

Cross-Site Scripting Attack (CS915)

Overview

Cross-site scripting (XSS) is a type of vulnerability commonly found in web applications. XSS attacks occur when an attacker **uses a web application to send malicious code**, generally in the form of a browser-side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere **a web application uses input from a user within the output it generates without validating or encoding it**.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page¹.

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 networks, 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, you 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.

Lab environment setup

- Download the lab setup files from [here](#) (save them in any folder under /home/seed **EXCEPT** the shared folder if you use VirtualBox).
- Read the manual on docker [here](#) (for background learning)
- Download files from [here](#) (containing example scripts for the attack)

This lab is based on the [Cross-Site Scripting \(XSS\) Attack lab](#) with adaptations.

¹ <https://owasp.org/www-community/attacks/xss/>

DNS Setup

We need to map the name of the web server to the 10.9.0.5 IP address hosted by the container. 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
```

As an editor, you may use vi, vim or nano that are available on the SEED VM. In the example below, we use vi.

```
$ sudo vi /etc/hosts
```

Container Setup

enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment. A detailed explanation of the content in this file and all the involved Dockerfile can be found in the user manual, which is linked to the website of this lab.

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 the SEED VM.

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

// 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.

```
$ 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 (in the example above, it's sufficient to type "docksh 96" for "9652715c8e0a")

Build the containers

```
[06/02/21]seed@VM:~/.../Labsetup$ dcbuild
Building elgg
Step 1/11 : FROM handsonsecurity/seed-elgg:original
original: Pulling from handsonsecurity/seed-elgg
da7391352a9b: Already exists
14428a6d4bcd: Already exists
2c2d948710f2: Already exists
d801bb9d0b6c: Pull complete
9c11a94ddf64: Pull complete
81f03e4cea1b: Pull complete
0ba9335b8768: Pull complete
8ba195fb6798: Pull complete
264df06c23d3: Pull complete
Digest: sha256:728dc5e7de5a11bea1b741f8ec59ded392bbeb9eb2fb425b8750773ccda8f706
Status: Downloaded newer image for handsonsecurity/seed-elgg:original
--> e7f441caa931
```

Start the containers

```
[06/02/21]seed@VM:~/.../Labsetup$ dcup
Creating network "net-10.9.0.0" with the default driver
Creating elgg-10.9.0.5 ... done
Creating mysql-10.9.0.6 ... done
Attaching to mysql-10.9.0.6, elgg-10.9.0.5
```

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).

MySQL database



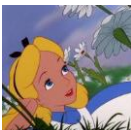

Containers are usually disposable, so once it is destroyed, all the data inside the containers are lost. For this lab, we do want to keep the data in the MySQL database, so we do not lose our work when we shutdown our container. To achieve this, we have mounted the mysql data folder on the host machine (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 is kept. If you do want to start from a clean database, you can remove this folder:

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

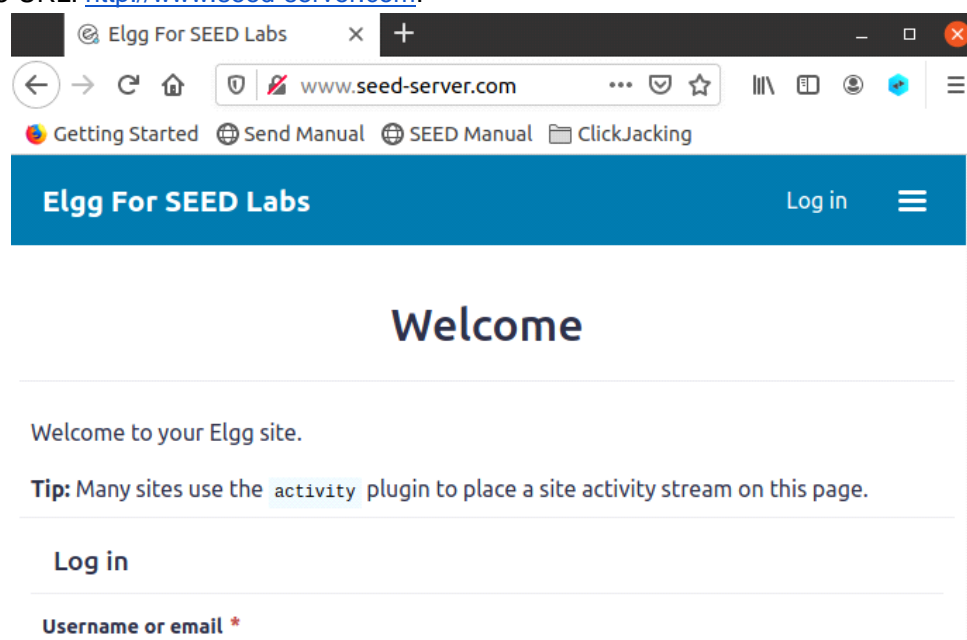
User accounts

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

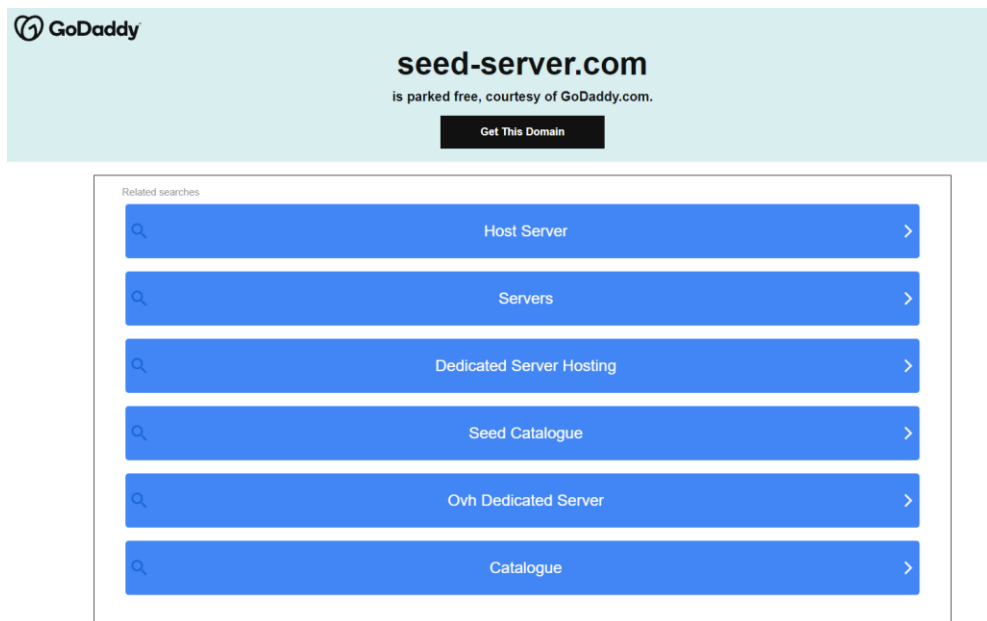
Attacker: Samy Password: seedsamy 	Investigator: Charlie Password: seedcharlie 
Victim: Alice Password: seedalice 	Victim: Bobby Password: seedbobby 

Final testing

After setting up the DNS and containers, you should see the page below in your browser using the URL: <http://www.seed-server.com>.



Note: if you see the below page rather than the above page, maybe you need to clean your browser cache and try again.



Task Overview

- Task 1: Display a Simple Alert Message
- Task 2: Add Samy to Alice's Friend List
- Task 3: Add "Samy is my hero" to Alice's Profile
- Task 4: Create a Self-Propagating XSS Worm
- Task 5: Countermeasures

Task 1: Display a Simple Alert Message

Objective:

The objective of this task is to embed a JavaScript program in Samy 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.

