# CS 656 LAB 5
# CSRF

**Notes:**

- This is an individual lab.
- The code and other answers you submit MUST be entirely your own work, and you are bound by the WSU Academic Integrity Policy (`https://www.wichita.edu/about/student_conduct/ai.php`). You MAY consult with other students about the conceptualization of the tasks and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone. You may consult published references, provided that you appropriately cite them in your reports and programs, as you would do in an academic paper.
- Read the entire document carefully before you start working on the lab.

GOOD LUCK!

## 1 Overview

The objective of this lab is to help students understand the Cross-Site Request Forgery (CSRF) attack. A CSRF attack involves a victim user, a trusted site, and a malicious site. The victim user holds an active session with a trusted site while visiting a malicious site. The malicious site injects an HTTP request for the trusted site into the victim user session, causing damages.

In this lab, students will be attacking a social networking web application using the CSRF attack. The open-source social networking application is called `Elgg`, which has already been installed in our VM. `Elgg` has countermeasures against CSRF, but we have turned them off for the purpose of this lab. This lab covers the following topics:

- Cross-Site Request Forgery attack
- CSRF countermeasures: Secret token
- HTTP GET and POST requests
- JavaScript

**Lab Environment.** This lab has been tested on Ubuntu 20.04. You will be given access to the remote Ubuntu environment with necessary packages installed. You may also run your own virtual machine by downloading it from the SEED labs website (see the details here: `https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md`).

## 2 Submission

Submit a PDF document with your answers to the questions in this lab. Your report should have a subheading for each question, and your answers should be inside the corresponding subheading. If applicable, list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

**Note:** *Your report must contain your Name (Lastname, Firstname) and WSU ID. Use the following format for your report's PDF filename:* `lab5report_YOUR_WSU_ID.pdf`. *There is a 10% reduction*

*of points if your report does not follow the correct filename format and/or missing name/ID inside the document.*

## 2.1 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. 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.

**The Elgg container.** We host the Elgg web application using the Apache web server. The website setup is included in `apache_elgg.conf` inside the Elgg image folder. The configuration specifies the URL for the website and the folder where the web application code is stored.

```
<VirtualHost *:80>
DocumentRoot /var/www/elgg
ServerName   www.seed-server.com
<Directory /var/www/elgg>
Options FollowSymlinks
AllowOverride All
Require all granted
</Directory>
</VirtualHost>
```

**The Attacker container.** We use another container (`10.9.0.105`) for the attacker machine, which hosts a malicious website. The Apache configuration for this website is listed in the following:

```
<VirtualHost *:80>
DocumentRoot /var/www/attacker
ServerName   www.attacker32.com
</VirtualHost>
```

Since we need to create web pages inside this container, for convenience, as well as for keeping the pages we have created, we mounted a folder (`labsetup/attacker` on the hosting VM) to the container's `/var/www/attacker` folder, which is the `DocumentRoot` folder in our Apache configuration. Therefore, the web pages we put inside the `attacker` folder on the VM will be hosted by the attacker's website. We have already placed some code skeletons inside this folder.

**DNS configuration.** We access the Elgg website, the attacker website, and the defense site using their respective URLs. We need to add the following entries to the `/etc/hosts` file, so these hostnames are mapped to their corresponding IP addresses. You need to use the root privilege to change this file (using `sudo`). It should be noted that these names might have already been added to the file due to some other labs. If they are mapped to different IP addresses, the old entries must be removed.

```
10.9.0.5        www.seed-server.com
10.9.0.105      www.attacker32.com
```

