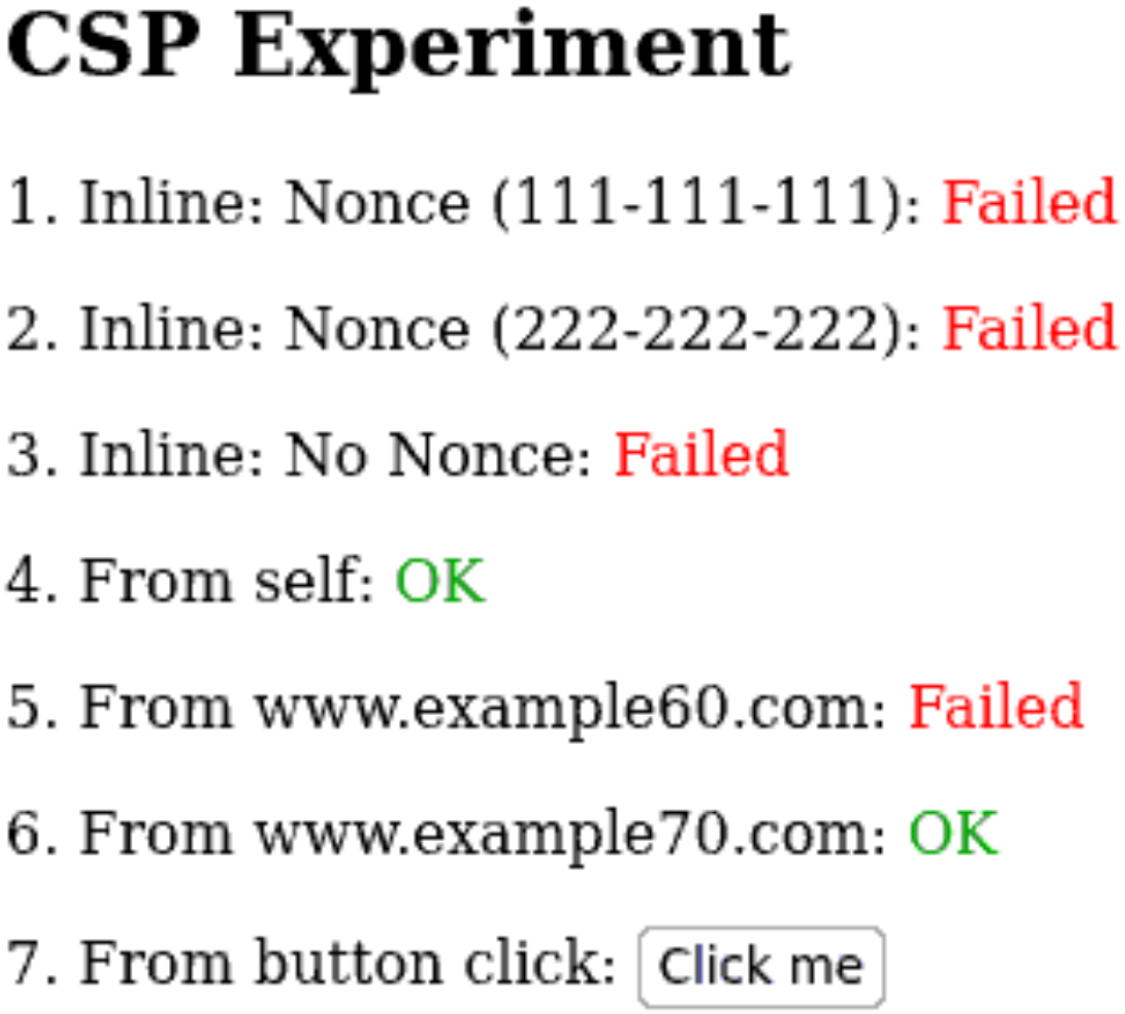
Here, we can see that this has worked correctly from Samy to Bob and then to Alice.