Steps

1. Log into Samy's Elgg account (password: seedsamy), and go to the profile page

Samy

 Edit avatar

 Edit profile

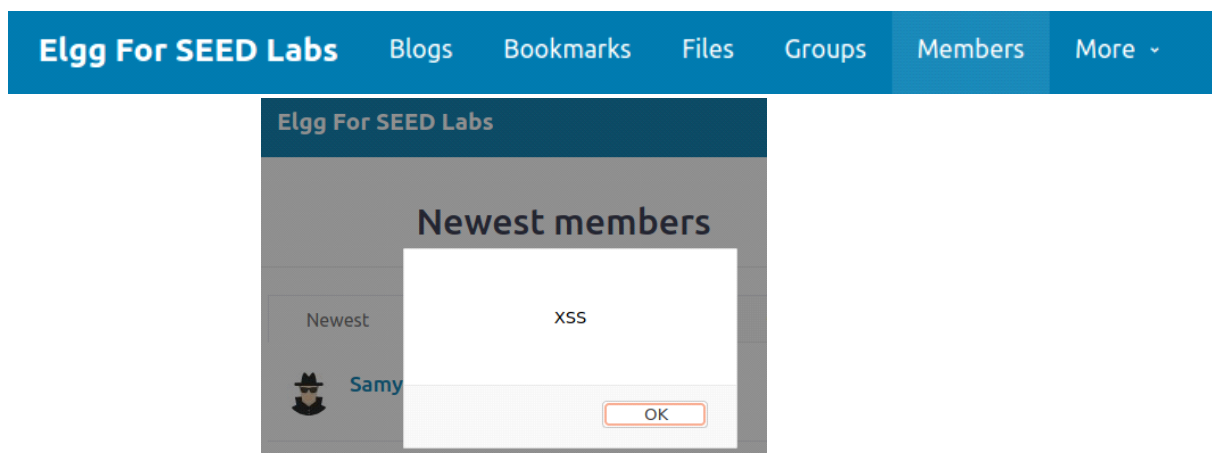
2. Inject a script into the profile's "Brief description" field (use **"Edit HTML"**, not the rich text mode)

Brief description

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

Public

3. Log out
4. Log into Alice's account, and go to the "Members" Tab. Find Samy's profile. You should expect to see the following alert message.



5. Log out

Task 2: Add Samy to Alice's Friend List

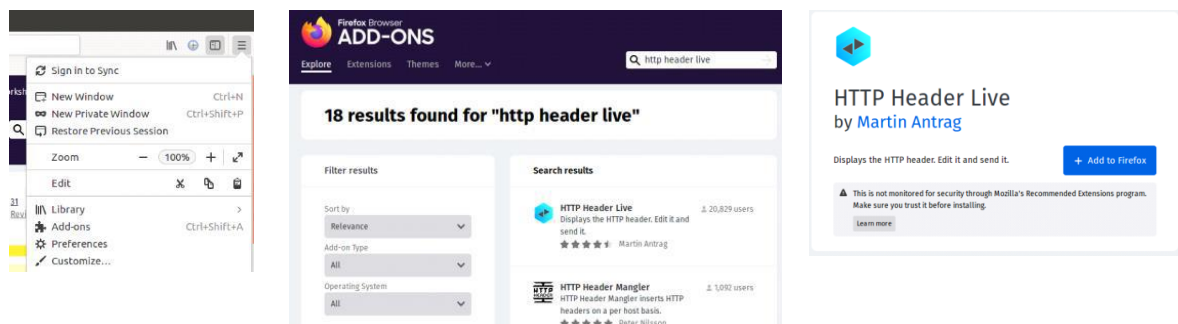
In this task, we need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. The objective of the attack is to add Samy as a friend to the victim.

