LAB TASK- 03

Name: Khairatun Hissan

ID: 180042103

Program: SWE

DEPT: CSE

Course code: SWE-4504

Group: A

Cross-Site Scripting (XSS) Attack Lab

(Web Application: Elgg)

Overview

Cross-site scripting (XSS) is a type of vulnerability commonly found in web applications. This vulnerability makes it possible for attackers to inject malicious code (e.g. JavaScript programs) into the victim's web browser. Using this malicious code, attackers can steal a victim's credentials, such as session cookies.

In this lab, we need to exploit this vulnerability to launch an XSS attack on the modified Elgg, in a way that is similar to what Samy Kamkar did to MySpace in 2005 through the notorious Samy worm.

This lab covers the following topics:

1. Cross-Site Scripting attack.
2. XSS worm and self-propagation.
3. Session cookies.
4. HTTP GET and POST requests.
5. JavaScript and Ajax.
6. Content Security Policy (CSP)

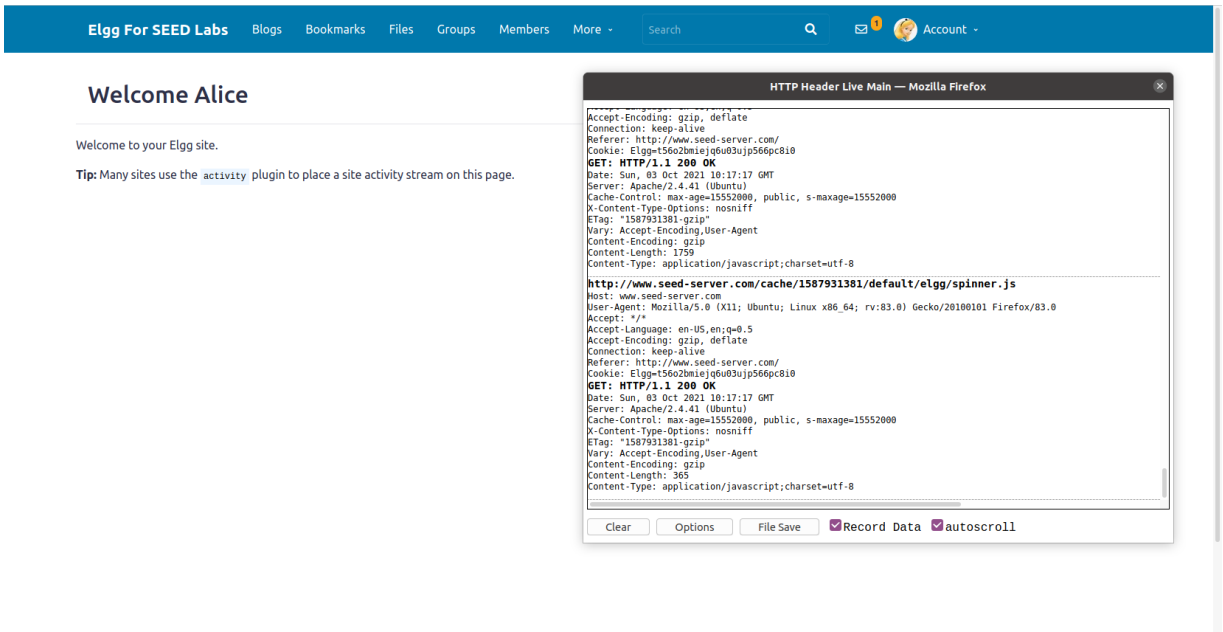
Lab Tasks

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

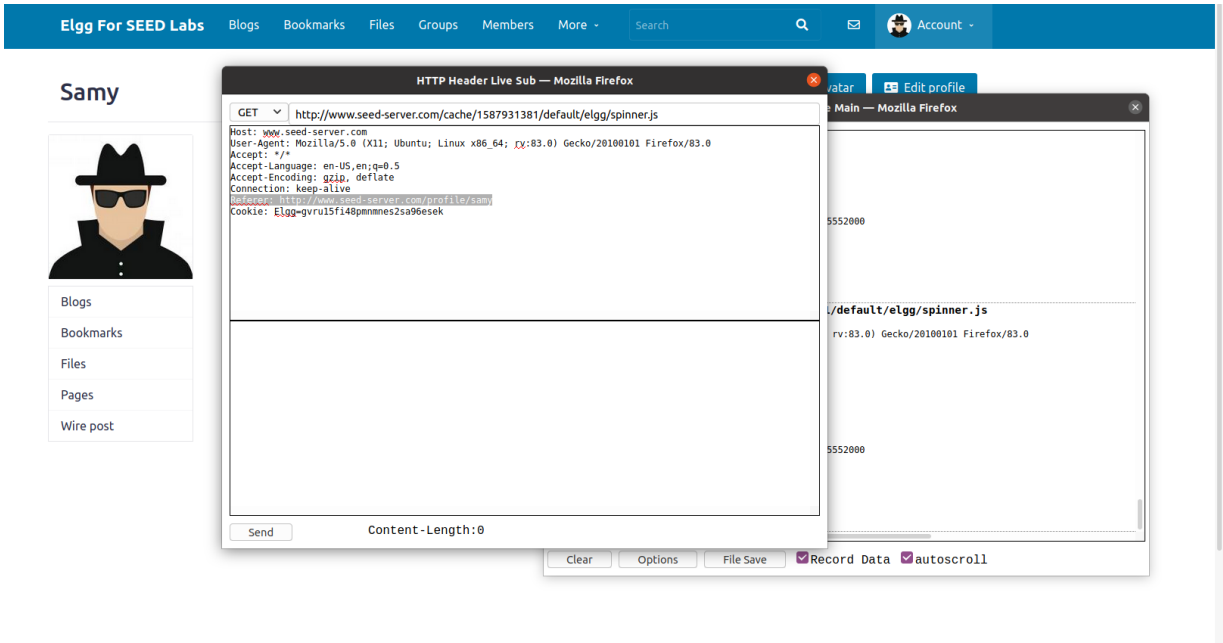
In this lab, we need to construct HTTP requests. To figure out what an acceptable HTTP request in Elgg looks like, we can use a Firefox add-on called "HTTP Header Live" for this purpose.