Task 7: Defeating XSS Attacks Using CSP

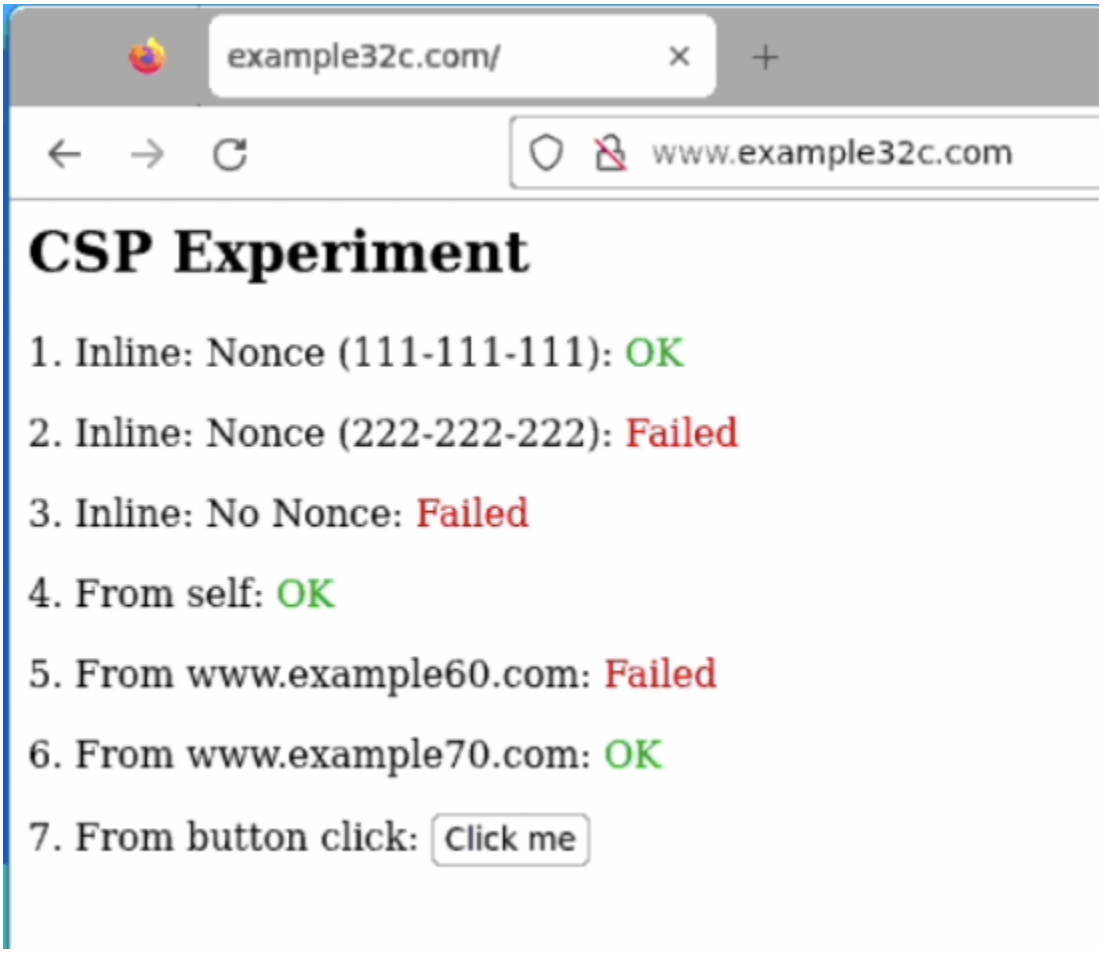
Shows example32a website. Here, we can see that it is a vulnerable website as it lacks proper restrictions when interacting with scrips externally. It allows the execution of scripts in two ways: with and without a nonce. Additionally, this website permits the execution of scripts within its domain. It also stands out as the only website that facilitates the execution of JavaScript through the 'Click me' button.



Shows example32b website. Enforces stricter security measures in its CSP configurations. It explicitly prevents script execution using the specified nonces and strictly forbids execution without a nonce. It does grant permission for script execution within its domain. It also allows the execution of scripts originating from example70.com while it blocks execution from example60.com

