**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?**

In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. For example, this might be to change the email address on their account, to change their password, or to make a funds transfer. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. 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?**

For a CSRF attack to be possible, 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.

For example, suppose an application contains a function that lets the user change the email address on their account. When a user performs this action, they make an HTTP request like the following:

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.

With these conditions in place, the attacker can construct a web page containing the following 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>

If a victim user visits the attacker's web page, the following will happen:

* The attacker's page will trigger an HTTP request to the vulnerable web site.
* If the user is logged in to the vulnerable web site, their browser will automatically include their session cookie in the request (assuming [SameSite cookies](https://portswigger.net/web-security/csrf" \l "common-defences-against-csrf) are not being used).
* The vulnerable web site will process the request in the normal way, treat it as having been made by the victim user, and change their email address.

**Note**

Although CSRF is normally described in relation to cookie-based session handling, it also arises in other contexts where the application automatically adds some user credentials to requests, such as HTTP Basic authentication and certificate-based authentication.

**How to construct a CSRF attack**

Manually creating the HTML needed for a CSRF exploit can be cumbersome, particularly where the desired request contains a large number of parameters, or there are other quirks in the request. The easiest way to construct a CSRF exploit is using the [CSRF PoC generator](https://portswigger.net/burp/documentation/desktop/tools/engagement-tools/generate-csrf-poc) that is built in to [Burp Suite Professional](https://portswigger.net/burp/pro):

* Select a request anywhere in Burp Suite Professional that you want to test or exploit.
* From the right-click context menu, select Engagement tools / Generate CSRF PoC.
* Burp Suite will generate some HTML that will trigger the selected request (minus cookies, which will be added automatically by the victim's browser).
* You can tweak various options in the CSRF PoC generator to fine-tune aspects of the attack. You might need to do this in some unusual situations to deal with quirky features of requests.
* Copy the generated HTML into a web page, view it in a browser that is logged in to the vulnerable web site, and test whether the intended request is issued successfully and the desired action occurs.

**How to deliver a CSRF exploit**

The delivery mechanisms for cross-site request forgery attacks are essentially the same as for reflected XSS. Typically, 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.

Note that some simple 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. In the preceding example, if the request to change email address can be performed with the GET method, then a self-contained attack would look like this:

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

**What is the difference between XSS and CSRF?**

[Cross-site scripting](https://portswigger.net/web-security/cross-site-scripting) (or XSS) allows an attacker to execute arbitrary JavaScript within the browser of a victim user.

[Cross-site request forgery](https://portswigger.net/web-security/csrf) (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 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. Conversely, a successful 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.
* 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. Conversely, XSS is "two-way", in that the attacker's injected script can issue arbitrary requests, read the responses, and exfiltrate data to an external domain of the attacker's choosing.

**Can CSRF tokens prevent XSS attacks?**

Some XSS attacks can indeed be prevented through effective use of CSRF tokens. Consider a simple [reflected XSS](https://portswigger.net/web-security/cross-site-scripting/reflected) vulnerability that can be trivially exploited like this:

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

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. The clue here is in the name: "cross-site scripting", at least in its [reflected](https://portswigger.net/web-security/cross-site-scripting/reflected) form, involves a cross-site request. By preventing an attacker from forging a cross-site request, the application prevents trivial exploitation of the XSS vulnerability.

Some important caveats arise here:

* If a [reflected XSS](https://portswigger.net/web-security/cross-site-scripting/reflected) 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](https://portswigger.net/web-security/cross-site-scripting/stored) vulnerabilities. If a page that is protected by a CSRF token is also the output point for a [stored XSS](https://portswigger.net/web-security/cross-site-scripting/stored) vulnerability, then that XSS vulnerability can be exploited in the usual way, and the XSS payload will execute when a user visits the page.

**Common defences against CSRF**

Nowadays, 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. Since 2021, Chrome enforces Lax SameSite restrictions by default. As this is the proposed standard, we expect other major browsers to adopt this behavior in future.
* **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. This is generally less effective than CSRF token validation.