Example:

Log in as Alice:

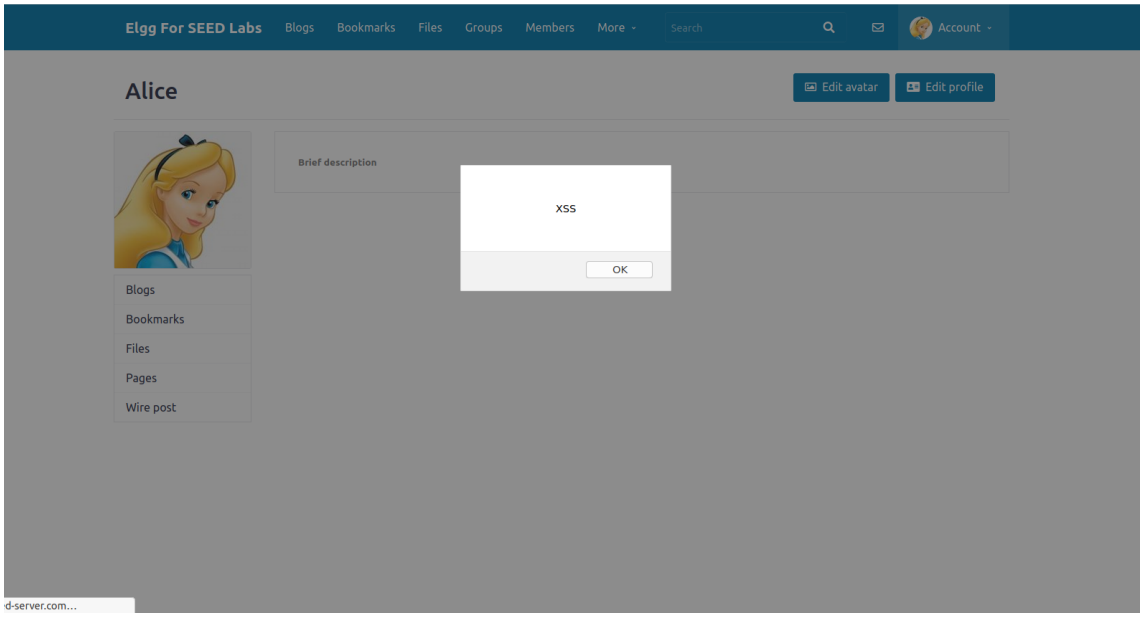


View Samy's profile:



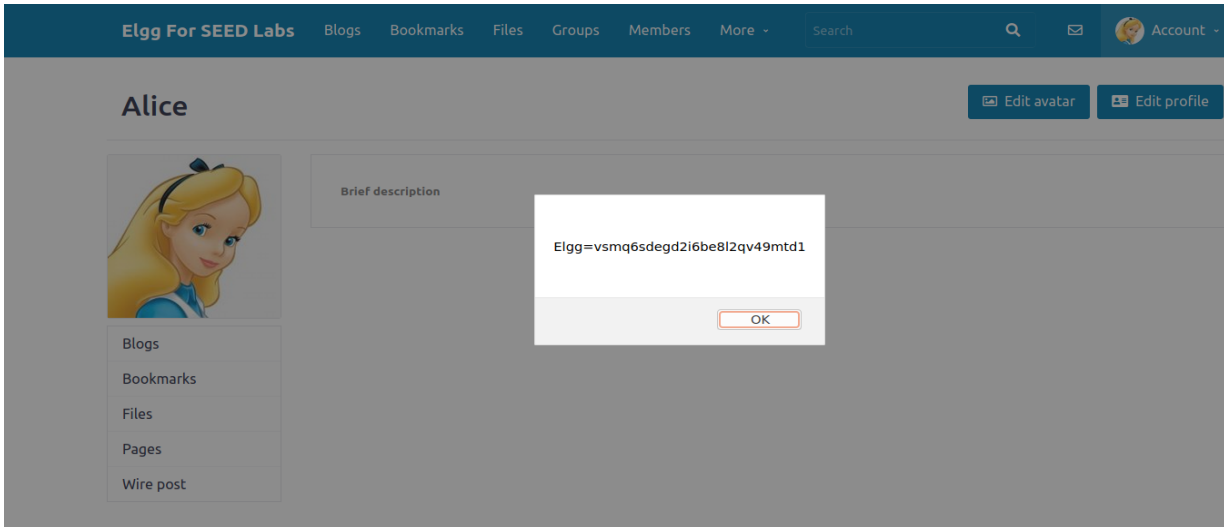
Task 1: Posting a Malicious Message to Display an Alert Window

The objective of this task is to embed a JavaScript program in your Elgg profile, such that when another user views your profile, the JavaScript program will be executed and an alert window will be displayed.



Task 2: Posting a Malicious Message to Display Cookies

The objective of this task is to embed a JavaScript program in your Elgg profile, such that when another user views your profile, the user's cookies will be displayed in the alert window.