**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` directory, 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     |  seedelgg
alice     |  seedalice
boby      |  seedboby
charlie   |  seedcharlie
samy      |  seedsamy
----------------------------
```

# 3 Lab Tasks

## 3.1 Task 1: Observing HTTP Request [10 Points]

In Cross-Site Request Forget attacks, we need to forge HTTP requests. Therefore, we need to know what a legitimate HTTP request looks like and what parameters it uses, etc. We can use a Firefox add-on called `"HTTP Header Live"` for this purpose. The goal of this task is to get familiar with this tool. Instructions on how to use this tool is given in the Guideline section (see §4.1). Use this tool to capture an HTTP GET request and an HTTP POST request in Elgg.

**Deliverable.** Include a screen shot of the output of `HTTP Header Live` showing *(a)* a GET request and *(b)* a POST request in the `www.seed-server.com` website. Identify the parameters used in each of the requests that you included and label them. Clearly mark task numbers in your report.

## 3.2 Task 2: CSRF Attack using GET Request [30 Points]

In this task, we need two people in the Elgg social network: Alice and Samy. Samy wants to become a friend to Alice, but Alice refuses to add him to her Elgg friend list. Samy decides to use the CSRF attack to achieve his goal. He sends Alice an URL (via an email or a posting in Elgg); Alice, curious about it, clicks on the URL, which leads her to Samy's web site: `www.attacker32.com`. Pretend that you are Samy, describe how you can construct the content of the web page, so as soon as Alice visits the web page, Samy is added to the friend list of Alice (assuming Alice has an active session with Elgg).

To add a friend to the victim, we need to identify what the legitimate Add-Friend HTTP request (a GET request) looks like. We can use the `"HTTP Header Live"` Tool to do the investigation. In this task, you are not allowed to write JavaScript code to launch the CSRF attack. Your job is to make the attack successful

as soon as Alice visits the web page, without even making any click on the page (hint: you can use the `img` tag, which automatically triggers an HTTP GET request).

Elgg has implemented a countermeasure to defend against CSRF attacks. In Add-Friend HTTP requests, you may notice that each request includes two weird-looking parameters, `__elgg_ts` and `__elgg_token`. These parameters are used by the countermeasure, so if they do not contain correct values, the request will not be accepted by Elgg. We have disabled the countermeasure for this lab, so there is no need to include these two parameters in the forged requests.

**Deliverables.** Clearly mark task numbers in your report.

(2.a) Submit your HTML code used to design the web page that launches the attack described above.
(2.b) Submit the log of the `HTTP Header Live` and identify the HTTP request sent by your malicious web page to the Elgg website that resulted in Samy being added to Alice's friend list.

### 3.3 Task 3: CSRF Attack using POST Request [30 Points]

After adding himself to Alice's friend list, Samy wants to do something more. He wants Alice to say "Samy is my Hero" in her profile, so everybody knows about that. Alice does not like Samy, let alone putting that statement in her profile. Samy plans to use a CSRF attack to achieve that goal. That is the purpose of this task. One way to do the attack is to post a message to Alice's Elgg account, hoping that Alice will click the URL inside the message. This URL will lead Alice to your (i.e., Samy's) malicious web site `www.attacker32.com`, where you can launch the CSRF attack.

The objective of your attack is to modify the victim's profile. In particular, the attacker needs to forge a request to modify the profile information of the victim user of Elgg. Allowing users to modify their profiles is a feature of Elgg. If users want to modify their profiles, they go to the profile page of Elgg, fill out a form, and then submit the form—sending a POST request—to the server-side script `/profile/edit.php`, which processes the request and does the profile modification.

The server-side script `edit.php` accepts both GET and POST requests, so you can use the same trick as that in Task 2 to achieve the attack. However, in this task, you are required to use the POST request. Namely, attackers (you) need to forge an HTTP POST request from the victim's browser, when the victim is visiting their malicious site. Attackers need to know the structure of such a request. You can observe the structure of the request, i.e., the parameters of the request, by making some modifications to the profile and monitoring the request using the "`HTTP Header Live`" tool. You may see something similar to the following. Unlike HTTP `GET` requests, which append parameters to the URL strings, the parameters of HTTP `POST` requests are included in the HTTP message body (see the contents between the two ☆ symbols):

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

POST /action/profile/edit HTTP/1.1
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) ...
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seed-server.com/profile/elgguser1/edit
Cookie: Elgg=p0dci8baqrl4i2ipv2mio3po05
Connection: keep-alive
```

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 642
__elgg_token=fc98784a9fbd02b68682bbb0e75b428b&__elgg_ts=1403464813  ✰
&name=elgguser1&description=%3Cp%3Iamelgguser1%3C%2Fp%3E
&accesslevel%5Bdescription%5D=2&briefdescription= Iamelgguser1
&accesslevel%5Bbriefdescription%5D=2&location=US
......                                                              ✰
```

After understanding the structure of the request, you need to be able to generate the request from your attacking web page using JavaScript code. To help you write such a JavaScript program, we provide a sample code in the following code fence. You can use this sample code to construct your malicious web site for the CSRF attacks. This is only a sample code, and you need to modify it to make it work for your attack.