For a more detailed description of each of these defenses, as well as how they can potentially be bypassed, check out the following materials. These include interactive labs that let you practice what you've learned on realistic, deliberately vulnerable targets.

## 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, 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>

Submitting this form results in the following request:

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

When implemented correctly, CSRF tokens help protect against CSRF attacks by making it difficult for an attacker to construct a valid request on behalf of the victim. As the attacker has no way of predicting the correct value for the CSRF token, they won't be able to include it in the malicious request.

#### Note

CSRF tokens don't have to be sent as hidden parameters in a POST request. Some applications place CSRF tokens in HTTP headers, for example. The way in which tokens are transmitted has a significant impact on the security of a mechanism as a whole. For more information, see [How to prevent CSRF vulnerabilities](https://portswigger.net/web-security/csrf/preventing).

## Common flaws in CSRF token validation

CSRF vulnerabilities typically arise due to flawed validation of CSRF tokens. In this section, we'll cover some of the most common issues that enable attackers to bypass these defenses.

### 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](https://portswigger.net/web-security/csrf):

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 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](https://portswigger.net/web-security/csrf):

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 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.

### CSRF token is tied to a non-session cookie

In a variation on the preceding vulnerability, 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:

POST /email/change HTTP/1.1

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 68

Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF; csrfKey=rZHCnSzEp8dbI6atzagGoSYyqJqTz5dv

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

This situation is harder to exploit but is still vulnerable. If the web site contains any behavior that allows an attacker to set a cookie in a victim's browser, then an attack is possible. 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.

#### Note

The cookie-setting behavior does not even need to exist within the same web application as the [CSRF vulnerability](https://portswigger.net/web-security/csrf). Any other application within the same overall DNS domain can potentially be leveraged to set cookies in the application that is being targeted, if the cookie that is controlled has suitable scope. For example, a cookie-setting function on staging.demo.normal-website.com could be leveraged to place a cookie that is submitted to secure.normal-website.com.

### 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. This is sometimes called the "double submit" defense against CSRF, and is advocated because it is simple to implement and avoids the need for any server-side state:

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 again perform a CSRF attack if the web site contains any cookie setting functionality. Here, the attacker doesn't need to obtain a valid token of their own. They simply invent a token (perhaps in the required format, if that is being checked), 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.

# 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](https://portswigger.net/web-security/cors) exploits.

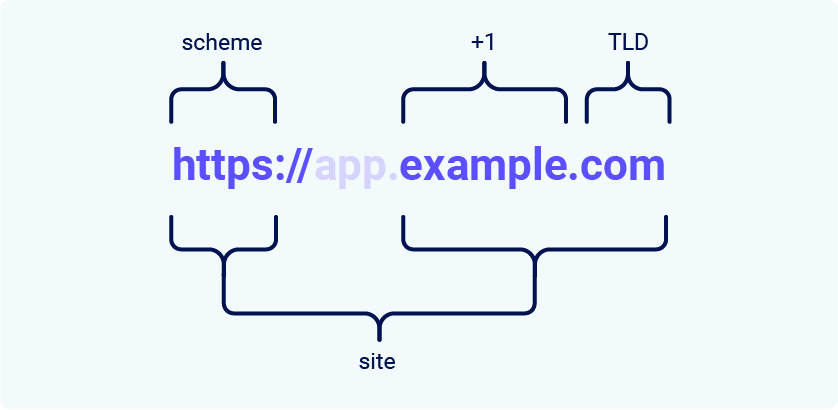
Since 2021, Chrome applies Lax SameSite restrictions by default if the website that issues the cookie doesn't explicitly set its own restriction level. This is a proposed standard, and we expect other major browsers to adopt this behavior in the future. As a result, it's essential to have solid grasp of how these restrictions work, as well as how they can potentially be bypassed, in order to thoroughly test for cross-site attack vectors.