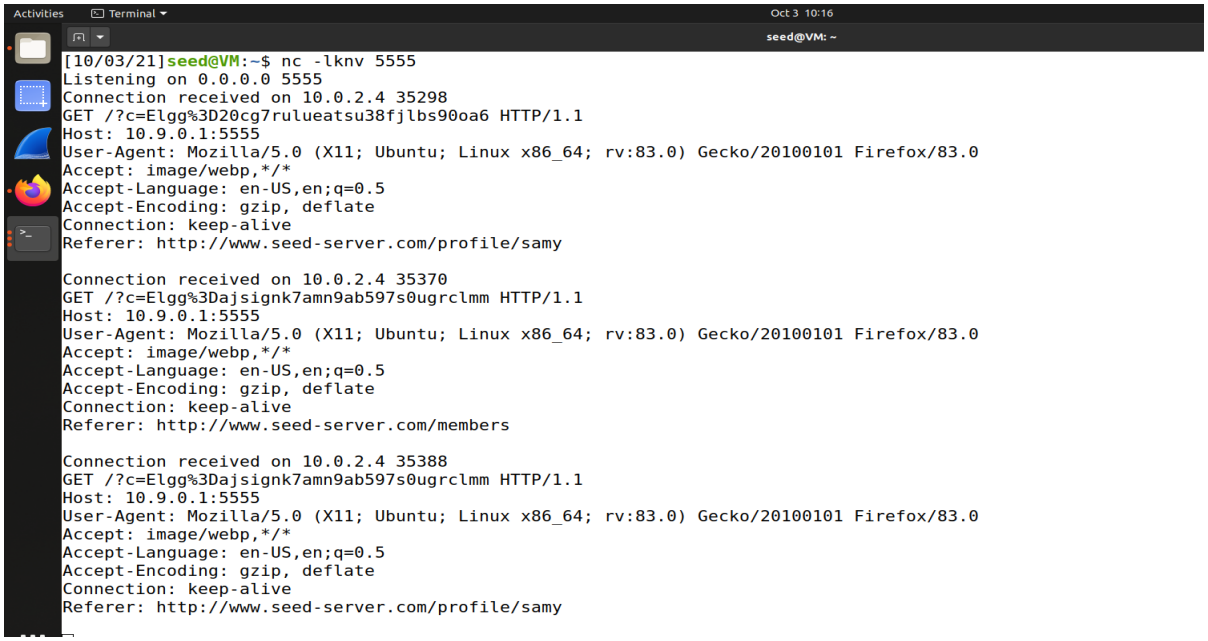


Task 3: Stealing Cookies from the Victim's Machine

.In this task, the attacker wants the JavaScript code to send the cookies to himself/herself. To achieve this, the malicious JavaScript code needs to send an HTTP request to the attacker, with the cookies appended to the request.

Inserting malicious script in attacker's profile:

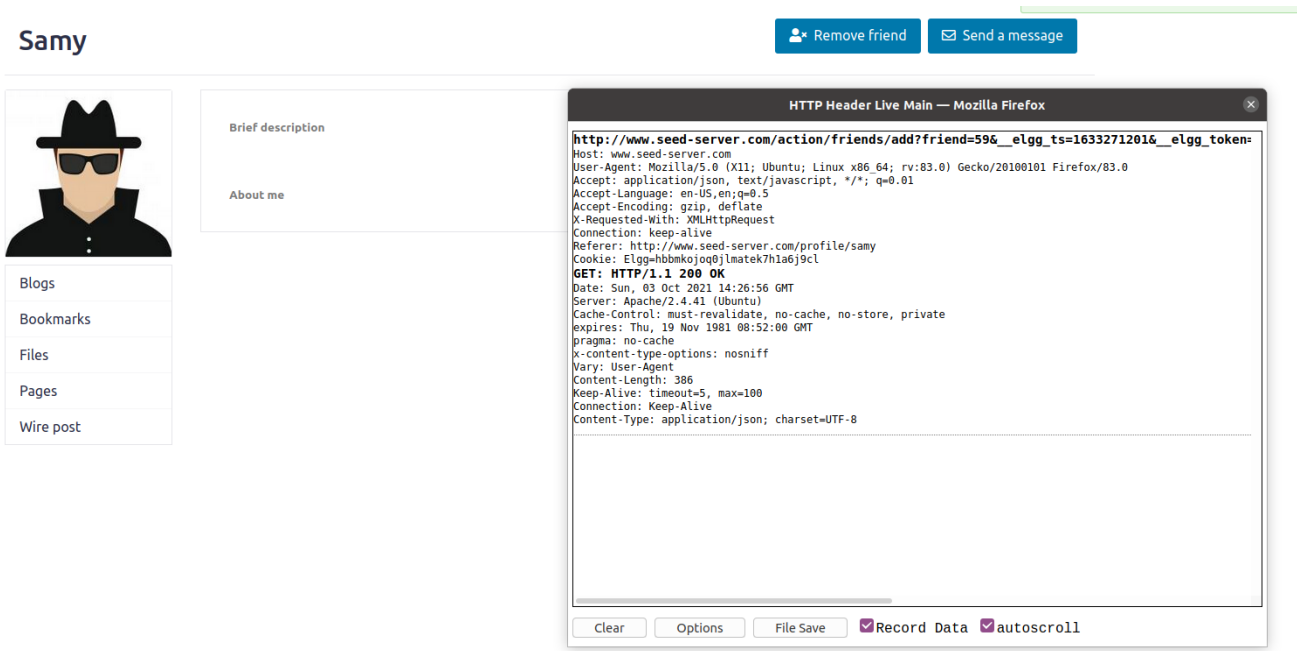
Victim visiting attacker's profile and attacker stealing victim's cookie:



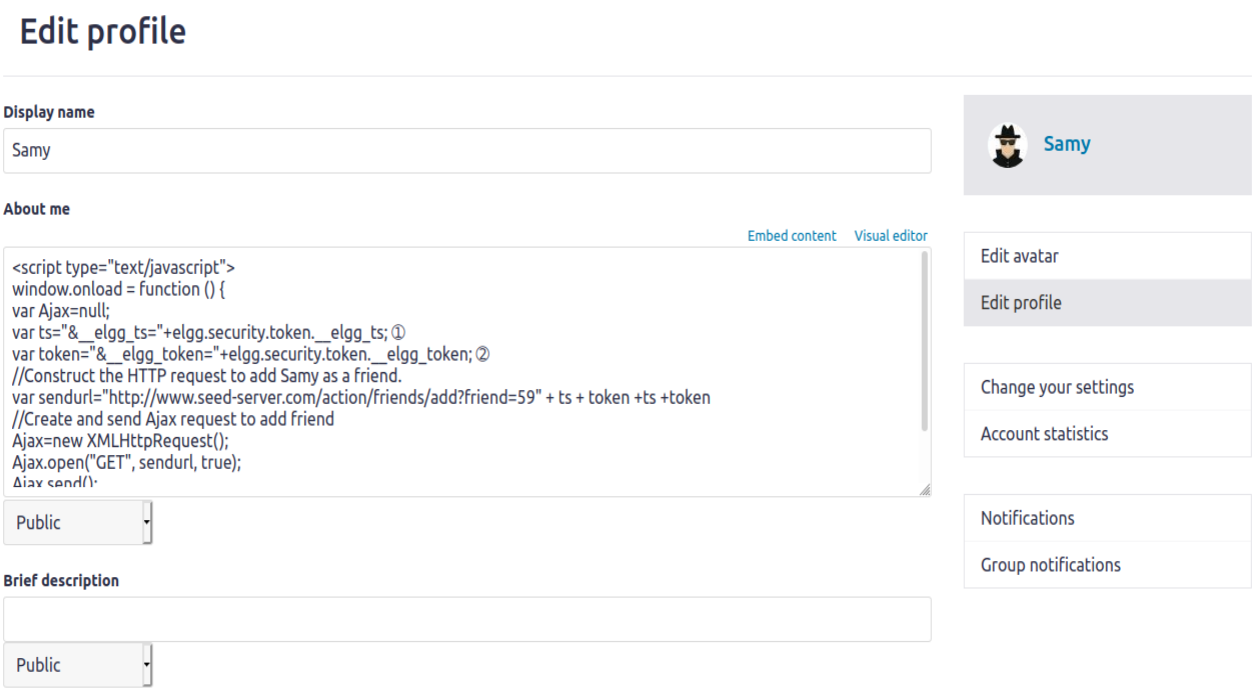
Task 4: Becoming the Victim's Friend

In this task, we need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. The objective of the attack is to add Samy as a friend to the victim.

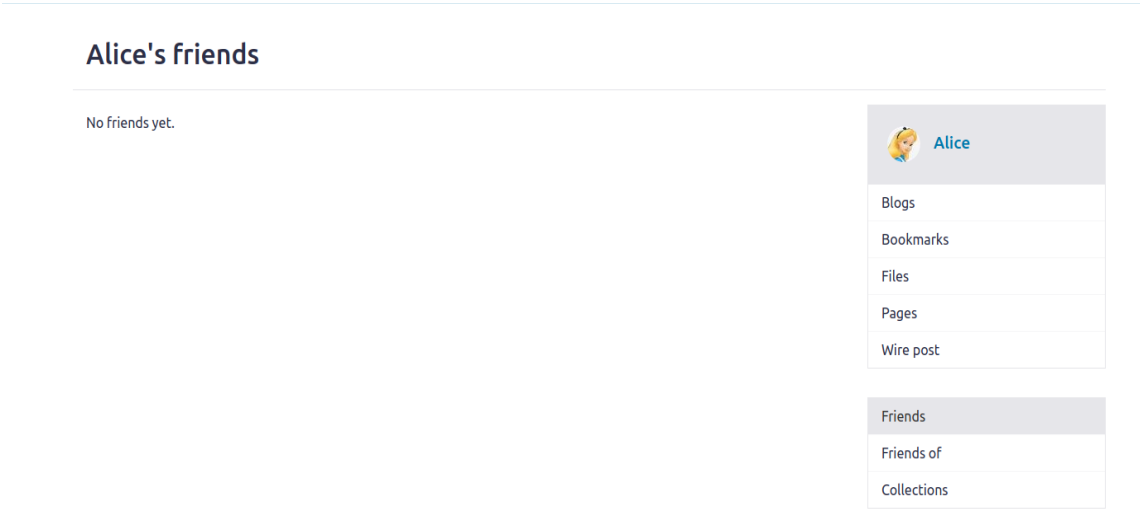
Observe the HTTP request by adding Samy as a friend from a Fake account (Boby's):