```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">

function forge_post()
{
   var fields;

   // The following are form entries need to be filled out by attackers.
   // The entries are made hidden, so the victim won't be able to see them.
   fields += "<input type='hidden' name='name' value='****'>";
   fields += "<input type='hidden' name='briefdescription' value='****'>";
   fields += "<input type='hidden' name='accesslevel[briefdescription]'
   value='2'>";                          ①
   fields += "<input type='hidden' name='guid' value='****'>";

   // Create a <form> element.
   var p = document.createElement("form");

   // Construct the form
   p.action = "http://www.example.com";
   p.innerHTML = fields;
   p.method = "post";

   // Append the form to the current page.
   document.body.appendChild(p);

   // Submit the form
   p.submit();
}


// Invoke forge_post() after the page is loaded.
window.onload = function() { forge_post();}
</script>
</body>
</html>
```

In Line ①, the value `2` sets the access level of a field to public. This is needed, otherwise, the access level

will be set by default to private, so others cannot see this field. It should be noted that when copy-and-pasting the above code from a PDF file, the single quote character in the program may become something else (but still looks like a single quote). That will cause syntax errors. Replacing all the single quote symbols with the one typed from your keyboard will fix those errors.

**Deliverables.** Clearly mark task numbers in your report.

(3.a) Submit the HTML file used to launch the attack described above.
(3.b) Show the part of the `HTTP Header Live` log that contains the POST request generated by your script, and identify it.
(3.c) Answer the following questions:

3.c.i. The forged HTTP request needs Alice's user id (guid) to work properly. If Samy targets Alice specifically, before the attack, he can find ways to get Alice's user id. Samy does not know Alice's Elgg password, so he cannot log into Alice's account to get the information. Describe how Samy can solve this problem.

3.c.ii. If Samy would like to launch the attack to anybody who visits his malicious web page. In this case, he does not know who is visiting the web page beforehand. Can he still launch the CSRF attack to modify the victim's Elgg profile?

### 3.4 Task 4: Enabling Elgg's Countermeasure [30 Points]

CSRF is not difficult to defend against. Initially, most applications put a secret token in their pages, and by checking whether the token is present in the request or not, they can tell whether a request is a same-site request or a cross-site request. This is called *secret token* approach. To defend against CSRF attacks, web applications can embed a secret token in their pages. All the requests coming from these pages must carry this token, or they will be considered as a cross-site request, and will not have the same privilege as the same-site requests. Attacker will not be able to get this secret token, so their requests are easily identified as cross-site requests.

Elgg uses this secret-token approach as its built-in countermeasures to defend against CSRF attacks. We have disabled the countermeasures to make the attack work. Elgg embeds two parameters `__elgg_ts` and `__elgg_token` in the request. The two parameters are added to the HTTP message body for the POST requests and to the URL string for the HTTP GET requests. The server will validate them before processing a request.

**Embedding secret token and timestamp to web pages.** Elgg adds security token and timestamp to all the HTTP requests. The following HTML code is present in all the forms where user action is required. These are two hidden fields; when the form is submitted, these two hidden parameters are added to the request:

```
<input type = "hidden" name = "__elgg_ts" value = "" />
<input type = "hidden" name = "__elgg_token" value = "" />
```

Elgg also assign the values of the security token and timestamp to JavaScript variables, so they can be easily accessed by the JavaScript code on the same page.

```
elgg.security.token.__elgg_ts;
elgg.security.token.__elgg_token;
```

The secret token and timestamp are added to Elgg's web pages by the `vendor/elgg/elgg/views/default/input/securitytoken.php` module. The code snippet below shows how they are dynamically added to web pages.

```
$ts = time();
$token = elgg()->csrf->generateActionToken($ts);

echo elgg_view('input/hidden', ['name' => '__elgg_token', 'value' =>
 $token]);
echo elgg_view('input/hidden', ['name' => '__elgg_ts', 'value' => $ts]);
```

**Secret token generation.** Elgg's security token is a hash value (md5 message digest) of the site secret value (retrieved from database), timestamp, user session ID and random generated session string. The code below shows the secret token generation in Elgg (in `vendor/elgg/elgg/engine/classes/Elgg/Security/Csrf.php`).

```
/**
* Generate a token from a session token (specifying the user),
* the timestamp, and the site key.
*/
public function generateActionToken($timestamp, $session_token = '') {
   if (!$session_token) {
      $session_token = $this->session->get('__elgg_session');
      if (!$session_token) {
         return false;
      }
   }

   return $this->hmac
   ->getHmac([(int) $timestamp, $session_token], 'md5')
   ->getToken();
}
```

**Secret token validation.** The elgg web application validates the generated token and timestamp to defend against the CSRF attack. Every user action calls the `validate` function inside `Csrf.php`, and this function validates the tokens. If tokens are not present or invalid, the action will be denied and the user will be redirected. In our setup, we added a `return` at the beginning of this function, essentially disabling the validation.