The Challenge



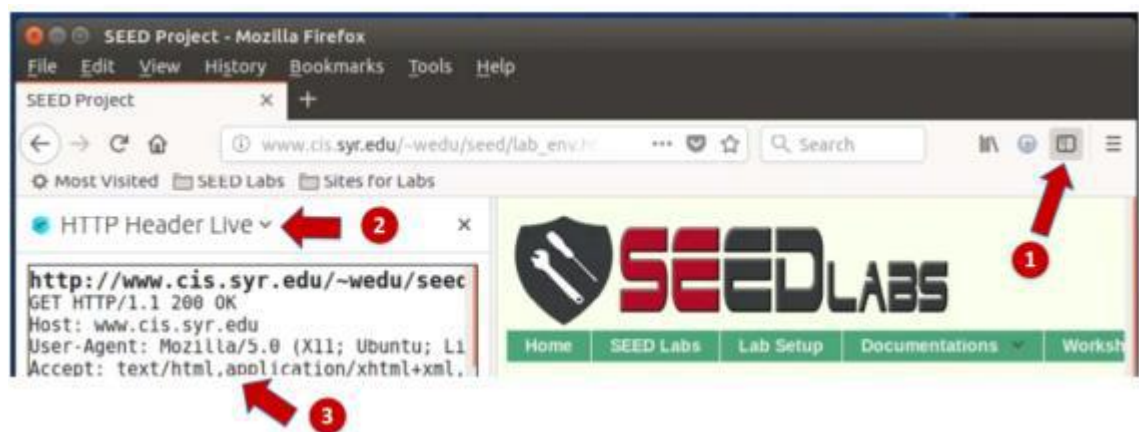
To add a friend for the victim, we should first find out how a legitimate user adds a friend in Elgg. More specifically, we need to figure out what is sent to the server when a user adds a friend. Firefox's **HTTP Header Live** add-on can help us get the information. It can display the contents of any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

Update "HTTP Header Live" Add-on



Investigation

1. Open HTTP headers live



2. Log into Charlie's Elgg account
3. Add Samy to the friend list.

4. Find the corresponding HTTP request in HTTP Header Live

HTTP Request for Adding Friends (Elgg)



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

```
<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>
```

Hint: fill in the sendurl according to the format of the url string captured by the HTTP Header Live add-on. A sample script is provided in the lab files in case you need help. But you're encouraged to fill in the sendurl by yourself before looking at the sample script.

Questions (provide answers in the post-lab assignment).

- **Question 1:** Explain the purpose of (1) and (2), why are they needed?
- **Question 2:** If the Elgg application only provides the Rich Text Editor mode for the "About Me" field, i.e., you cannot switch to the Text mode ("Edit HTML"), can you still launch a successful attack?

Debugging JavaScript code

If you encounter errors in the JavaScript program, you can use the following method.

The JavaScript code in the screenshot below has an error

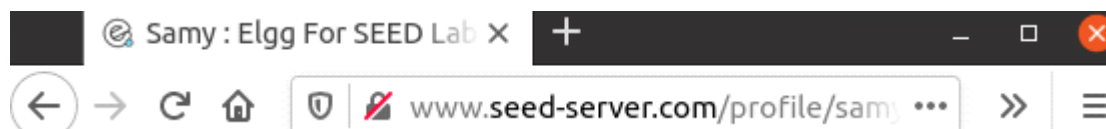
In the example below, alert is not spelt correctly