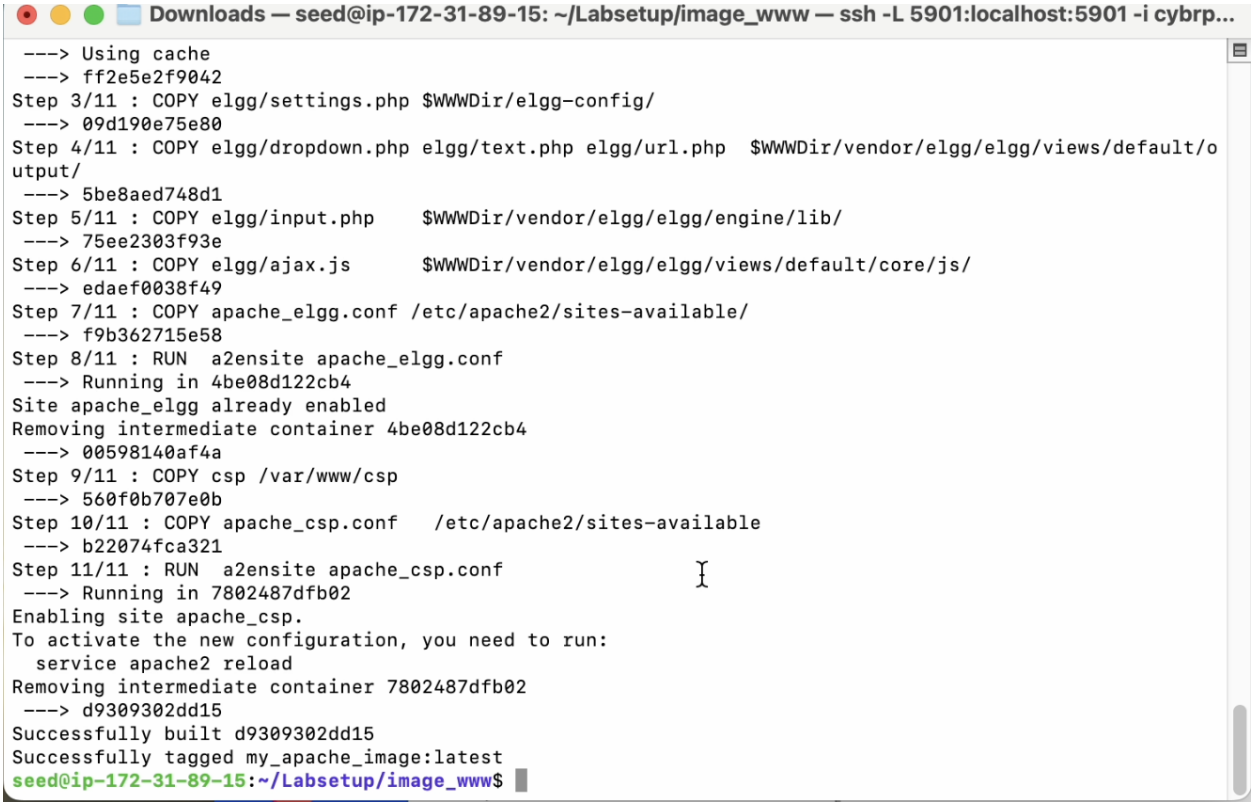


Shows example32c website. Similar to example32b, it enforces stricter security measures in its CSP configurations. Allows scripts to be run with a nonce of 111-111-111 and allows execution of scripts originating from example70.com while it blocks execution from example60.com.