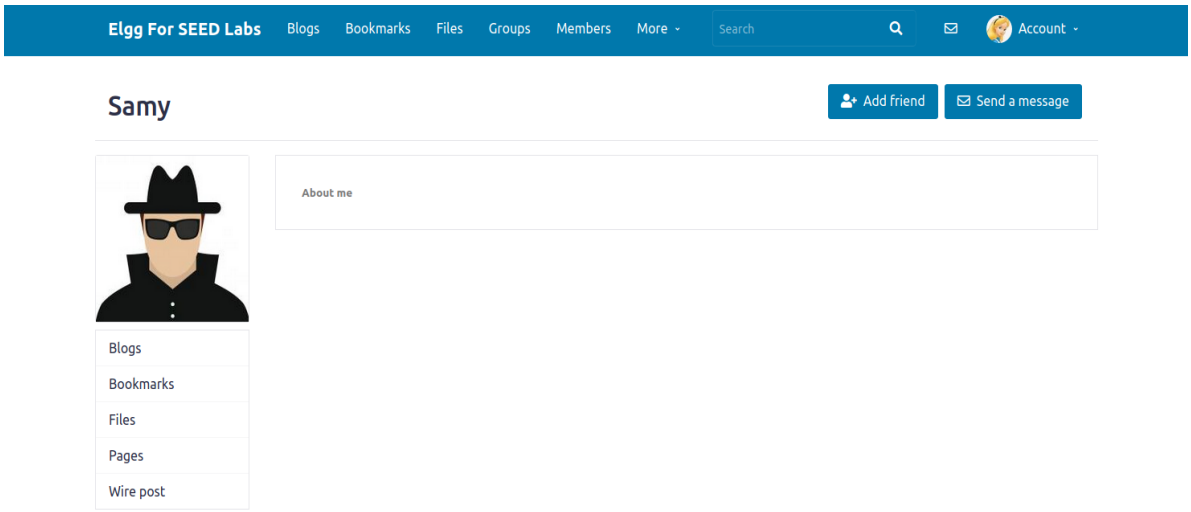
Inserting malicious script in attacker's (Samy's) profile:



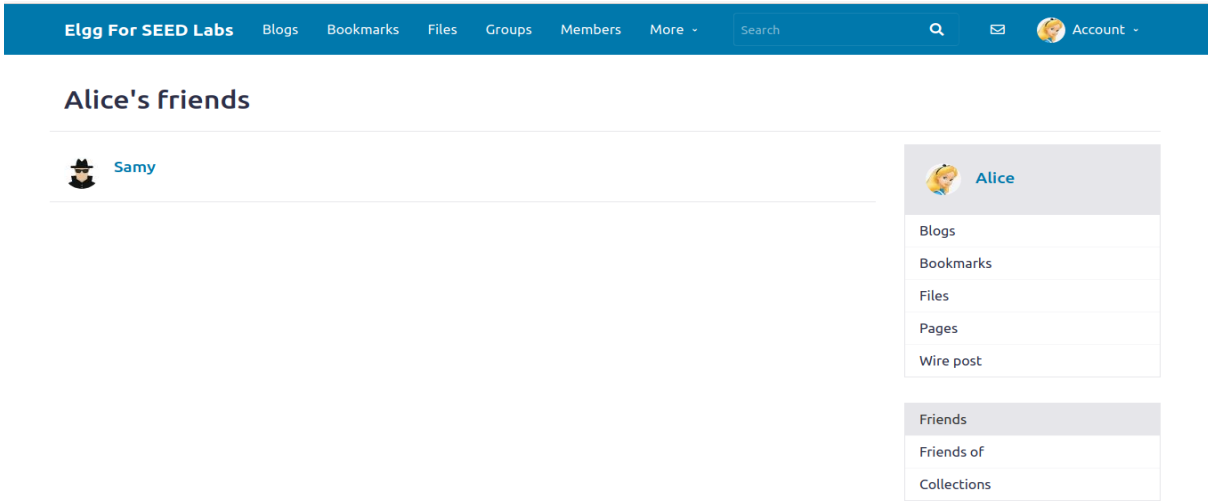
Alice has no friend before viewing the attacker's (Samy's) profile:



Alice viewing Samy's profile: This time the malicious script is loaded in Alice's profile and automatically Samy is added to her friend list.



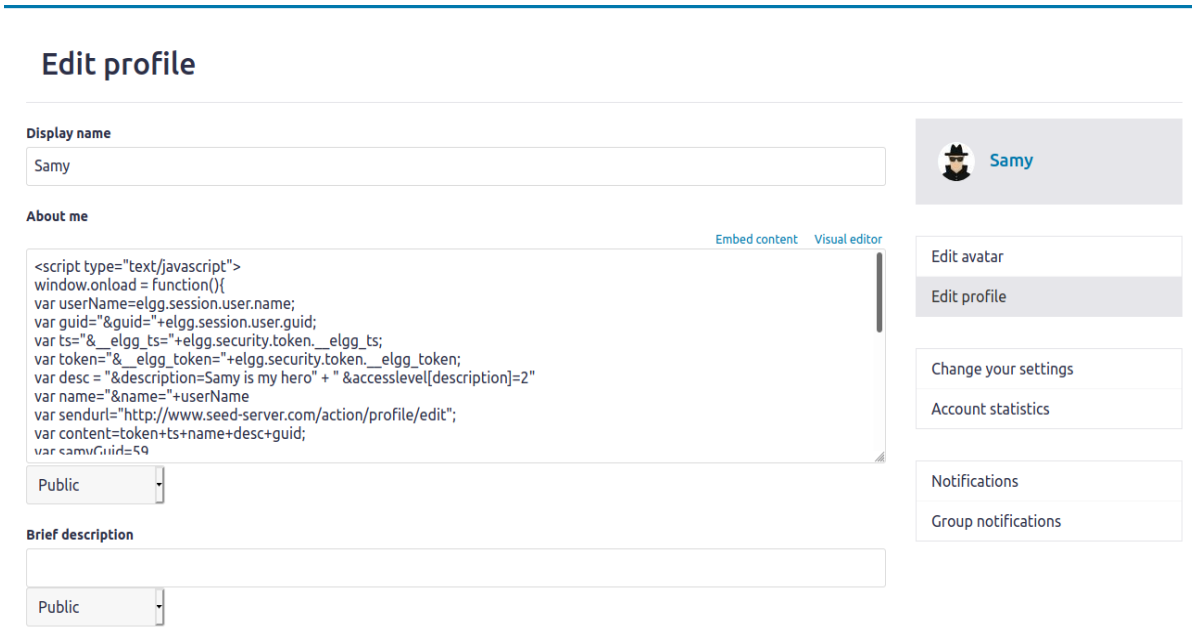
After that Alice's Friend list:



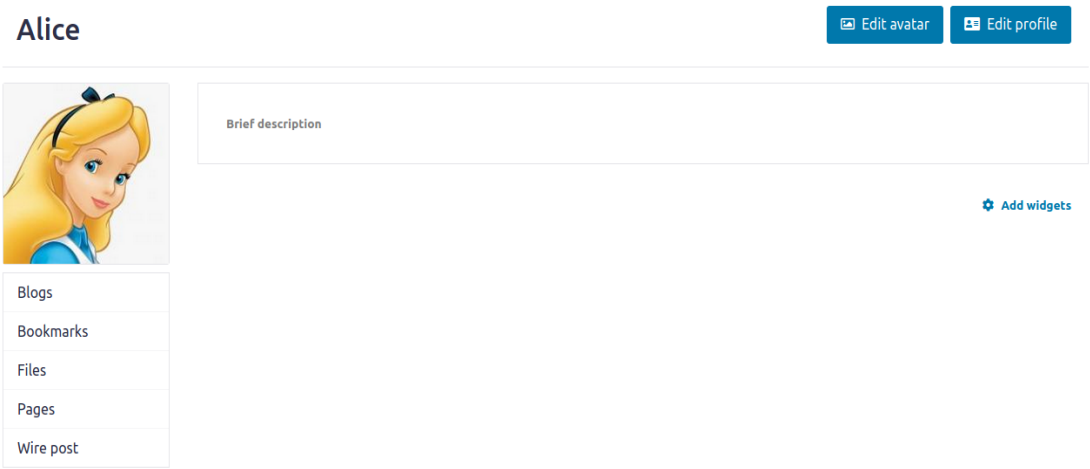
Task 5: Modifying the Victim's Profile

The objective of this task is to modify the victim's profile when the victim visits Samy's page. Specifically, modify the victim's "About Me" field.

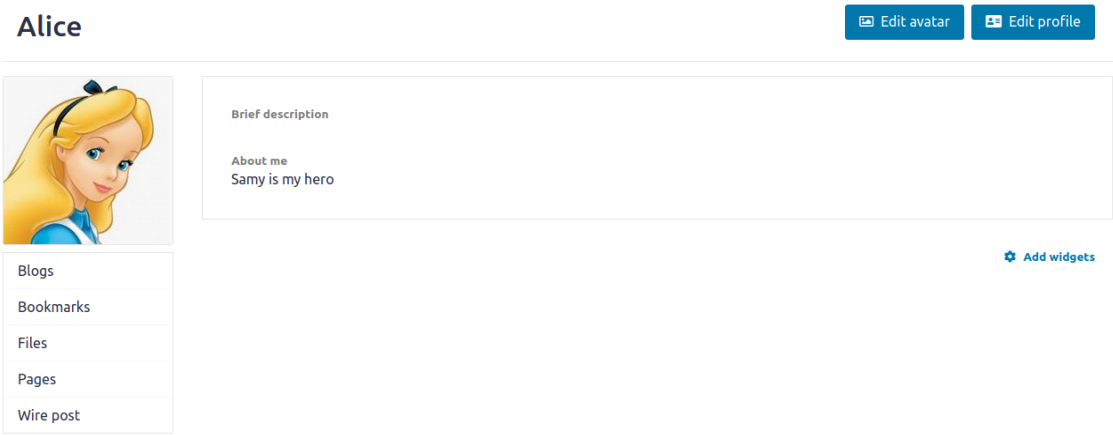
Inserting malicious script in attacker's (Samy's) profile:



Alice's profile before viewing Samy's profile:



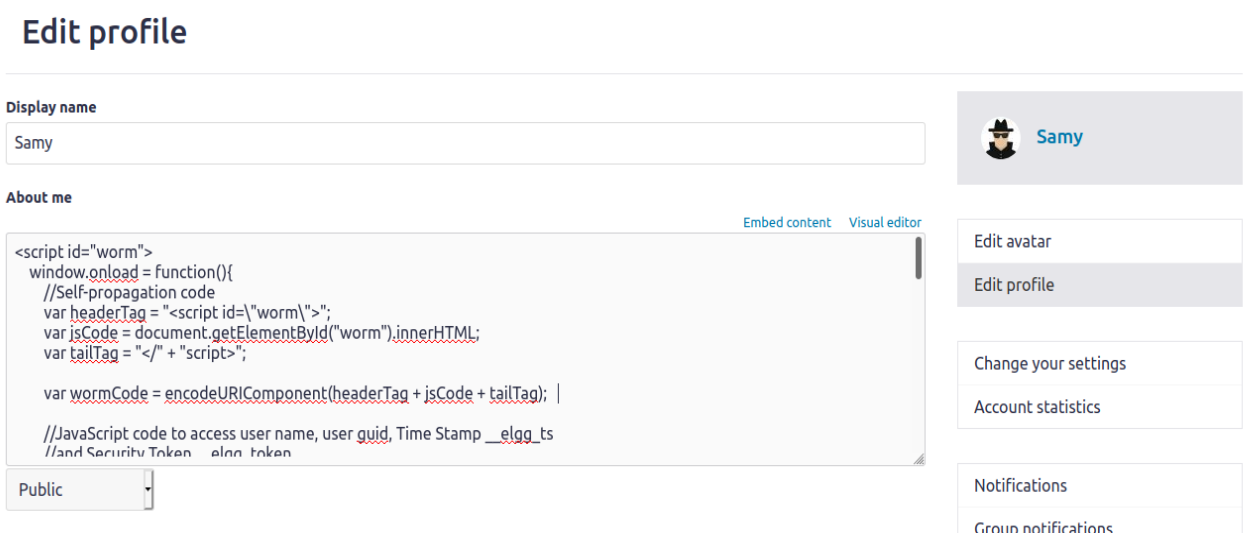
Alice's profile after viewing Samy's profile:



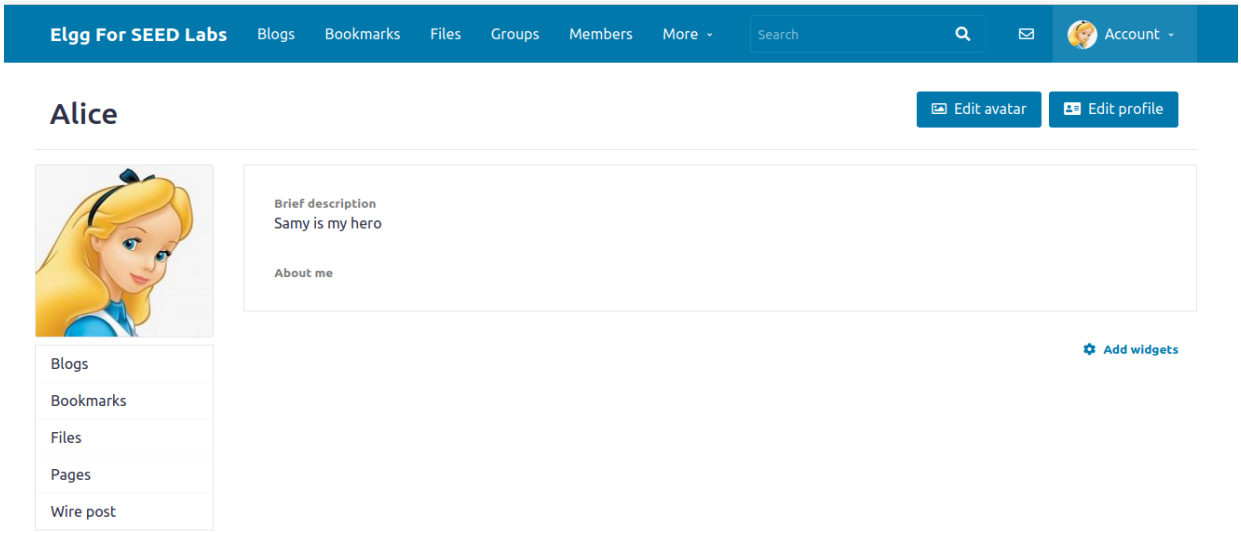
Task 6: Writing a Self-Propagating XSS Worm

To become a real worm, the malicious JavaScript program should be able to propagate itself. Namely, whenever some people view an infected profile, not only will their profiles be modified, the worm will also be propagated to their profiles, further affecting others who view these newly infected profiles.

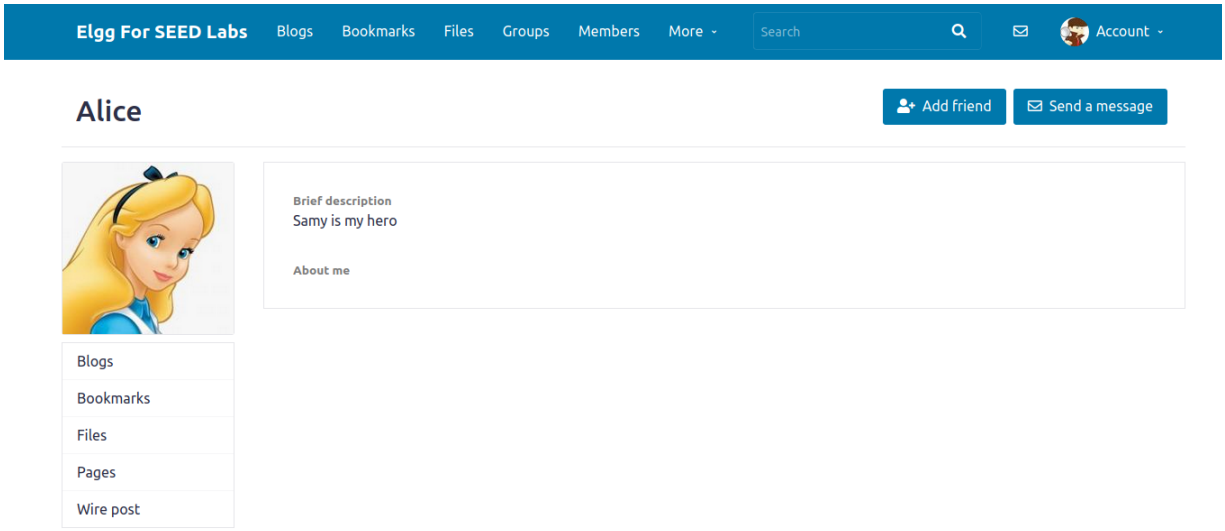
Inserting malicious script in attacker's (Samy's) profile:



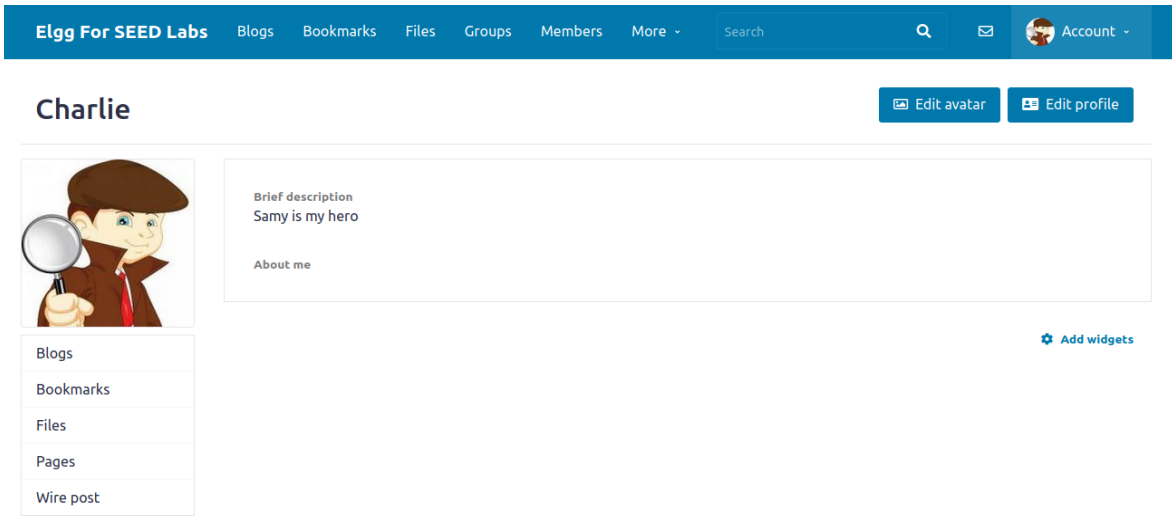
Alice's profile after viewing Samy's profile:



Charlie viewing Alice's profile:



Charlie's profile after viewing Alice's profile:

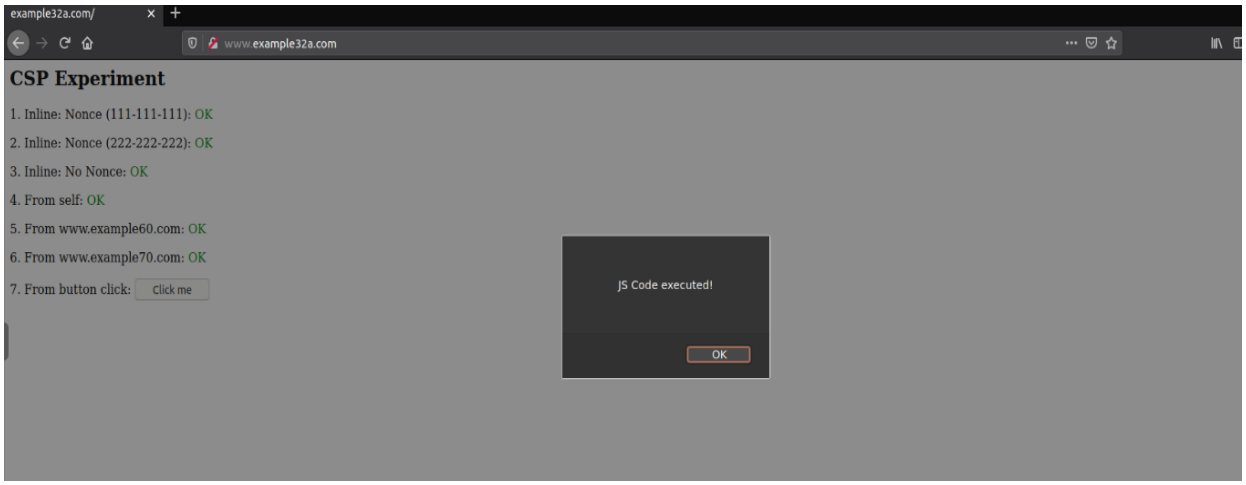


Task 7: Defeating XSS Attacks Using CSP

Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross-Site Scripting (XSS) and data injection attacks.

The example32(a|b|c) servers host the same web page index.html, which is used to demonstrate how the CSP policies work. On this page, there are six areas, area1 to area6. Initially, each area displays "Failed". The page also includes six pieces of JavaScript code, each trying to write "OK" to its corresponding area. If we can see OK in an area, that means, the JavaScript code corresponding to that area has been executed successfully; otherwise, we would see Failed. There is also a button on this page. If it is clicked, a message will pop up, if the underlying JavaScript code gets triggered.

In www.example32a.com , upon clicking the button, the JS code is executed.



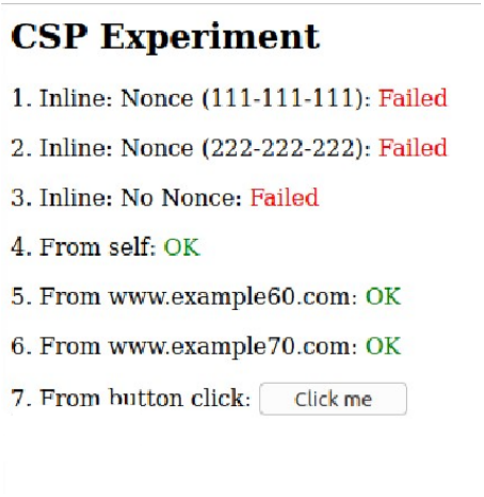
We also find that each area displays "Okay".

In www.example32b.com, from area1 to area6, area 1, 2, 3, and 5 displays "Failed"

We set up the websites and Apache will add the specified CSP header to all the response from this site.

```
# 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' *example60.com *.example70.com \
           "
</VirtualHost>
```

After executing dcbuild again, we run it, and the required area shows “OK”



In www.example32c.com, from area1 to area6, area 2, 3, and 5 displays "Failed".

However, instead of accessing index.html, the entry point of this site is phpindex.php, which is a PHP program. This program, listed below, adds a CSP header to the response generated from the program

```
<?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';?>
```

Now after executing dcbuild again, we run it, and the required area shows “OK”