Brief description

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

Public

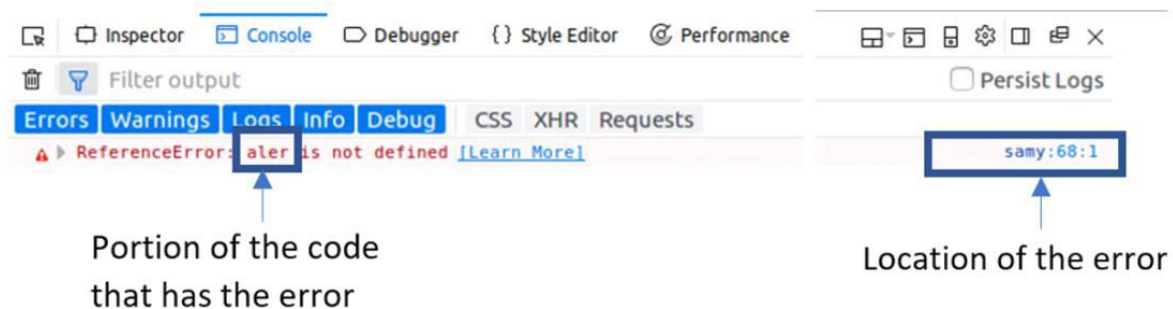
Go to Web Console



Web Developer > Web Console



If there is an error, you can easily find where it is



Let's do the attack

In Attacker's Account (use the code in [add_friend.js](#))

- 1) Log into Samy's Elgg account.
- 2) Go to Profile, click "[Edit HTML](#)" to enter the plaintext mode (we can't use the rich text mode), then inject malicious code into the "[About Me](#)" text field.
- 3) Log out.

In victim's account

- 1) Log into Alice's Elgg account, check Alice's friend list.
- 2) Visit Samy's profile.
- 3) Check Alice's friend list again.
- 4) Log out.

Task 3: Modify Alice's Profile

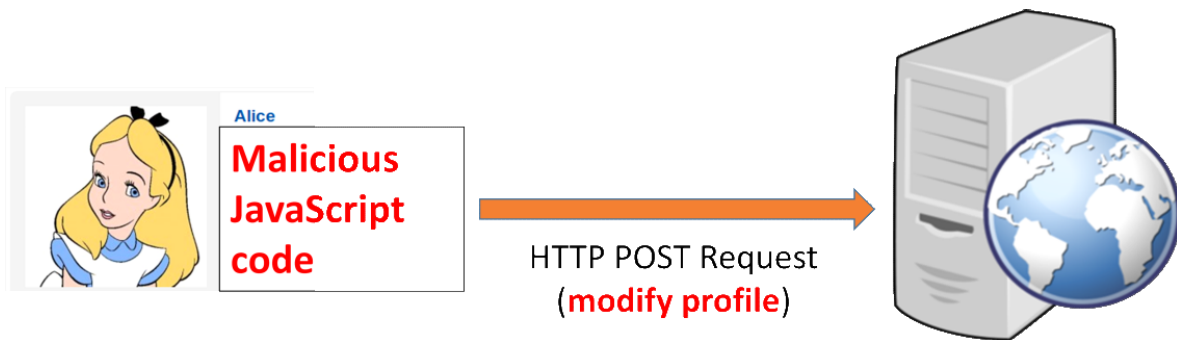
The objective of this task is to modify the victim's profile when the victim visits Samy's page. Specifically, modify the victim's "About Me" field. We will write an XSS worm to complete the task.



Modify Alice's Profile to show
"Samy is my Hero"

Investigation

Similar to the previous task, we need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. To modify the profile, we should first find out how a legitimate user edits or modifies 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.



Investigation

- 1) Open HTTP Headers Live
- 2) Log into Charlie's Elgg account
- 3) Modify Profile
- 4) Find the corresponding HTTP request in HTTP Header Live