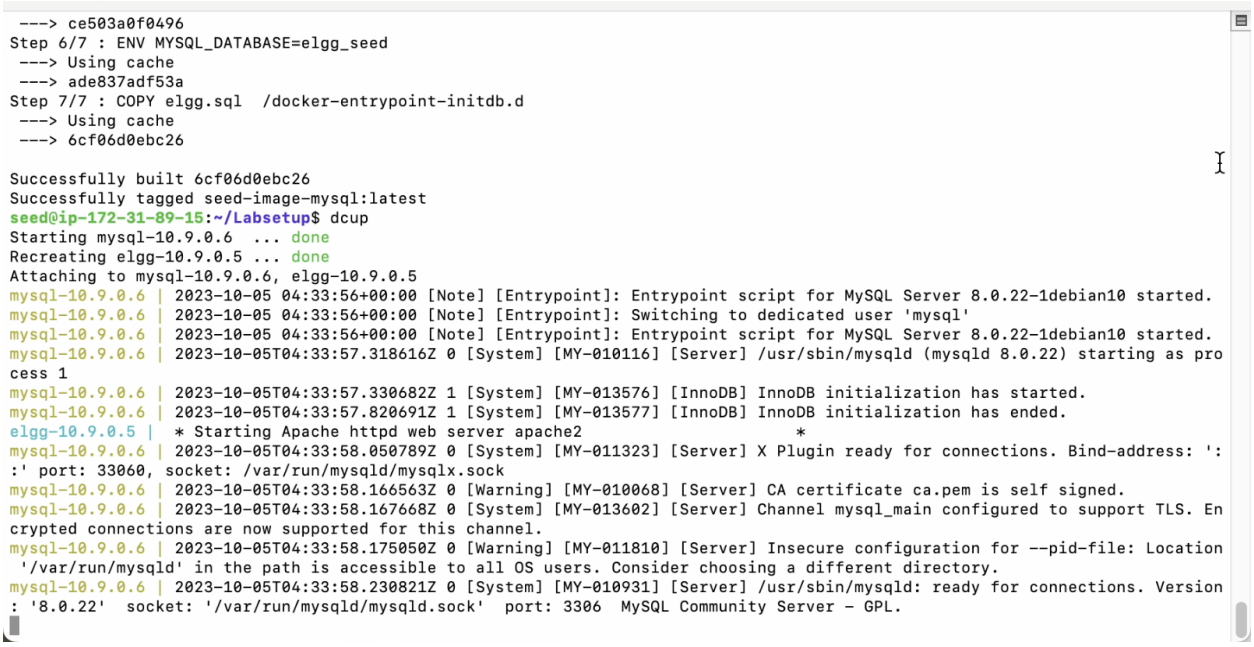


Change the server's configuration on example32b so 5 and 6 are OK. Must meet the requirement to open security rule that allows www.example60.com. We updated '*.example60.com' to the script source 'apache.csp.conf'. This will enable scripts from a subdomain of example60.com to be whitelisted and executed on example32b. We also found that example32c uses phpindex.php, so that we can edit that later.

Rebuild everything using the command 'dcbuild'.



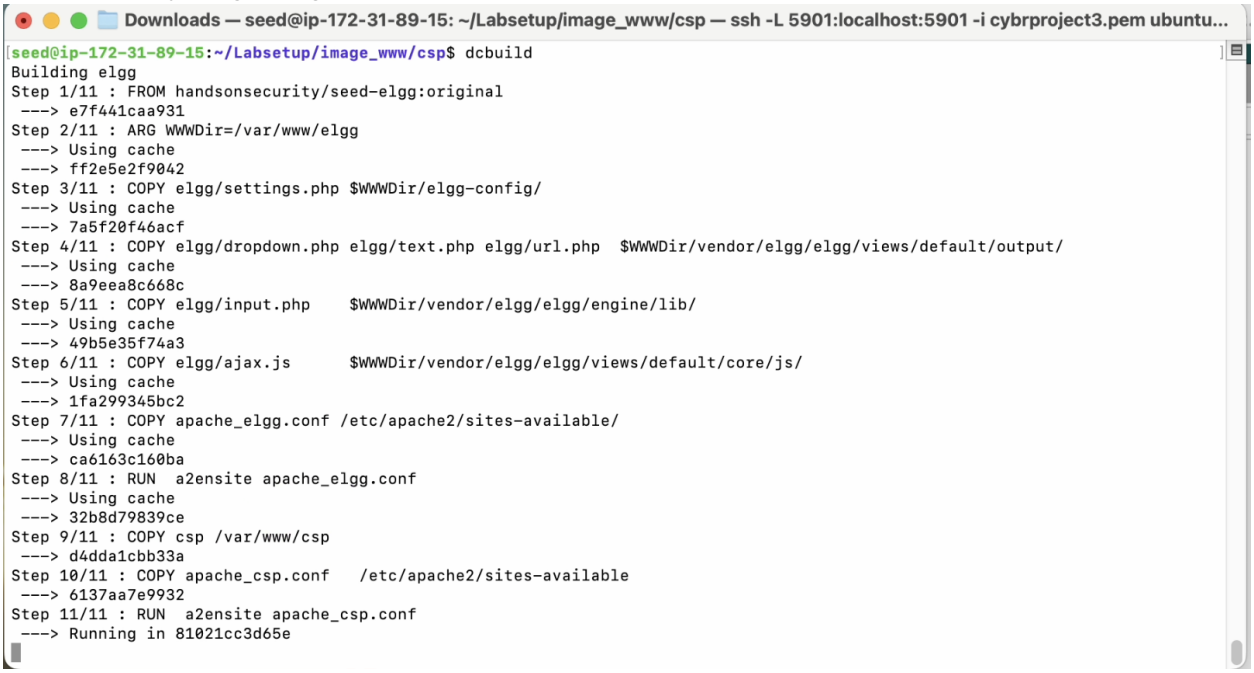
Executed again using dcup



We can now see the changes have allowed scripts from example60.com to be executed.