In this section, we'll first cover how the SameSite mechanism works and clarify some of the related terminology. We'll then look at some of the most common ways you may be able to bypass these restrictions, enabling CSRF and other cross-site attacks on websites that may initially appear secure.

## What is a site in the context of SameSite cookies?

In the context of SameSite cookie restrictions, 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.



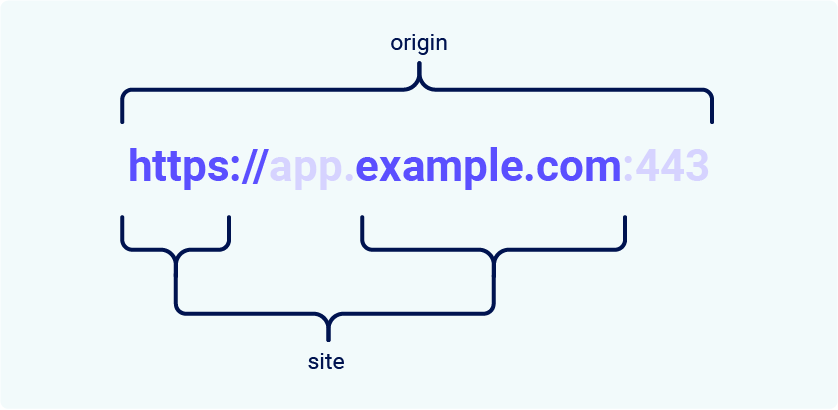
#### Note

You may come across the term "effective top-level domain" (eTLD). This is just a way of accounting for the reserved multipart suffixes that are treated as top-level domains in practice, such as .co.uk.

## 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.

Two URLs are considered to have the same origin if they share the exact same scheme, domain name, and port. Although note that the port is often inferred from the scheme.



As you can see from this example, the term "site" is much less specific as it only accounts for the scheme and last part of the domain name. Crucially, this means that a cross-origin request can still be same-site, but not the other way around.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

This is an important distinction as it means that any vulnerability enabling arbitrary JavaScript execution can be abused to bypass site-based defenses on other domains belonging to the same site. We'll see an example of this in one of the labs later.

## How does SameSite work?

Before the SameSite mechanism was introduced, browsers sent cookies in every request to the domain that issued them, even if the request was triggered by an unrelated third-party website. 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, which induce the victim's browser to issue a request that triggers a harmful action on the vulnerable website. As these requests typically require a cookie associated with the victim's authenticated session, the attack will fail if the browser doesn't include this.

All major browsers currently support the following SameSite restriction levels:

* [Strict](https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions#strict)
* [Lax](https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions#lax)
* [None](https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions#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, along with their preferred value:

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

Although this offers some protection against CSRF attacks, none of these restrictions provide guaranteed immunity, as we'll demonstrate using deliberately vulnerable, interactive labs later in this section.

#### Note

If the website issuing the cookie doesn't explicitly set a SameSite attribute, Chrome automatically applies Lax restrictions by default. This means that the cookie is only sent in cross-site requests that meet specific criteria, even though the developers never configured this behavior. As this is a proposed new standard, we expect other major browsers to adopt this behavior in future.

### Strict

If a cookie is set with the SameSite=Strict attribute, browsers will not send it in any cross-site requests. 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.

This is recommended when setting cookies that enable the bearer to modify data or perform other sensitive actions, such as accessing specific pages that are only available to authenticated users.

Although this is the most secure option, it can negatively impact the user experience in cases where cross-site functionality is desirable.

### 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.

This means that the cookie is not included in cross-site POST requests, for example. As POST requests are generally used to perform actions that modify data or state (at least according to best practice), they are much more likely to be the target of CSRF attacks.

Likewise, the cookie is not included in background requests, such as those initiated by scripts, iframes, or references to images and other resources.

#### 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.

With the exception of Chrome, this is the default behavior used by major browsers if no SameSite attribute is provided when setting the cookie.