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>
```

Questions.

- **Question 3:** Why do we need Line (1)? If we do not add this check, can the attack be successful? How come we do not have such a check in the add-friend attack ([add_friend.js](#))?

Let's do the attack

In attacker's account (use the code in [edit_profile.js](#))

- 1) Log into Samy's Elgg account.
- 2) Go to Profile, click "**Edit HTML**" to enter the **plaintext** mode, then inject malicious code into the "**About Me**" text field.

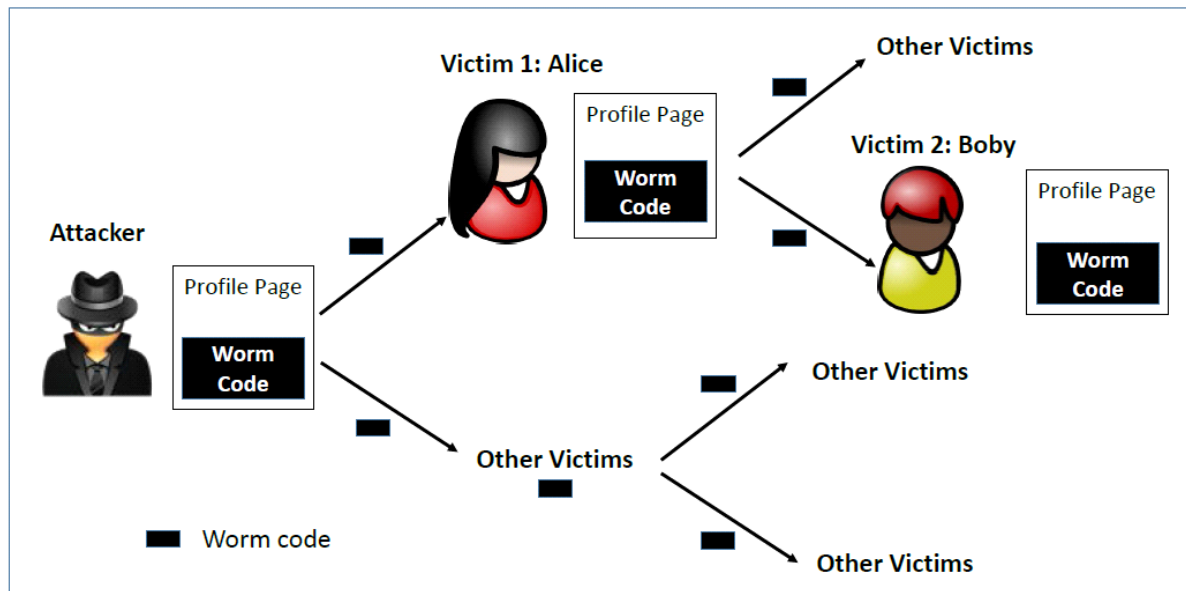


- 3) Log out

In Victim's account

- 1) Log into Alice's Elgg account, check Alice's profile.
- 2) Visit Samy's profile.
- 3) Check Alice's profile again.
- 4) Log out

Task 4: write a self-propagating 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 adds 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. In general, there are two approaches, a line approach (linking to an external JavaScript program) and a DOM approach (retrieving a copy of itself from the web page). In this task, we will use 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 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 is 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.

Let's do the attack

1. Place the worm in **Samy's** profile page (use the code in [self_propagation.js](#))
2. Log into **Alice's** account, and check her profile again
3. Visit **Samy's** profile, and check Alice's profile again
4. Log out
5. Log into **Boby's** Elgg account, check his profile
6. Visit **Alice's** profile, and check Bobby's profile again

Task 5: Defeating XSS 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 links 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.

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.

The example32a.com, example32b.com, example32c.com servers host the same web page index.html, which is used to demonstrate how the CSP policies work. You can find the index.html under Labsetup\image_www\csp.

Lab tasks

In Firefox, open the following URLs from your VM.

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

Questions.

- **Question 4:** Describe and explain your observations when you visit these websites.
- **Question 5:** How will you change the server configuration on example32b (modify the Apache configuration), so Areas 5 and 6 display OK?
- **Question 6:** How will you change the server configuration on example32c (modify the PHP code), so Areas 1, 2, 4, 5, and 6 all display OK?

After lab

- Type Ctrl-C to stop servers running in the container console.
- At the Labsetup folder, run **dcdown** to shutdown the container properly
- Run “sudo rm -r mysql_data/” (optional) to clean up the database if you want to.
- Complete the post-lab assignment.