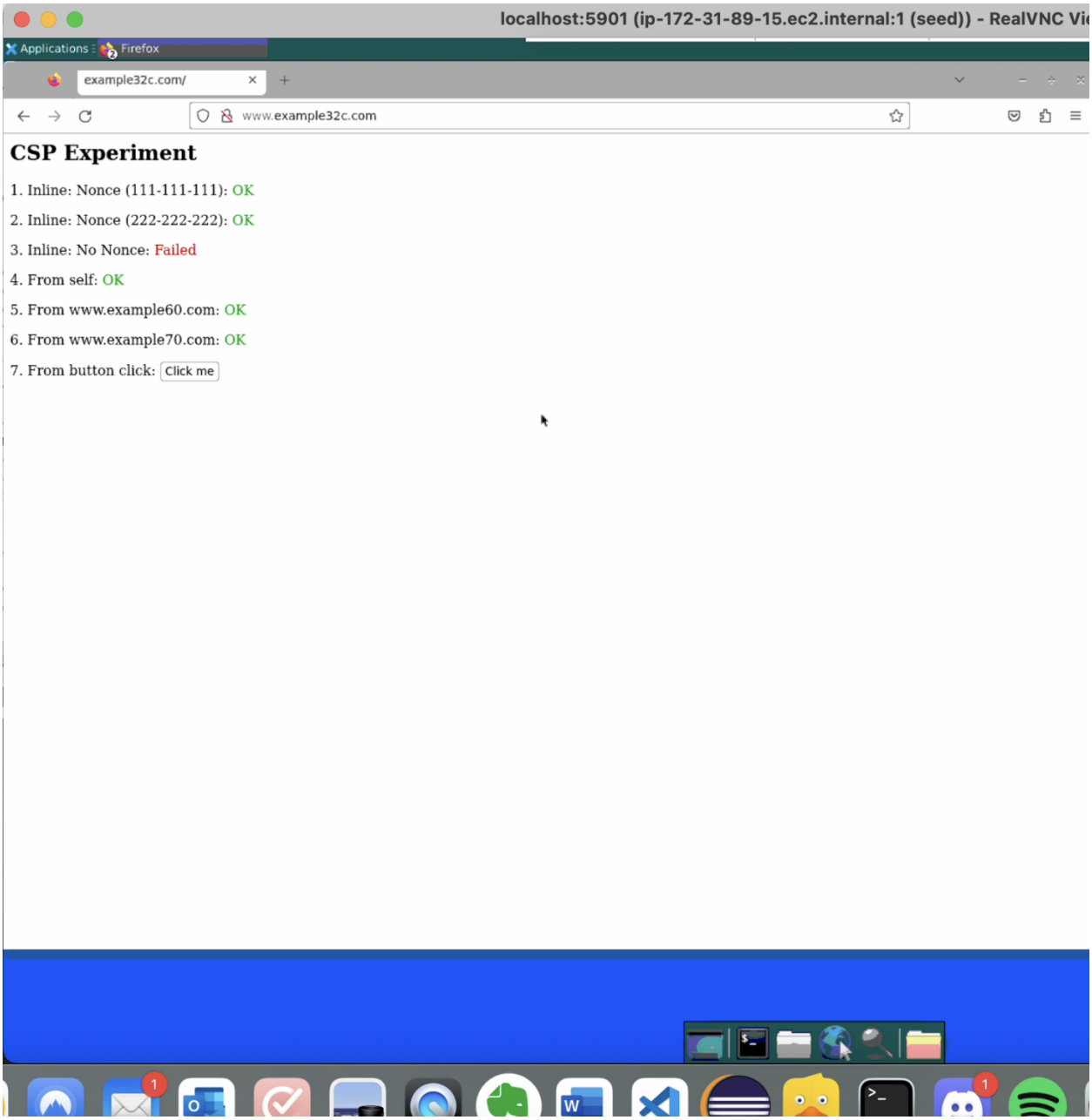
Change server configurations so Areas 1,2,4,5,6 displays OK. We found out earlier that website example32c uses phpindex.php. We, therefore, edit it to allow nonce: 111-111-111 and 222-222-222 to allow scripts with this nonce to be executed.

Rebuild everything using dcbuild



Execute everything using dcup

As we can see, the changes we made in the phpindex.php were effective, as Task 1,2,4,5,6 is now OK.



Please explain why CSP can help prevent Cross-Site Scripting attacks.
CSP stands as a highly effective defense mechanism against XSS attacks by regulating the origins from which content can be fetched and executed on a web page. By regulating the origins from which content can be fetched and run on a webpage, CSP can defend against malicious scripts injected by potential attackers. It does this by disallowing the execution of scrips from unauthorized sources, thus rendering any rogue scrips ineffective against the web application. In the face of XSS attacks, where attackers attempt to insert harmful code. If the injected script fails to comply with the stringent CSP policy, it remains alert, rendering the

attacker's efforts pointless. Also, creating a whitelist of trusted domains or sources is another strategy it uses. Only scripts originating from these pre-approved sources are allowed to execute. Any attempt to load content from non-whitelisted domains is met with a resolute blockage, considerably diminishing the risk of XSS attacks that rely on loading malevolent scripts from external sources.