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. Tracking cookies are a typical example.

If you encounter a cookie set with SameSite=None or with no explicit restrictions, it's worth investigating whether it's of any use. When the "Lax-by-default" behavior was first adopted by Chrome, this had the side-effect of breaking a lot of existing web functionality. As a quick workaround, some websites have opted to simply disable SameSite restrictions on all cookies, including potentially sensitive ones.

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.

Set-Cookie: trackingId=0F8tgdOhi9ynR1M9wa3ODa; SameSite=None; Secure

## Bypassing SameSite Lax restrictions using GET requests

In practice, servers aren't always fussy about whether they receive a GET or POST request to a given endpoint, even those that are expecting a form submission. If they also use Lax restrictions for their session cookies, either explicitly or due to the browser default, you may still be able to perform a [CSRF attack](https://portswigger.net/web-security/csrf) by eliciting a GET request from the victim's browser.

As long as the request involves a top-level navigation, the browser will still include the victim's session cookie. The following is one of the simplest approaches to launching such an attack:

<script>

document.location = 'https://vulnerable-website.com/account/transfer-payment?recipient=hacker&amount=1000000';

</script>

Even 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">

<input type="hidden" name="\_method" value="GET">

<input type="hidden" name="recipient" value="hacker">

<input type="hidden" name="amount" value="1000000">

</form>

Other frameworks support a variety of similar parameters.

## 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.

One possible gadget is a client-side redirect that dynamically constructs the redirection target using attacker-controllable input like URL parameters. For some examples, see our materials on [DOM-based open redirection](https://portswigger.net/web-security/dom-based/open-redirection).

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.

If you can manipulate this gadget to elicit a malicious secondary request, this can enable you to bypass any SameSite cookie restrictions completely.

Note that the equivalent attack is not possible with server-side redirects. In this case, browsers recognize that the request to follow the redirect resulted from a cross-site request initially, so they still apply the appropriate cookie restrictions.

## 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.

Make sure you thoroughly audit all of the available attack surface, including any sibling domains. In particular, vulnerabilities that enable you to elicit an arbitrary secondary request, such as [XSS](https://portswigger.net/web-security/cross-site-scripting), can compromise site-based defenses completely, exposing all of the site's domains to cross-site attacks.

In addition to classic CSRF, don't forget that if the target website supports [WebSockets](https://portswigger.net/web-security/websockets), this functionality might be vulnerable to [cross-site WebSocket hijacking](https://portswigger.net/web-security/websockets/cross-site-websocket-hijacking) ([CSWSH](https://portswigger.net/web-security/websockets/cross-site-websocket-hijacking)), which is essentially just a [CSRF attack](https://portswigger.net/web-security/csrf) targeting a WebSocket handshake. For more details, see our topic on [WebSocket vulnerabilities](https://portswigger.net/web-security/websockets).

## 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.

As mentioned earlier, if a website doesn't include a SameSite attribute when setting a cookie, Chrome automatically applies Lax restrictions by default. However, to avoid breaking single sign-on (SSO) mechanisms, it doesn't actually enforce these restrictions for the first 120 seconds on top-level POST requests. As a result, there is a two-minute window in which users may be susceptible to cross-site attacks.

#### Note

This two-minute window does not apply to cookies that were explicitly set with the SameSite=Lax attribute.

It's somewhat impractical to try timing the attack to fall within this short window. On the other hand, if you can find a gadget on the site that enables you to force the victim to be issued a new session cookie, you can preemptively refresh their cookie before following up with the main attack. For example, completing an OAuth-based login flow may result in a new session each time as the OAuth service doesn't necessarily know whether the user is still logged in to the target site.

To trigger the cookie refresh without the victim having to manually log in again, you need to use a top-level navigation, which ensures that the cookies associated with their current [OAuth](https://portswigger.net/web-security/oauth) session are included. This poses an additional challenge because you then need to redirect the user back to your site so that you can launch the CSRF attack.