```
public function validate(Request $request) {
   return; // Added for SEED Labs (disabling the CSRF countermeasure)

   $token = $request->getParam('__elgg_token');
   $ts = $request->getParam('__elgg_ts');
   ... (code omitted) ...
}
```

**Turn on the countermeasure.** To turn on the countermeasure, get into the Elgg container, go to the `/var/www/elgg/vendor/elgg/elgg/engine/classes/Elgg/Security` folder, remove the

`return` statement from `Csrf.php`. A simple editor called `nano` is available from inside the container.

**It should be noted (important)** that when we launch the edit-profile attack while the countermeasure is enabled, the failed attempt will cause the attacker's page to be reloaded, which will trigger the forged POST request again. This will lead to another failed attempt, so the page will be reloaded again and another forged POST request will be sent out. This endless loop will slow down your computer. Therefore, after verifying that the attack failed, kill the tab to stop the endless loop.

**Deliverables.** Clearly mark task numbers in your report.

(4.a) After turning on the countermeasure as described above, try both the GET and POST CSRF attacks again, and describe your observations.

(4.b) Identify the secret tokens in the HTTP request in the relevant section of the `HTTP Header Live` tool log.

(4.c) Explain why the attacker cannot send these secret tokens in the CSRF attack and what prevents them from finding out the secret tokens from the web page?

# 4 Guidelines

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

The instruction on how to enable and use "`HTTP Header Live`" 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).

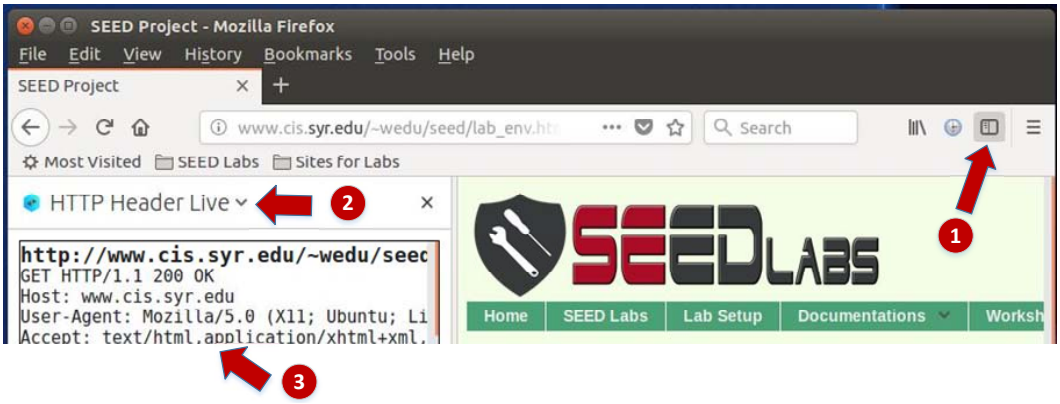



Figure 1: Enable the HTTP Header Live Add-on

## 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.

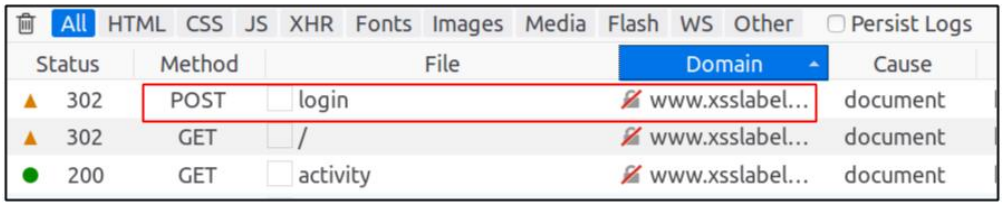



Figure 2: HTTP Request in Web Developer Network Tool

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).

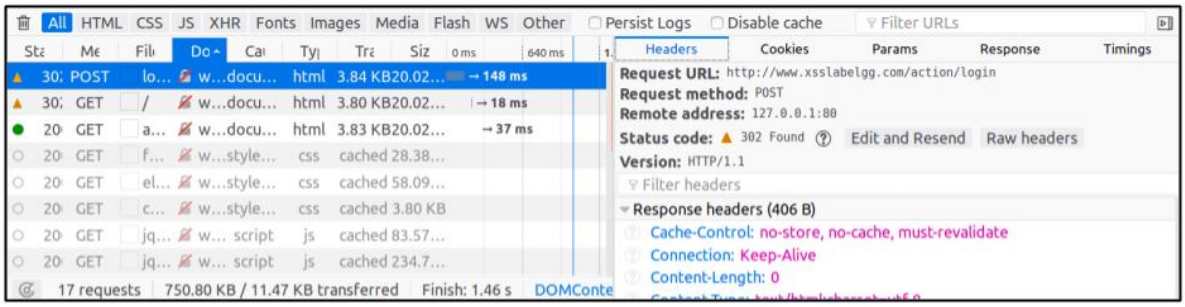


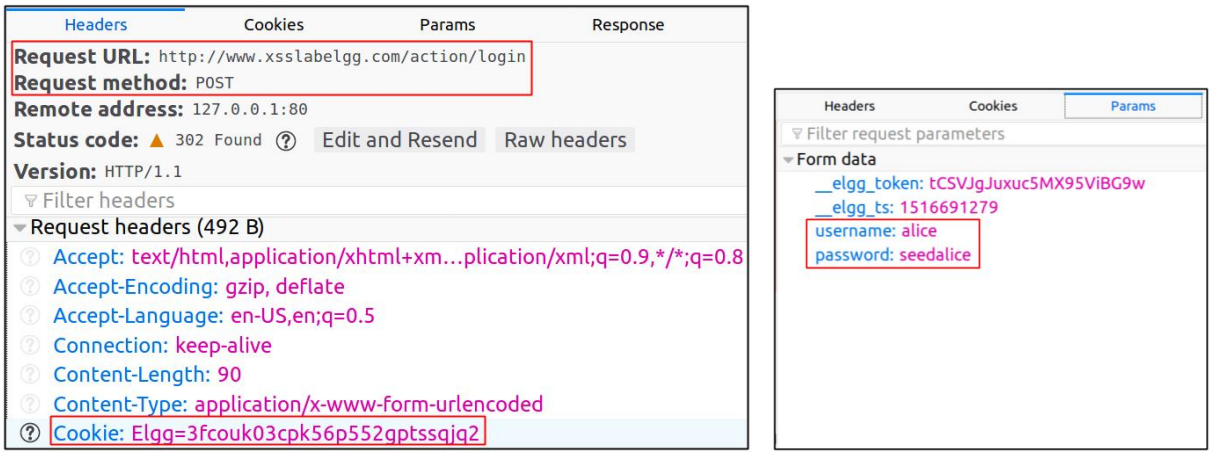

Figure 3: HTTP Request and Request Details in Two Panes

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.

**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:




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

Figure 4: HTTP Headers and Parameters

```
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.

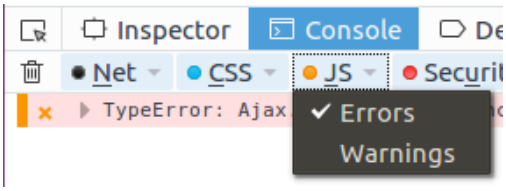



Figure 5: Debugging JavaScript Code (1)

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.

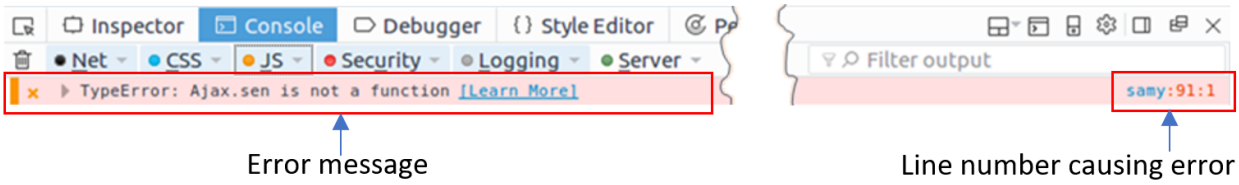



Figure 6: Debugging JavaScript Code (2)

Copyright © 2022 Monowar Hasan.
This document is adopted from Dr. Sergio Salinas Monroy and orignally based on the SEED Labs developed by Dr. Wenliang Du. The contents of this document are licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. A human-readable summary of (and not a substitute for) the license is the following: You are free to copy and redistribute the material in any medium or format. You must give appropriate credit. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You may not use the material for commercial purposes.