Alternatively, you can trigger the cookie refresh from a new tab so the browser doesn't leave the page before you're able to deliver the final attack. A minor snag with this approach is that browsers block popup tabs unless they're opened via a manual interaction. For example, the following popup will be blocked by the browser by default:

window.open('https://vulnerable-website.com/login/sso');

To get around this, you can wrap the statement in an onclick event handler as follows:

window.onclick = () => {

window.open('https://vulnerable-website.com/login/sso');

}

This way, the window.open() method is only invoked when the user clicks somewhere on the page.

# Bypassing Referer-based CSRF defenses

Aside from defenses that employ CSRF tokens, 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. It is generally added automatically by browsers when a user triggers an HTTP request, including by clicking a link or submitting a form. Various methods exist that allow the linking page to withhold or modify the value of the Referer header. This is often done for privacy reasons.

## 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](https://portswigger.net/web-security/csrf) in a way that causes the victim user's browser to drop the Referer header in the resulting request. There are various ways to achieve this, but the easiest is using a META tag within the HTML page that hosts the [CSRF attack](https://portswigger.net/web-security/csrf):

<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:

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

Likewise, 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

#### Note

Although you may be able to identify this behavior using Burp, you will often find that this approach no longer works when you go to test your proof-of-concept in a browser. In an attempt to reduce the risk of sensitive data being leaked in this way, many browsers now strip the query string from the Referer header by default.

You can override this behavior by making sure that the response containing your exploit has the Referrer-Policy: unsafe-url header set (note that Referrer is spelled correctly in this case, just to make sure you're paying attention!). This ensures that the full URL will be sent, including the query string.

# How to prevent CSRF vulnerabilities

In this section, we'll provide some high-level guidance on how you can protect your own websites from the kinds of vulnerabilities we've demonstrated in our [CSRF](https://portswigger.net/web-security/csrf) labs.

## Use CSRF tokens

The 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.

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

If you need further assurance beyond the strength of the CSPRNG, you can generate individual tokens by concatenating its output with some user-specific entropy and take a strong hash of the whole structure. This presents an additional barrier to an attacker who attempts to analyze the tokens based on a sample that are issued to them.

### 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. The token will then be included as a request parameter when the form is submitted:

<input type="hidden" name="csrf-token" value="CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz" />

For additional safety, the field containing the CSRF token should be placed as early as possible within the HTML document, ideally before any non-hidden input fields and before any locations where user-controllable data is embedded within the HTML. This mitigates against various techniques in which an attacker can use crafted data to manipulate the HTML document and capture parts of its contents.

An alternative approach, of placing the token into the URL query string, is somewhat less safe because the query string:

* Is logged in various locations on the client and server side;
* Is liable to be transmitted to third parties within the HTTP Referer header; and
* can be displayed on-screen within the user's browser.

Some applications transmit CSRF tokens within a custom request header. This presents a further defense against an attacker who manages to predict or capture another user's token, because browsers do not normally allow custom headers to be sent cross-domain. However, the approach limits the application to making CSRF-protected requests using XHR (as opposed to HTML forms) and might be deemed over-complicated for many situations.

CSRF tokens should not be transmitted within cookies.

### 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. If the request does not contain any token at all, it should be rejected in the same way as when an invalid token is present.

## Use Strict SameSite cookie restrictions

In addition to implementing robust CSRF token validation, we recommend explicitly setting your own [SameSite](https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions) 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.

Even if all browsers eventually adopt the "Lax-by-default" policy, this isn't suitable for every cookie and can be more easily bypassed than Strict restrictions. In the meantime, the inconsistency between different browsers also means that only a subset of your users will benefit from any SameSite protections at all.

Ideally, 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.

If possible, we recommend isolating insecure content, such as user-uploaded files, on a separate site from any sensitive functionality or data. When testing a site, be sure to thoroughly audit all of the available attack surface belonging to the same site, including any of its sibling domains.